# Data Structures Fundamentals with C# - Exam Preparation

This document defines the examination problems for ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/3921/data-structures-fundamentals-with-csharp-november-2022). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.org/Contests/3412/Data-Structures-Fundamentals-with-CSharp-Exam-19-March-2022).

What does this document define and describes?

## Vacc Ops

Everybody is talking about vaccines and green certificates. Also, as you can guess, the systems managing the personal doctors and their patients are morally obsolete. Now it’s **your job** to implement a new and fresh one.

You are given a skeleton with a class **VaccOps** that implements the **IVaccOps interface.**

The VaccOps works with **Doctor** & **Patient entities**, **all** **doctors and patients** are identified by their **unique** **names**. Implements all the operations from the **interface:**

* **void AddDoctor(Doctor d)** – **adds** a doctor. If there **is** **a doctor** **with** **the** **same** **name** added before throw ArgumentException().
* **void AddPatient(Doctor d, Patient p)** – **adds a patient for** the provided **doctor**. If the doctor does not exist, throw ArgumentException().
* **IEnumerable<Doctor> GetDoctors() –** returns all added doctors. If there aren’t any - return empty collection
* **IEnumerable<Patient> GetPatients()** **–** returns all added patients. If there aren’t any - return empty collection
* **bool Exist(Doctor d) –** returns whether the **Doctor** has been **added** or **not**
* **bool Exist(Patient p) –** returns whether the **Patient** has been **added** or **not**
* **Doctor RemoveDoctor(string name)** – **removes Doctor by provided name and all his patients.** If the doctors does not exist - throw ArgumentsException()
* **void ChangeDoctor(Doctor from, Doctor to, Patient p) –** Move the patients from the first doctor to another. If any of the provided entities does not exist - throw ArgumentsException()
* **IEnumerable<Doctor> GetDoctorsByPopularity(int popularity)** – **return only** doctors with popularity **equal** to specified. If there are not any return empty collection
* **IEnumerable<Patient> GetPatientsByTown(string town)** – **return only** patients from the specified town. If there are not any return empty collection
* **IEnumerable<Patient> GetPatientsInAgeRange(int lo, int hi)** – returns all **patients** with the **specified age range - inclusive upper and lower bound .** If there aren’t return empty collection
* **IEnumerable<Doctor> GetDoctorsSortedByPatientsCountDescAndNameAsc()** – returns all doctors sorted by their patients count descending and the doctor's name ascending. If there are not any return empty collection
* **IEnumerable<Patient>GetPatientsSortedByDoctorsPopularityAscThenByHeightDescThenByAge()** – returns all patients sorted by their doctor's popularity ascending then by the patient's height descending and then by patient’s age ascending. If there are not any return empty collection

## Vacc Ops – 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 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: **add**, **size**, **remove**, **get** 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, **edge** **cases** will mostly be ignored such as throwing exceptions etc…

## Coupon Ops

You are given a skeleton with a class **CouponOperations** that implements the **ICouponOperations interface.**

The CouponOperations works with **Coupon** & **Website entities**, **all** **websites and coupons** are identified by their **unique** **domains & codes**. Implements all the operations from the **interface:**

* **void RegisterSite(Website w)** – **adds** a website. If there **is** **a website with** **the** **same** **domain** added before, throw ArgumentException().
* **void AddCoupon(Website w, Coupon c)** – **adds a coupon for** the provided **website**. If the website does not exist, throw ArgumentException().
* **Website RemoveWebsite(string domain) – removes the website**. If there is no website with the provided domain - throw ArgumentException(). Removing the website should delete all its coupons.
* **Coupon RemoveCoupon(string code)** **–** **removes the coupons**. If there is no coupons with the provided code - throw ArgumentException()
* **bool Exist(Website w) –** returns whether the **Website** has been **added** or **not**
* **bool Exist(Coupon c) –** returns whether the **Coupon** has been **added** or **not**
* **IEnumerable<Website> GetSites()** – **return collection of all added websites.** If there aren’t any, return an empty collection.
* **IEnumerable<Coupon> GetCouponsForWebsite(Website w) –** return **all coupons** for the provided website. If there are not any - return an empty collection. If the website does not exist throw ArgumentException()
* **void UseCoupon(Website w, Coupon c)** – **remove** the provided coupon for the provided website. If the coupon is not for the provided website, throw ArgumentException(). Both website and coupon should be existent, otherwise throw ArgumentException()
* **IEnumerable<Coupon> GetCouponsOrderedByValidityDescAndDiscountPercentageDesc()** – **return all coupons** orderedby their validity descending and then by their discount percentage descending
* **IEnumerable<Website> GetWebsitesOrderedByUserCountAndCouponsCountDesc()** – return all websites ordered by their user count descending and available coupons descending

## Coupon Ops – 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 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: **add**, **size**, **remove**, **get** 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, **edge** **cases** will mostly be ignored such as throwing exceptions etc…