Data Structures – Advanced – Exam Preparation

This document defines the examination example problems for "Data Structures – Advanced (Java)" course @ Software University.

Please submit your solutions (source code) of all below described problems in Judge.

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**.

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 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 tests logic. **Do not change the packages** or move any of the files provided inside the skeleton if you have to add 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 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, for example **equals ()** and **hashCode ()** etc…

1. Microsystem

You have to implement a structure that keeps track of the Microsystem store for computers. Your structure will have to support the following functionalities:

- `createComputer(computer)` – you have to create a new computer and add it to the store. If there is already a computer with that number, throw `IllegalArgumentException`
- `contains(int number)` – checks if a computer with the provided number exists in the store.
- `count()` – returns the count of computers in the store
- `getComputer(number)` – returns the computer with the given number. If there isn't such throw `IllegalArgumentException`.

- **`remove(int number)` –** removes the computer with the provided number. If there isn't such throw **`IllegalArgumentException`**
- **`removeWithBrand(brand)` –** removes all computers with the given brand. If there aren't any throw **`IllegalArgumentException`**
- **`upgradeRam(ram, number)` –** finds the computer with the given number and sets its ram to the given one (only if the given one is bigger). If there isn't a computer with the provided number throw **`IllegalArgumentException`**
- **`getAllFromBrand(brand)` –** finds all computers with the provided brand. Order them by price descending. If there aren't any return empty collection.
- **`getAllWithScreenSize(screenSize)` –** finds all computers with screen size equal to the given. Order them by number descending. If there aren't any return empty collection.
- **`getAllWithColor(color)` –** finds all computers with the same color as the given. Order them by price descending. If there aren't any return empty collection.
- **`getInRangePrice(minPrice, maxPrice)` –** finds all computers with price between the given inclusive. Order them by price descending. If there aren't any return empty collection.

1.5 Performance Tests

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

2. VANI Planning

Your task is to implement a simple system for invoice processing.

You have a class Invoice which has the following properties:

- **`String serialNumber`** – unique number for each invoice
- **`String companyName`** – the invoice company name
- **`Double subtotal` –** the subtotal of the invoice
- **`Enum department`** – the department to which the invoice belongs
- **`LocalDate issueDate`** – the date when the invoice was created
- **`LocalDate dueDate`** – the deadline for the invoice

You need to support the following operations (and they should be **fast**):

- **`create(Invoice)`**– Add the invoice to the agency archive. You will need to implement the **`contains()`** method as well. If the invoice number already exists throw **`IllegalArgumentException`**
- **`contains(number)`** – checks if an invoice with the given number is present in the archive
- **`count`** – returns the number of invoices in the archive
- **`payInvoice(dueDate)`** – find all invoices whose due date is exactly the given one and set their subtotal to 0. If there aren't any throw **`IllegalArgumentException`**.
- **`throwInvoice(SerialNumber)`** – remove the invoice with the given number. If the invoice doesn't exist throw **`IllegalArgumentException`**
- **`throwPayed()`** – remove all invoices which were payed

- `getAllInvoiceInPeriod(start, end)` – find all invoices which were created in the given period inclusive. Order them by date of creating ascending, then by due date as second parameter ascending. If there aren't any return empty collection.
- `searchBySerialNumber (String SerialNumber)` – return all invoices whose number contains the given one. If there aren't any throw `IllegalArgumentException`
- `throwInvoiceInPeriod(start, end)` – remove all invoices which have due date between the given range exclusive. If there aren't any throw `IllegalArgumentException`.
- `getAllFromDepartment(department)` – finds all invoices in the provided department. Order them by subtotal descending as first parameter, then by creating date ascending. If there aren't any return empty collection.
- `getAllByCompany(company)` – finds all invoices created by the given company. Order them by their number descending. If there aren't any return empty collection.
- `extendDeadline(dueDate, days)` – find all invoices whose due date is equal to the given one and extend their due date by the days given. If there aren't any throw `IllegalArgumentException`.

2.5 Performance Tests

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

"I'll be more enthusiastic about encouraging thinking outside the box when there's evidence of any thinking going on inside it." ― **Terry Pratchett**