**Data Structures C# Exam Retake**

# **Barber Shop**

You are given a skeleton with a class **BarberShop** that implements the **IBarberShop interface.**

The BarberShop works with **Barber** & **Client entities**, **all** **barbers and clients** are identified by their **unique** **names**. Implements all the operations from the **interface:**

* **void AddBarber(Barber b)** – **adds** a barber. If there **is** **a barber** **with** **the** **same** **name** added before, throw ArgumentException().
* **void AddClient(Client c)** – **adds a client**. If a client with the same name exists, throw ArgumentException().
* **bool Exist(Barber b) –** returns whether the **Barber** has been **added** or **not**
* **bool Exist(Client c)** **–** returns whether the **Client** has been **added** or **not**
* **IEnumerable<Barber> GetBarbers() –** returns all added barbers. If there aren’t any - return empty collection
* **IEnumerable<Client> GetClients() –** returns all added clients. If there aren’t any - return empty collection
* **void AssignClient(Barber b, Client c)** – **adds a client for** the provided **barber**. If the barber or the client does not exist, throw ArgumentException()
* **void DeleteAllClientsFrom(Barber b) –** **Deletes** all assigned clients for the provided **barber.** If the barber does not exist throw ArgumentException()
* **IEnumerable<Client> GetClientsWithNoBarber()** – **return only** clients with no assigned barber
* **IEnumerable<Barber> GetAllBarbersSortedWithClientsCountDesc()** – **return all added barbers** ordered by their clients count descending. If there are not any barbers return empty collection
* **IEnumerable<Barber> GetAllBarbersSortedWithStarsDecsendingAndHaircutPriceAsc()** – returns all **barbers** sorted by their stars descending and their haircut price ascending
* **IEnumerable<Client> GetClientsSortedByAgeDescAndBarbersStarsDesc()** – return only clients who are assigned to berber and sorted by their age descending and by their barber stars descending

# **Barber Shop – 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…

# **Trip Administrator**

You are given a skeleton with a class **TripAdministrations** that implements the **ITripAdministrations interface.**

The TripAdministrations works with **Company** & **Trip entities**, **all** **companies and trips** are identified by their **names and ids.** A company is allowed to have only **tripOrganizationLimit number of Trips to manage.** Implements all the operations from the **interface:**

* **void AddCompany(Company c)** – **adds** a company. If there **is** **a company with** **the** **same** **name** added before, throw ArgumentException().
* **void AddTrip(Company c, Trip t)** – **adds a trip for** the provided **company**. If the company does not exist, throw ArgumentException().
* **bool Exist(Company c) –**  returns whether the **Company** has been **added** or **not**
* **bool Exist(Trip t)** **–** returns whether the **Trip** has been **added** or **not**
* **void RemoveCompany(Company c) –** **remove** the provided company with all its **trips**. If the company does not exist, throw ArgumentException()
* **IEnumerable<Company> GetCompanies() –** return a collection of **all added companies**. If there are not any - return empty collection
* **IEnumerable<Trip> GetTrips() – r**eturn a collection of **all added trips**. If there are not any - return empty collection
* **void ExecuteTrip(Company c, Trip t) –** **remove** the trip for the **provided company.** If the company or trip does not exist - throw ArgumentException(). If the trip is not for the provided company again - throw ArgumentException()
* **IEnumerable<Company> GetCompaniesWithMoreThatNTrips(int n)** – return all companies with more than N trips.
* **IEnumerable<Trip> GetTripsWithTransportationType(Transportation t)** – **return all trips** filtered by the transportation type
* **IEnumerable<Trip> GetAllTripsInPriceRange(int lo, int hi)** – return trips in between provided price range **inclusive**

# **Trip Administrator – 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…