**Data Structures Advanced with C# - Retake Exam**

1. **Expression Passion – 100 pts**

The Expressionist is a functional module for logical and mathematical expression calculations. It provides features for analytical observation of registered expressions.

You are given a skeleton with a class **Expressions** that implements the **IExpressionist interface.**

This **Expressionist** works with **Expression** entities. All **Expression** entities are identified by a **unique Id**.

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

* **Id** – string
* **Value** – string
* **Type –** one of the following values (**Operator** / **Value**)
* **Left Child** – an Expression object
* **Right Child** – an Expression object

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

* **void AddExpression(Expression expression)** – **adds** an **expression** to the **Expressionist**. This method should only work if the Expressionist is **empty** (no expressions added).
  + If there are expressions and this method is called - **throw ArgumentException()**
* **void AddExpression(Expression expression, String parentId)** – **adds** an **expression** as a child to the Expression with the given **parentId** in the **Expressionist**.
  + If the **parent expression** does **NOT** have a **Left Child** – set the newly added expression as its **left child**.
  + If the **parent expression** has a **Left Child** but does **NOT** have a **Right Child** – set the newly added expression as its **right child**.
  + If the parent expression has both a Left Child and a Right Child - **throw ArgumentException()**
  + If there is **no such parent expressions** with the given **parentId** in the Expressionist - **throw ArgumentException()**
  + **NOTE**: There will be **no cases** in which the parent will have a Right Child, but no Left Child – there is no need to validate this.
  + **NOTE:** There will be **no cases** in which this method is called, when the Expressionist is empty. There is no need to validate this.
* **bool Contains(Expression expression)** –returns whether the expression is **contained** inside the **Expressionist**.
* **int Count() –** returns the **total count** of all **expressions**.
* **Expression GetExpression(string expressionId)** – **retrieves** the **expression** with the given **expressionId.**   
  If there is no such **expression** - **throw ArgumentException()**
* **void RemoveExpression(string expressionId)** – **removes** the **expression** with the given **expressionId** from the **Expressionist.**   
  If there is no such **expression** - **throw ArgumentException()**
  + **NOTE**: This should **also remove all expressions** that are **children** of the **given expression** (this includes the children of their children and so on) – e.g. the whole hierarchy below the currently removed expression should be **removed**.
  + If the expression is a Left Child of another expression, you should replace it with the Right Child and the parent expression should remain only with a Left Child.
    - **Example –** A has left child B and right child C
    - We remove B
    - A now has left child C and right child – none.
  + If the expression is a Right Child of another expression, you should just remove it.
    - **Example –** A has left child B and right child C
    - We remove C
    - A now has left child B and right child – none.
* **string Evaluate()** – By now you should have noticed that this is a tree-like structure. You should perform an **in-order traversal** on all expressions starting from the root.
  + During the traversal, you should stringify each expression according to these scenarios:
  + If the current expression has type – **Value**, you should just add it’s **value property** to the result.
  + If the current expression has type – **Operator**, you should add the result of the evaluate function of it’s **left child**, followed by a **whitespace**, then the **value property** of the expression, followed by another whitespace, and then the result of the evaluate function of the **right child**. The whole thing should be wrapped in **parentheses**.
  + Example: A (type – operator, value = "**+**") has children B (type – value, value = "5") and C (type – value, value = "10").
  + Evaluate(A) = "(" + evaluate(B) + " " + A.value + " " + evaluate(C) + ")" Evaluate(A) = (5 + 10).

If there aren’t any activities – return an **empty string**.

* 1. **Expression Passion – 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** **behavior** will be important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…

1. **Package Manager Lite – 100 pts**

You most certainly have heard of NuGet by now – it’s a package management system. Package management systems are quite complex, as you’ll see in your next task. **Packages** also **depend** on **other packages** – they have dependencies.

You are given a skeleton with a class **PackageManager** that implements the **IPackageManager interface.**

This **PackageManager** works with **Package** entities. All **Package** entities are identified by a **unique Id**.

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

* **Id** – string
* **Name** – string
* **Version**– string
* **Release Date** – DateTime object

Implement the following functionalities to make the **Package Manager** fully operative:

* **void RegisterPackage(Package package)** – **adds** a **Package** to the **Package Manager**.  
  If a **package** with the **same name** and **version** as the added one, already exists- **throw ArgumentException()**
* **void RemovePackage(string packageId)** – **removes** the **package** with the given **id** from the **Package Manager**.
  + This also means that all associations with this package are removed. E.g. If any package has the removed package as a dependency – it should be removed from its dependencies.
  + If there is no such **package** - **throw ArgumentException()**
* **void AddDependency(string packageId, string dependencyId)** – **adds** the **package** with the given **dependencyId** as a **dependency** to the package with the given **packageId** from the **Package Manager**.If **either one** of the **packages** does not exist - **throw ArgumentException()**
  + If there is no such **package** - **throw ArgumentException()**
* **bool Contains(Package package)** –returns whether the **package** is **contained** inside the **Package Manager**.
* **int Count() –** returns the **total count** of all **packages**.
* **IEnumerable<Package> GetDependants(Package package)** – **returns** **all packages** that have the given Package as a **dependency**.If there are **no such packages** – return an **empty collection**.
* **IEnumerable<Package> GetIndependentPackages() –** returns **all** **packages** that have **0 dependencies**.

Results should be ordered by **Release** **Date** in **descending order** and by **version** In alphabetical (ascending) order.

If there are **no such packages** – return an **empty collection**.

* **IEnumerable<Package> GetOrderedPackagesByReleaseDateThenByVersion()** – returns **all** **packages** ordered by Release date in descending order, then by **version** in **alphabetical (ascending) order**.

**NOTE**: If there are 2 or more packages with the **same name** (e.g. 2 or more versions of the same package) return the one with the **latest version**.

If there aren’t any packages – return an **empty collection**.

**NOTE: If all sorting criteria fails, you should order by order of input. This is for all methods with ordered output.**

* 1. **Package Manager Lite – 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** **behavior** will be important, **edge** **cases** will mostly be ignored such as throwing exceptions etc…