**Exercise: Hash Tables Sets and Dictionaries**

This document defines the lab for ["Data Structures – Advanced (C#)" course @ Software University](https://softuni.bg/trainings/3923/data-structures-advanced-with-csharp-december-2022).

Please submit your solutions (**source code**) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2606/06-Hash-Tables-Sets-and-Dictionaries-Exercise).

## Royale Arena

You have been invited by the blue king himself. He is in a huge need for an application to help him fight in the glorious Royale Arena. As one of the top players on the arena, he has a reputation to maintain, therefore he cannot afford to lose. The application must be able to keep and query all the cards that exist within the game. Each card has its own specifics. The trick is that the intervals between the battles are so small, that the application must be fast in order to help the king.

**Battlecard** will hold:

* **int Id** – unique card id
* **CardType Type** – enumeration of battlecard
* **string Name** – the name of the card
* **double Damage** – the damage of the card
* **double Swag** – the swag of the card

You need to support the following operations (and they should be **fast**):

* **Add()** – Add a battle card to the arena. You will need to implement the **Contains()** methods as well.
* **Contains(Battlecard)** – checks if a given battlecard is present in the arena.
* **Count** – returns the number of cards in the arena
* **ChangeCardType(Id, Type)** – changes the status of the battlecard with the given id or throws **InvalidOperationException** if no such card exists.
* **GetById(Id)** – return the card with the given **Id**. If such a card doesn't exist, throw **InvalidOperationException**.
* **RemoveById(Id)** – Removes the card with the given id, otherwise throws **InvalidOperationException**
* **GetByCardType(Type)** – Returns the cards with the given **card type ordered by damage descending then by id**. If there are no cards with the given type, throw **InvalidOperationException**
* **GetByTypeAndDamageRangeOrderedByDamageThenById(type, lo, hi)** – returns all cards with given type and damage between lo (inclusive) and hi (inclusive) ordered by damage descending, then by id ascending. If there are no such types throw **InvalidOperationException**
* **GetByCardTypeAndMaximumDamage(type, damage)** – returns all cards with given type and damage less or equal to a maximum allowed amount **ordered** by damage descending, **then** by id. Throws an **InvalidOperationException** such cards were not found.
* **GetByNameOrderedBySwagDescending(name)** – search for all cards with a specific name and return them **InvalidOperationException** by swag descending then by id. If there are no such cards throw **InvalidOperationException**
* **GetByNameAndSwagRange(name, lo, hi)** – returns all cards with given name and swag between lo (inclusive) and hi (exclusive) ordered by swag descending then by id. If there are no such cards throw **InvalidOperationException.**
* **FindFirstLeastSwag(n) –** Returns the first n cards with least swag. If there are two identical swags, **order** by id. If the argument passed **exceeds** the count of the arena, throw **InvalidOperationException**
* **GetAllInSwagRange(lo, hi)** – returns all cards within a **range** ordered by swag(the range is inclusive). Returns an empty collection if no such cards were found.
* **GetEnumerator()** – Iterate the arena by insertion order

**Restrictions:**

* You are **not allowed to change the interface**.
* You can add to the **Battlecard** class, but don't remove anything.
* You can edit the **Arena** class if it implements the **IArena** interface.

## Word Cruncher

Write a program that receives some **strings** and **forms another** string that is required. On the **first line** you will be given **all the strings** separated by **comma and space**. On the next line you will be given the **string** that needs to be **formed from the given strings**. For more clarification see the examples below.

### Input

* On the first line you will receive the **strings** (separated by comma and space **", "**)
* On the next line you will receive the **target string**

### Output

* Print each of the found ways to form the required string in sorted order

### Constrains

* There might be **repeating elements** in the input

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| text, me, so, do, m, ran  somerandomtext | so me ran do m text |
| this, th, is, Word, cruncher, cr, h, unch, c, r, un, ch, er  Wordcruncher | Word c r un ch er  Word c r unch er  Word cr un c h er  Word cr un ch er  Word cr unch er  Word cruncher |

## Hierarchy

A **Hierarchy** is a data structure that stores elements in a hierarchical order. See the example:

Graphical user interface, application

Description automatically generated

It supports the following operations:

* **Add(element, child)** - adds **child** to the hierarchy as a child of **element**.
  + Throws an exception if **element** does not exist in the hierarchy.
  + Throws an exception if **child** already exists (duplicates are not allowed).
* **Remove(element)** - removes the element from the hierarchy.
  + If it has children, they become children of the element's parent.
  + If element is root node, throws an exception.
* **Count** - returns the count of all elements in the hierarchy
* **Contains(element)** - determines whether the element is present in the hierarchy.
* **GetParent(element)** - returns the parent of the element.
  + Throws an exception if **element** does not exist in the hierarchy.
  + Returns the **default value for the type** (e.g. **int** → **0**, **string** → **null**, etc.) if element has no parent.
* **GetChildren(element)** - returns a collection of all direct children of the element in order of their addition.
  + Throws an exception if **element** does not exist in the hierarchy.
* **GetCommonElements(Hierarchy other)** - returns a collection of all elements that are present in both hierarchies (order does not matter).
* **ForEach()** - enumerates over all elements in the hierarchy by levels.
  + In the image above, the elements would be enumerated as such - **Leonidas** -> **Xena the Princess Warrior** -> **General Protos** -> **Gorok** -> **Bozat** -> **Subotli** -> **Kira** -> **Zaler**.

### Input and Output

You are given a **Visual Studio C# project skeleton** (unfinished project) holding the interface **IHierarchy**, the unfinished class **Hierarchy** and **tests** covering its **functionality** and its **performance**.

Your task is to **finish this class** to make the tests run correctly.

* You are **not allowed to change the tests**.
* You are **not allowed to change the interface**.