**Data Structures – Advanced – Data Structures Augmentation – Lab**

This document defines the examination example problems for ["Data Structures – Advanced (Java)" course @ Software University](https://softuni.bg/trainings/3924/data-structures-advanced-with-java-december-2022). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2435/07-Data-Structures-Augmentation-Lab).

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**.

# Person Collection

We want to implement a **"collection of persons" data structure** that performs fast enough the following operations:

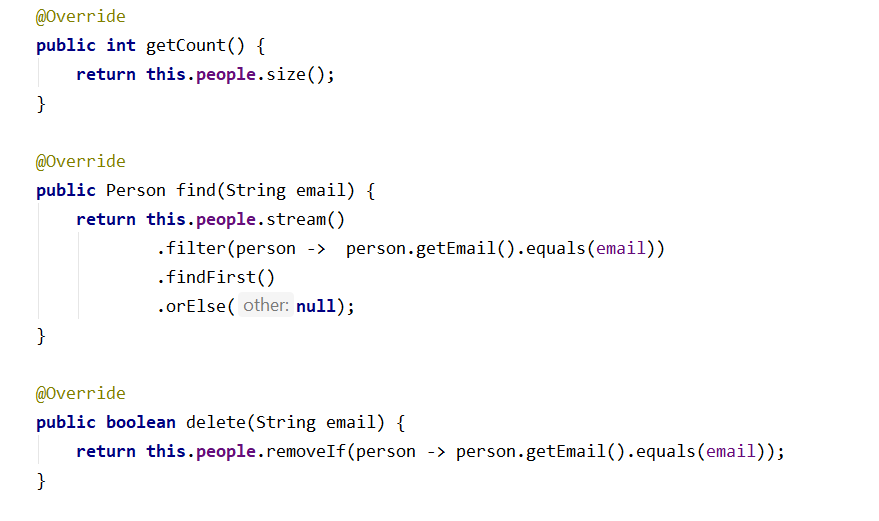
* **add(*email*, *name*, *age*, *town*)**
  + The *email* is unique (it uniquely identities the person)
  + If the *email* already exists returns **false** (without adding the person), otherwise return **true**
* **find(*email*)**
  + Returns the **Person** object or **null** (if it does not exits)
* **delete(*email*)**
  + Returns **true** (successfully deleted) or **false** (not found)
* **findAll(*email\_domain*)**
  + Returns a sequence of matched persons sorted by *email*
* **findAll(*name*, *town*)**
  + Returns a sequence of matched persons sorted by *email*
* **findAll(*start\_age*, *end\_age*)**
  + Returns a sequence of matched persons **(ranges are inclusive)** sorted by *age*, then by *email* (as second criteria)
* **findAll(*start\_age*, *end\_age*, *town*)**
  + Returns a sequence of matched persons **(ranges for age are inclusive)** sorted by *age*, then by *email* (as second criteria)

## Start with a Straightforward (and Slow) Solution

First, let**'**s start with a **simple, straightforward (and slow) solution** – implement the **"**person collection**"** data structure as List<Person>.

The **finder methods** could be implemented by **using Stream API queries** with straightforward **filtering** and **sorting** using lambda expressions. First, let’s define the – add(…) method It just **creates a new person** and **appends it to the underlying list** and should also checks for existing person.

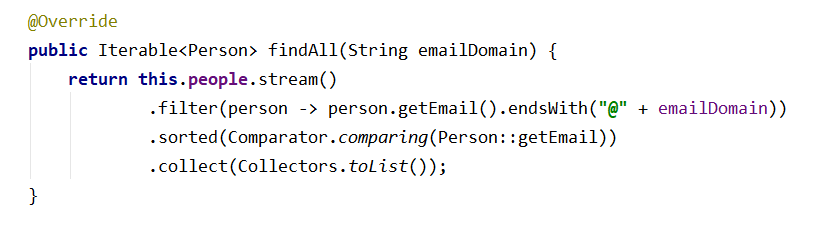
Next, let**'**s add the getCount(), find(email) and delete(email) methods that work over the underlying list-based collection:



## Finder Methods in the Slow Solution

Now, let**'**s continue by implementing the **finder methods** in the slow list-based implementation.

First, implement **finding people by email domain (the domain is everything after the '@' char or ASCII 64)**. The results should be sorted by email



Similarly, implement **the other finder methods**. Finding people by name and town works the same way like finding by email domain.

Finding people **by age range** works by simple filtering the underlying list of persons, then by applying the requested **sorting order** (sort first **by age**, then **by email** as second criteria).

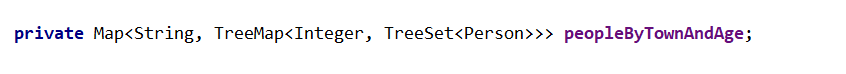
Finally, we implement **finding by age range and town**, just like the previous finder methods.

## More Efficient Underlying Data Structures

Now let’s implement an **improved solution**, which uses **more efficient underlying data structures**.

