**Data Structures Advanced with C# Exam – 21.11.2020**

This document defines the exam preparation for ["Data Structures – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/3113/data-structures-advanced-with-csharp-october-2020).

Please submit your solutions (**source code**) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2615/Data-Structures-Advanced-with-CSharp-Exam-21-Nov-2020).

# Dog Vet

You are given a skeleton with a class **DogVet** that implements the **IDogVet interface.**

This **DogVet** works with **Dog** & **Owner entities**, **all** **dogs and owners** are identified by their **unique** **Ids** (there will **not** **be** **two** dogs/owners with the same **Id**). A **Owner cannot own two dogs with the same Name.**  Implements all the operations from the **interface:**

* **void AddDog(Dog dog, Owner owner)** – **adds** a dog to if there **is** **a dog** **with** **the** **same** **id** added before **throw ArgumentException()**.If the owner has a dog with the same name - **throw ArgumentException()**
* **bool Contains(Dog dog)** – **returns** whether the dog **is** **present** inside the **DogVet** **or** **not**
* **int Size –** returns the **number** of dogs
* **Dog GetDog(string name, string ownerId) –** **returns** the **dog** by the specified **name**, if there is **no** such dog **or** owner **throw ArgumentException()**
* **Dog RemoveDog(string name, Owner owner)** – **removes** the dog by the name specified and returns it – if such dog exists,  **if the dog or owner is not present throw ArgumentException()**
* **IEnumerable<Dog> GetDogsByOwner(string ownerId) - returns the dogs by ownerId.** If the owner is not present **throw ArgumentException()**
* **IEnumerable<Dog> GetDogsByBreed(Breed breed)** – Gets all dogs by breed type. If there aren’t any dogs **throw ArgumentException()**
* **void Vaccinate(string name, string ownerId)** – **increases** the numbers of Vaccines property of the Dog. If the dog or the owner are not present **throw ArgumentException**
* **void Rename(string oldName, string newName, string ownerId)** – this method should **change** the Dog name **oldName** with the **newName**, however if there dog or the owner are not present **throw ArgumentException()**
* **IEnumerable<Dog> GetAllDogsByAge(int age)** – returns all the **dogs** with the **specified age. If there aren’t any throw ArgumentException()**
* **IEnumerable<Dog> GetDogsInAgeRange(int lowerBound, int upperBound)** – returns the **dogs** with **age** in the range specified the lower bound is **inclusive** the upper bound is **inclusive. If there aren’t any dogs in the specified range. Return empty collection.**
* **IEnumerable<Dog> GetAllOrderedByAgeThenByNameThenByOwnerNameAscending()** – returns all dogs ordered by age ascending then by name ascending then by owner name ascending . If there are none return an **empty collection**
  1. **Dog Vet – 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** **behaviour** will be important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…

# Fit Gym

You are given a skeleton with a class **FitGym** that implements the **IGym interface.**

This class stores two types of entities: **Members** and **Trainers**, those entities are identified by their **unique** **ids** (there will **not** **be** **two** members and trainers with the same ids). Implements all the operations from the **interface**:

* **void AddMember(Member member) –** adds **member** to the **fit gym** if the member is **previously** **stored** throw ArgumentException()
* **void HireTrainer(Trainer trainer)** - adds **trainer** to the **fit gym.** If the trainers is **previously stored** throw ArgumentException()
* **void Add(Trainer trainer, Member member) –** adds **member** to the **gym**. The members are **trained by** the given trainer, so you have to store the members in some sort of **relation** to its trainer. If the **members does not** exist add it, if it has been previously added **assign** the trainer to him. If the trainer does not exist **OR** the member **already has a** trainer throw ArgumentException()
* **bool Contains(Member member)** – returns whether the **member** is **stored** inside the fit gym or **not**
* **bool Contains(Trainer trainer)** – returns whether the **trainer** is **stored** inside the fit gym or **not**
* **Trainer FireTrainer(int id)** – removes the **trainer** form the fit gym and **returns** **only** the **trainer.** If there is no such trainer throw ArgumentException()
* **Member RemoveMember(int id)** – removes the **member** form the fit gym and **returns** **it.** If there is no such member throw ArgumentException()
* **int MemberCount** – returns the number of **members** stored
* **int TrainerCoint** – returns the number of **trainers** stored
* **IEnumerable<Members> GetMembersInOrderOfRegistrationAscendingThenByNamesDescending()** – returns the members ordered by **registration date ascending** then by their **names descending**. If there are no members - return **empty** collection.
* **IEnumerable<Trainer> GetTrainersInOrdersOfPopularity()** – returns the trainers ordered by their **popularity** **ascending**. If there are not trainers in the gym - return **empty** collection
* **IEnumerable<Member>** **GetTrainerMembersSortedByRegistrationDateThenByNames(Trainer trainer)** – returns all the **members by it’s trainer** ordered by registration date then by names ascending. If the trainer has no members assigned to him - empty collection.
* **IEnumerable<Member>** **GetMembersByTrainerPopularityInRangeSortedByVisitsThenByNames(int lo, int hi)** – returns all members which trainer’s popularity is in the **range lo >= popularity <= hi,** sorted by **visits** ascending, then by **names ascending**. If there are no such members, return an **empty** collection.
* **Dictionary<Trainer, HashSet<Members>>** **GetTrainersAndMemberOrderedByMembersCountThenByPopularity()** – returns a **dictionary** with **trainers** as a **key** and **hash set as a value** of **his members. Ordered by trainer’s members count then by trainer’s popularity ascending.** 
  1. **Fit Gym – 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**, **Count**, **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** **behaviour** will be considered important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…