Data Structures Java Exam

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

1. Renovation

You are given a skeleton with a class **RenovationImpl** that implements the **Renovation interface**.

The **RenovationImpl** works with **Laminate** & **Tile entities**. Implements all the operations from the **interface** by following basic renovation principles. When delivering items, we always put them on top of the previous item. Once we need Tiles or Laminate, we take a box from the top of the pile. You can assume all **measurements are in meters**.

- `void deliverTile(Tile tile)` - **adds a tile**. We don't need more than 30 sq. m. of tiles. If we're going to receive more than that we should throw an `IllegalArgumentException`. An exception is thrown when the total delivered tile area **exceeds 30 sq. m.**, regardless of whether the tiles are delivered one by one or in a single delivery.
- `void deliverFlooring(Laminate laminate)` - **adds a laminate** to the pile.
- `double getDeliveredTileArea()` - **returns the total area** of tiles delivered.
- `boolean isDelivered(Laminate laminate)` - **returns** whether the **Laminate has been already delivered**.
- `void returnTile(Tile tile)` - we've found an issue with our tiles and they should be returned. The implementation should verify that the item was previously delivered. If the **tile was never delivered** – throw an `IllegalArgumentException`. Items are **checked one by one** and removed from the collection that **preserves the order of delivery**. This collection should operate in way, ensuring that the **last item delivered** is the **first to be checked**. The **last item checked**, should be the **first to be returned** in the collection.
- `void returnLaminate(Laminate laminate)` - we've found an issue with our laminate and they should be returned. The implementation should verify that the item was previously delivered. If the **laminate was never delivered** – throw an `IllegalArgumentException`. Items are **checked one by one** and removed from the collection that **preserves the order of delivery**. This collection should operate in way, ensuring that the **last item delivered** is the **first to be checked**. The **last item checked**, should be the **first to be returned** in the collection.
- `Collection<Laminate> getAllByWoodType(WoodType wood)` – **returns** all **laminates made from the specified wood type.** If there aren't any - return an **empty collection**.
- `Collection<Tile> getAllTilesFitting(double width, double height)` - **return** only **tiles that fit the specified dimensions**(inclusive).
- `Collection<Tile> sortTilesBySize()` - **return all tiles** ordered by their area ascending. If their area is equal, sort them by depth ascending. If there are **no tiles**, return an **empty collection**.
- `Iterator<Laminate> layFlooring()` – it's time to lay the flooring. Returns an **iterator** on all **delivered** (and **not returned**) laminates starting from the **most recently delivered** one and going to the **first delivery**.

2. Renovation – 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: **deliverTile, deliverFlooring**, 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. Craftsman Lab

You are given a skeleton with a class **CraftsmanLabImpl** that implements the **CraftsmanLab interface**. The CraftsmanLab works with **ApartmentRenovation** & **Craftsman entities**, **all apartments** are identified by their **addresses** and all **craftsmen** by their **name**. Implement all the operations from the **interface**:

- **`void addAraptment(ApartmentRenovation job)`** - **adds** an apartment for renovation. If there is an **apartment with the same address** added before, throw an **`IllegalArgumentException`**.
- **`void addCraftsman(Craftsman craftsman)`** - **adds a craftsman**. If the **craftsman already exists** - throw **`IllegalArgumentException.`**
- **`boolean exist(ApartmentRenovation job)`** - **returns** whether the **ApartmentRenovation** has been **added** or **not**.
- **`boolean exist(Craftsman c)`** – **returns** whether the **Craftsman** has been **added** or **not**.
- **`void removeCraftsman(Craftsman c)`** – **remove** the provided **craftsman**. If this craftsman is missing or is already assigned to do a renovation, throw an **`IllegalArgumentException.`**
- **`Collection<Craftsman> getAllCraftsmen()`** - **return** a collection of **all craftsmen** (**that were not removed**). If there are not any - return an **empty collection**.
- **`void assignRenovations()`** - **assign** each **ApartmentRenovation** to a **Craftsman** following this logic - go over the apartments in the order **they were added**, and give each apartment to the craftsman with the lowest **totalEarnings** so far. Keep in mind to recalculate the total earnings using the **workHoursNeeded** and **hourlyRate** fields after each assignment. If an apartment already has an assigned craftsman **do not** reassign it. It's important to note that **only one craftsman can be assigned to work on a specific apartment** renovation. If a craftsman is already assigned to a renovation, they **cannot be simultaneously assigned to another renovation**. This ensures that each renovation project has a dedicated craftsman, and craftsmen do not work on multiple renovations simultaneously.
- **`Craftsman getContractor(ApartmentRenovation job)`** - **return** who is the **craftsman doing this renovation**. If nobody is assigned to this apartment throw an **`IllegalArgumentException.`**
- **`Craftsman getLeastProfitable()`** – **return** the least profitable **Craftsman**. If there are **no craftsmen** added throw an **`IllegalArgumentException.`**
- **`Collection<ApartmentRenovation> getApartmentsByRenovationCost()`** - **return all apartments** ordered by the cost of the renovation (use the same calculation as in **assignRenovation**). If an apartment **does not have** an assigned craftsman assume the cost is the number of hours required to perform the renovation. The **most expensive** apartment renovation should **be first**.
- **`Collection<ApartmentRenovation> getMostUrgenRenovations(int count)`** - **return** the **count** most urgent renovations (order by their **deadline**). If there are less than **count** renovations - **return all**.

4. Craftsman Lab – 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: **addAprtment, addCraftsman**, 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.