Data Structures Java Exam

Do not modify the interface, the package, or **anything from the given resources**. In **Judge,** you only **upload the archive of the corresponding package**.

1. Race Manager

You are given a skeleton with a class **`RaceManagerImpl`** that implements the **`RaceManager`** interface.

The **RaceManagerImpl** works with **athletes**. Each **`Athlete`** holds a **name** and **age**.

Implement all the operations from the **interface**:

- **`void enroll(Athlete athlete)`** – adds a participant for our race. If this Athlete **is already enrolled**, throw **`IllegalArgumentException`**. We should store Athletes in such a way that when we use **`start()`** the first enrolled athlete will be the first to start, the second enrolled athlete should be the second to start, and so on.
- **`public boolean isEnrolled(Athlete athlete)`** - returns **true** if the given athlete is **already enrolled** in the race, otherwise returns **false**.
- **`void start()`** – this lets the next athlete (in terms of enrolment order) start the course. If there are no Athletes waiting to start throw **`IllegalArgumentException.`**
- **`void retire(Athlete athlete)`** – There is some issue with this athlete's race and they won't be able to complete it. If this athlete has **never started the race** - throw **`IllegalArgumentException`**.
- **`void finish(Athlete athlete)`** - This athlete successfully completed the race. If the Athlete **did never start** - throw **`IllegalArgumentException.`** All athletes who successfully finished the race should be kept in the **order of finishing**.
- **`Athlete getLastFinishedAthlete()`** – returns the **last athlete** to cross the finish line. If there are **no finished Athletes** - throw **`IllegalArgumentException`**
- **`int currentRacingCount()`** – return the number of athletes that are currently racing on the course. **Return 0 if there are no athletes**, **no started athletes** or **all athletes have finished**.
- **`Collection<Athlete> getAllAthletesByAge()`** – returns all enrolled athletes **ordered by their age** youngest to oldest. If **there aren't any** - return an **empty collection**. The method will check **all enrolled athletes**, not only those who have started in the race.
- **`Collection<Athlete> getAllNotFinishedAthletes()`** – return only the athletes that never finished. This includes athletes who did not start or retired. **Sort** the athletes by their **names alphabetically**.
- **`Iterator<Athlete> getScoreBoard()`** – We want to show a scoreboard at the finish line presenting finished athletes from the **most recent finisher to the first finisher.**

2. Race Manager – Performance

For this task, you will only be required to submit **the code from the previous problem**.

- If you are having a problem with this task, you should **perform a 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: enroll, start, etc.
- You can submit code to this problem **without full coverage** from the previous problem, **not all test cases will be considered**, only the **general behavior** will be important, and **edge cases** will mostly be ignored such as throwing exceptions, etc.

3. Storage Service

You are given a skeleton with a class `StorageServiceImpl` that implements the `StorageService` interface. The `StorageService` works with `StorageUnit` & **Box entities**. **The storage units** are identified by their **id** and keep track of their **total available space** (this is the **initial capacity** and should not change) and **total used space** (the **sum of all boxes currently stored**). The **boxes** are also **identified by id** and have measurements to calculate their volume when storing. We can assume there is no space lost when we put boxes inside the storage units – we'll calculate everything based on the box's volume and the available space within a storage unit.

Implement all the operations from the **interface**:

- **`void rentStorage(StorageUnit unit)`** – adds another **`StorageUnit`** in which we can store boxes. If there is a storage unit with the **same id added before**, throw `IllegalArgumentException`.
- **`void storeBox(Box box)`** – stores a **Box** inside the unit with the most free space available. If the box was **already stored in any `StorageUnit`** - throw `IllegalArgumentException`. If the box is bigger than the available space inside the **`StorageUnit`** throw `IllegalArgumentException`. If there are **no storage units available** - throw `IllegalArgumentException`.
- **`boolean isStored(Box box)`** – returns whether the **Box** has been **stored or not**.
- **`boolean isRented(StorageUnit unit)`** – returns whether we have rented this **`StorageUnit`** or not.
- **`boolean contains(StorageUnit unit, String boxId)`** – returns if this **`StorageUnit` contains** the **Box** with the provided **id**. Be careful with the **validation of the parameters** passed to the test.
- **`Box retrieve(StorageUnit unit, String boxId)`** – try to retrieve the **Box** with the **provided id** from the provided **`StorageUnit`**. If the **box was never stored** or it is **not in this StorageUnit** throw `IllegalArgumentException`. Otherwise, remove the box from the **`StorageUnit`**, and the space is returned as free.
- **`int getTotalFreeSpace()`** – returns the **free space left** across all storage units.
- **`StorageUnit getMostAvailableSpaceUnit()`** – return the **`StorageUnit`** with **most available space**. If there are no storage units added throw **`IllegalArgumentException.`**
- **`Collection<Box> getAllBoxesByVolume()`** – return **all stored boxes** ordered by their volume smallest to largest. If there are **boxes with the same volume**, **order** them **by height from longest to shortest**. If there are **no stored boxes return an empty Collection**.
- **`Collection<StorageUnit> getAllUnitsByFillRate()`** – return **all storage units** ordered **by the percentage of space available** (highest to lowest). If there are storage units with the **same availability** - order them **by total space** (highest to lowest) they offer (not free space but total space available). If there are **no units**, return an **empty Collection**. You can calculate the **percentage of space available by dividing the free space of a unit by the total space** that the unit offers. You **don't need** to work with **floating point numbers** for that calculation.

4. Storage Service – Performance

For this task, you will only be required to submit the **code from the previous problem**.

- If you are having a problem with this task, you should **perform a 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: **rentStorage**, **storeBox,** etc.
- You can submit code to this problem **without full coverage** from the previous problem, **not all test cases** will be considered, only the **general behavior** will be important, and **edge cases** will mostly be ignored such as throwing exceptions, etc.