Let’s first **define the data structures** needed to perform efficiently the required operations:

* To **find a person by email** we can use a **hash-table**. We expect zero or one person to match given email address (recall that the *email* is unique)
* To find all **people matching given email address domain**, we can use a **hash-table**. It will use the email domain as **key** and a sorted set of people as **value.** Note that people for each email domain are **sorted internally by email**. In order this to work correctly, the Person class should implement Comparable<Person> and compare persons by their email
* To **find all** **people by name and town** we can use a hash-table. We can **combine the name + town** as a single string value and use it as **key** and use sorted set of people as value
* **Finding all people by age range** needs a data structure that uses the **age as key** and keeps the **ages sorted**. The **values** could be a sorted set of people (sorted by email). In this structure, when we look for all people by range of ages, we will get all distinct ages in increasing order and for each age we will get a set of people sorted by email.
* Finding in a fast way all **people matching certain town and certain range of ages** is more complicated. We need **double mapping**: first **map towns** to some structure, which **maps ages** to **sets of persons**. For example, we could have a Map that maps towns to TreeMaps<>:



In the above structure, when we look for all people by town and a range of ages, we will first lookup to find the dictionary of ages for given town, then will get all distinct ages in the specified range in increasing order and finally for each age we will get a set of people sorted by email.

We have **five separate data structures** that work together to implement efficiently the operations from the **"**person collection**"** data structure. When we use a **combination of data structures** we need to always keep all underlying data structures up to date:

* **Add** needs to add the new data to all underlying data structures.
* **Modify** needs to update all the underlying data structures to hold correct data.
* **Delete** needs to delete the data from all underlying data structures.

You are ready to implement the operations from the PersonCollection interface based on the above underlying data structures. You have two different classes for those implementations the slow and the fast one so you can do some testing on your own. Let’s do it. It should be really not that complicated since all you have to do is call the correct delegation of operations to the data structures in usage.

# Person Collection – Performance

Submit the code from the previous task and see how the tests will handle the both solutions you implemented.

# Shopping Center

A **shopping center** keeps a set of **products**. Each product has **name**, **price** and **producer**. Your task is to model the shopping center and design a **data structure holding the products**. Write a program that executes **N** commands, given in the input (a single command at a line):

* AddProduct name;price;producer– adds a product by given name, price and producer. If a product with the same name / producer/ price already exists, the newly added product does not affect the existing ones (duplicates are allowed). As a result the command prints **"Product added"**.
* DeleteProducts producer– deletes all products matching given producer. As a result the command prints **"X products deleted"** where **X** is the number of deleted products or **"No products found"** if no such products exist.
* DeleteProducts name;producer – deletes all products matching given product name and producer. As a result the command prints **"X products deleted"** where **X** is the number of deleted products or **"No products found"** if no such products exist.
* FindProductsByName name – finds all products by given product name. As a result the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. Print each product on a separate line. If no products exist with the specified name, the command prints **"No products found"**.
* FindProductsByProducer producer– finds all products by given producer. As a result the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. You should print each product on a single line**.** If no products exist by the specified producer, the command prints **"No products found"**.
* FindProductsByPriceRange fromPrice;toPrice – finds all products whose price is greater or equal than **fromPrice** and less or equal than **toPrice**. As a result the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. You should print each product on a separate line. If no products exist within the specified price range, the command prints **"No products found"**.

All string matching operations are **case-sensetive**. This task **uses** **input and output** for testing.

### Input

The input data should be read from the console.

* At the first line you will be given the number **N** of the commands.
* At each of the next **N** lines you will be given a command in the format described above.

The input data will always be valid and in the described format. There is no need to check it explicitly.

### Output

The output data should be printed on the console.

The output should contain the output from each command from the input.

### Constraints

* **N** will be between 1 and 50 000, inclusive.
* All strings specified in the commands (e.g. product names and producers) consist of alphabetical characters, numbers and spaces. Strings are case-sensitive.
* Prices are given as real numbers with up to 2 digits after the decimal point, (e.g. 133.58, 320.3, or 10)
* The **'**.**'** symbol is used as a decimal separator.
* Prices should be printed with exactly **2 digits** after the decimal point (e.g. 320.30 instead of 320.3).
* Allowed working time for your program: **1500 MS** (at the judge environment).
* Allowed memory: **32 MB**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 17  AddProduct IdeaPad Z560;1536.50;Lenovo  AddProduct ThinkPad T410;3000;Lenovo  AddProduct VAIO Z13;4099.99;Sony  AddProduct CLS 63 AMG;200000;Mercedes  FindProductsByName CLS 63 AMG  FindProductsByName CLS 63  FindProductsByName cls 63 amg  AddProduct 320i;10000;BMW  FindProductsByName 320i  AddProduct G560;999;Lenovo  FindProductsByProducer Lenovo  DeleteProducts Lenovo  FindProductsByProducer Lenovo  FindProductsByPriceRange 100000;200000  DeleteProducts Beer;Ariana  DeleteProducts CLS 63 AMG;Mercedes  FindProductsByName CLS 63 AMG | Product added  Product added  Product added  Product added  {CLS 63 AMG;Mercedes;200000.00}  No products found  No products found  Product added  {320i;BMW;10000.00}  Product added  {G560;Lenovo;999.00}  {IdeaPad Z560;Lenovo;1536.50}  {ThinkPad T410;Lenovo;3000.00}  3 products deleted  No products found  {CLS 63 AMG;Mercedes;200000.00}  No products found  1 products deleted  No products found |

*"*Curiosity and the urge to solve problems are the emotional hallmarks of our species.*"*

**Carl Sagan**