**Data Structures Advanced Exam – C#**

This document defines the exam preparation for ["Data Structures – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/3420/data-structures-advanced-with-csharp-august-2021).

Please submit your solutions (**source code**) of all below described problems in [Judge](https://judge.softuni.org/Contests/3247/Data-Structures-Advanced-Exam-19-Sept-2021).

# Re-Play – 50 pts

Re-Play is an interactive music platform, which provides functionality for registering songs, adding them in a queue and listening to them one by one.

You are given a skeleton with a class **RePlayer** that implements the **IRePlayer interface.**

This **RePlayer** works with **Track** entities. All **Track** entities are identified by a **unique Id**.

The **Track** entity contains the following properties:

* **Id** – string
* **Title** – string
* **Artist** – string (name of artist)
* **Plays** – integer (number of times the track has been played)
* **DurationInSeconds** - integer

Implement the following functionalities to make the **RePlayer** fully operative:

* **void AddTrack(Track track, string albumName)** – **adds** a **track** to the **RePlayer**. If there is **no album** with the **given name**, you should create it and store it. **NOTE**: Albums names are **unique** and **track titles** are unique **inside** the albums.
* **bool Contains(Track track)** –returns whether the track is **contained** inside the **RePlayer**.
* **int Count –** returns the **total count** of all **tracks**.
* **Track GetTrack(string title, string albumName) –** returns the **track** with the specified **title** from the **album** with the **specified name**. If there is no such **track** or **album** - **throw ArgumentException()**
* **IEnumerable<Track> GetAlbum(string albumName)** –returns **all tracks** from the **album** with the given name, **ordered** by **number** of **plays** in **descending order**. If there is no such **album** - **throw ArgumentException()**
* **void AddToQueue(string title, string albumName) –** adds the **track** with the specified **title** from the **album** with the **specified name** to the **Listening Queue**. If there is no such **track** or **album** - **throw ArgumentException()**
* **Track Play()** – returns the **first track** that **entered** the **Listening Queue**, **increases** the **number** of **plays** with **1** for the **corresponding track** and **removes** it from the **Listening Queue**. If there is no such **track** - **throw ArgumentException()**
* **void RemoveTrack(string title, string albumName)** – **removes** the **track** with the given **title** from the **album** with the **given name**, and from the **whole** **RePlayer**.   
  **NOTE**: If the given track is currently in the **Listening Queue**, it should be **removed** from the **Listening Queue** as it is being deleted from the **RePlayer**.If there is no such **track** or **album** - **throw ArgumentException()**
* **IEnumerable<Track> GetTracksInDurationRangeOrderedByDurationThenByPlaysDescending(int lowerBound, int upperBound)** – returns all of the **tracks** with **duration** in the range specified with **lower bound** and **upper bound.** Both bounds are **inclusive**.The results should be ordered by **duration** in **ascending order** and then by **number of** **plays** in **descending** order. If there aren’t any tracks in the specified range – return an **empty collection**.
* **IEnumerable<Track> GetTracksOrderedByAlbumNameThenByPlaysDescendingThenByDurationDescending()** – returns all of the **tracks** ordered by **album name** in **ascending order**, then by **number of plays** in **desceding order** and then by **duration** in **descending order**. If there aren’t any tracks – return an **empty collection**.
* **Dictionary<string, List<Track>> GetDiscography(string artistName)** – returns all tracks, **GROUPED** by **ALBUM NAME**, that are performed by the **given artist**. If artist does not exist or there aren’t any tracks by that name – **throw ArgumentException()**.
  1. **Re-Play – Performance – 100 pts**

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… Also, make sure you are using the correct data structures. ☺

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…

# Mobile-X – 100 pts

The 2 largest local platforms for Vehicle vendors have completely fallen due to some technical bugs. A local businessman by the name of Melon Usk decided to take advantage of this and create his own platform for people to sell their vehicles at. He has everything ready except the main functionality – guess who’s going to help him with that… That’s right, you!

Mobile-X is a system in which each user can add a vehicle to be sold. The vehicle ads for sale contain very basic information, but there is a lot of functionality that Melon needs you to do.

You are given a skeleton with a class **VehicleRepository** that implements the **IVehicleRepository interface.**

This **VehicleRepository** works with **Vehicle** entities. All **Vehicle** entities are identified by a **unique Id**.

The **Vehicle** entity contains the following properties:

* **Id** – string
* **Brand** – string
* **Model** – string
* **Location** – string
* **Color** – string
* **Horsepower** – int
* **Price –** double
* **IsVIP** – bool

Implement the following functionalities to make the **VehicleRepository** fully operative:

* **void AddVehicleForSale(Vehicle vehicle, string sellerName)** – **adds** a **Vehicle** to the **VehicleRepository**. If there is **no seller** with the **given name**, you should create it and store it. **NOTE**: Seller names are **unique**.
* **bool Contains(Vehicle vehicle)** –returns whether the vehicle is **contained** inside the **VehicleRepository**.
* **int Count –** returns the **total count** of all **vehicles**.
* **IEnumerable<Vehicle> GetVehicles(List<string> keywords) –** returns **all vehicles** that have a **brand**, **model**, **location** or **color** **equal** to **any** of the **given keywords**. The results should be **ordered** by **price** in **ascending order**, but **VIP vehicles** should **come first**, regardless of their price. **NOTE**: **VIP** **vehicles** should also be **ordered** by **price** in **ascending order**. If there aren’t any vehicles – return an **empty collection**.
* **IEnumerable<Vehicle> GetVehiclesBySeller(string sellerName)** –returns **all vehicles** from the **seller** with the given **name**, **ordered** by **order** of **entrance**. If there is no such **seller** - **throw ArgumentException()**
* **IEnumerable<Vehicle> GetVehiclesInPriceRange(int lowerBound, int upperBound)** –returns all of the **vehicles** with **price** in the range specified with **lower bound** and **upper bound.** Both bounds are **inclusive**.The results should be ordered by **horsepower** in **descending order**   
  If there aren’t any tracks in the specified range – return an **empty collection**.
* **Dictionary<string, List<Vehicle>> GetAllVehiclesGroupedByBrand()** – returns all vehicles, **grouped** by **brand**. **Vehicles** for each **brand** should be **ordered** by **price** in **ascending order**. If there aren’t any vehicles – **throw ArgumentException()**.
* **void RemoveVehicle(string vehicleId)** – **removes** the **vehicle** with the given **id** from the **VehicleRepository**.If there is no such **vehicle** - **throw ArgumentException()**
* **IEnumerable<Vehicle> GetAllVehiclesOrderedByHorsepowerDescendingThenByPriceThenBySellerName()** – returns all of the **vehicles** ordered by **horsepower** in **descending order**, then by **price** in **ascending** order, and then by **seller name** in **ascending order**. If there aren’t any vehicles – return an **empty collection**.
* **Vehicle BuyCheapestFromSeller(string sellerName)** – **removes** from the **VehicleRepository** and **returns** the **vehicle** with the **lowest price** from the **seller** with the **given name**. If there is no such **vehicle** or **seller** - **throw ArgumentException()**
  1. **Mobile-X – Performance – 50 pts**

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… Also, make sure you are using the correct data structures. ☺

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…