# Exercises: Combining Data Structures

Problems for exercises and homework for the "Data Structures and Algorithms Advanced" course from the official "Applied Programmer" curriculum.

You can check your solution here: <https://judge.softuni.org/Contests/3602/Combining-Data-Structures-Exercises>

Use the provided skeleton:

Graphical user interface, text, application

Description automatically generated

## People Collection

* bool AddPerson**(string email, string name, int age, string town)**
  + The email is **unique**.
  + If the email already exists returns **false** (without adding the person), otherwise return **true.**
* Person FindPerson**(string email)**
  + Returns the **Person** object or **null** (if it does not exist).
* Bool DeletePerson**(string email)**
  + Returns **true** (successfully deleted) or **false** (not found).
* IEnumerable<Person> FindPeople**(string emailDomain)**
  + Returns a sequence of matched persons sorted by email.
* IEnumerable<Person> FindPeople**(string name, string town)**
  + Returns a sequence of matched persons sorted by email.

First, let's take a look at the Person class. It simply has a few **properties** and a **constructor** that a person can have:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

The next class PersonCollection holds **unfinished** implementations of the methods for the structure:

Graphical user interface, application

Description automatically generated

The project comes also with **unit tests** and **performance tests** covering the functionality of the “**person collection**” **data structure.**

Text

Description automatically generated

Let’s implement a **solution**, which uses **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 **dictionary**. We expect zero or one person to match given email address (recall that the email is unique):



* To find all **persons matching given email address domain**, we can use a **dictionary**. It will use the **email domain** as **key** and a **sorted set of persons** as **value**:



* To **find all** **persons 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 persons** as **value**:



We have **three 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**.

We are ready to implement the operations based on the above underlying data structures. Let’s do it.

Don't forget to **initialize** your structures:

Graphical user interface, text, timeline

Description automatically generated

First, we will implement **add**, **edit,** and **delete**.

Before we do that let us discuss the **extension methods,** we will need to **simplify working with dictionaries holding sets of values**. These methods are **generic** because we want to use them for any kind of dictionaries holding collections of values.

**Extension methods in C#** provide a special syntax for attaching methods to a class without changing or inheriting it. **Generic methods in C#** allow data types of the input and output method parameters to be generic – data types become parameters. Combining extension methods with generics can extend the C# dictionaries (IDictionary interface) and add very helpful methods to them that **save a lot of code**.

Look at the DictionaryExtensions class in your main project. It provides several extension methods for dictionaries.

The method AppendValueToKey(key, value) **adds a value to the collection of keys mapped to certain key**. If the key does not exist, a collection is first created:

Graphical user interface, text, application

Description automatically generated

Another very helpful extension method for dictionaries holding collection of values is GetValuesForKey(key). It returns **all values for certain dictionary key**. It assumes the key holds a collection of values or does not exist. When the key does not exist, am empty sequence of values is retuned:

Graphical user interface, text, application

Description automatically generated

### AddPerson()

Let’s write the AddPerson() method. We should return false if we already **have this email address**:

Graphical user interface, text, application, email

Description automatically generated

If we don't, we **create** and **add** a new person:

Graphical user interface, text, application, email

Description automatically generated

We can take the email domain with a **split** and fill our other structures too:

Text

Description automatically generated

### Count

Next, let’s implement the Count **property** to **get the count of people in any collection** (it should be the same). It is trivial:

Text

Description automatically generated

### FindPerson()

Next comes the FindPerson(email) method, a simple method where we use a TryGetValue() **method** because the **given email** **could not exist** in our dictionary:

Graphical user interface, text, application

Description automatically generated

### DeletePerson()

Next, implement the DeletePerson(email) method. It starts similar to our AddPerson() method. Try to find the person and if you can't, return false:

Graphical user interface, text, application

Description automatically generated

After that we **remove it by email**, **find the domain**, **remove by domain**, and **finally remove by town**:

Text

Description automatically generated

Try to **run our unit tests** – some succeed, some fail:

Text

Description automatically generated

Alright, let's make all tests run!

### FindPeople() by Domain

Now let’s implement the next operation: FindPeople**(emailDomain)**.

The key emailDomain is guaranteed to exist in the personsByEmailDomain dictionary, so we don’t need an additional check. The key is guaranteed to be created when the person is added by the AddPerson() method.

Implementing this method is very similar to our other **find method**, this time we can use our **dictionary extensions**:

Graphical user interface, text, application

Description automatically generated

**Run the unit tests** again:

Text

Description automatically generated

One more test succeeds!

### FindPeople() by Name and Town

Now let’s implement the FindPeople**(name, town)** **method**. We use the same GetValuesForKey() extension method, but we **combine the name and town** to get a **combined key**. This key is used to lookup in the underlying dictionary personsByNameAndTown that maps {name + town} to set of people.

Graphical user interface, text, application

Description automatically generated

Run your tests, **all of them should succeed**:

Text

Description automatically generated

**Congratulations!** We have implemented correctly and efficiently the “person collection” data structure.

## 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):

* string AddProduct(string name, decimal price, string producer)
  + Adds a product by given **name**, **price**, and **producer**. Duplicates are **allowed**. As a result, the command prints “**Product added**”.
* string DeleteProducts(string producer)
  + Deletes all products matching given **producer**. As a result, the command prints “**X products deleted**” where **X** is the number of deleted products.
  + Prints “**No products found**” if no such products exist.
* string DeleteProducts(string name, string 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.
  + Prints “**No products found**” if no such products exist.
* string FindProductsByName(string 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**.
  + If no products exist with the specified **name**, the command prints “**No products found**”.
* string FindProductsByProducer(string 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.
  + If no products exist by the specified **producer**, the command prints “**No products found**”.
* string FindProductsByPriceRange(decimal fromPrice, decimal toPrice)
  + Finds all products whose price is **greater** or **equal** to **fromPrice** and **less** or **equal** to **toPrice**.
  + As a result, the command prints a **list** of products in format **{name;producer;price}**, ordered by **name**, **producer**, and **price**.
  + If no products exist within the specified price range, the command prints “**No products found**”.

All string matching operations are **case-sensetive**.

### Input

The input data should be **read from the console**.

* At the first line you will be given the number **N** **of 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 **decimal separator**.
* Prices should be printed with exactly **2 digits** after the decimal point (e.g., 320.30 instead of 320.3).

### Examples

|  |  |
| --- | --- |
| **Input Example** | **Output Example** |
| 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 |

### Hints

In the project skeleton we have given you **4 classes**.

Text

Description automatically generated

The ShoppingStructure class is **empty**. That is where you will **implement** the **methods** needed for the data structure.

The DictionaryExtensions class has an extension method you can use if you will **add to a nested dictionary**:

Graphical user interface, text, application

Description automatically generated

The Product class is an **implemented** class that can be used when **adding** values to your structure:

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

The last class, StartUp is where the code executes and where you will have to **handle each command**:

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text

Description automatically generated