Exercises: Intro to Data Structures

Problems for exercises and homework for the "Data Structures and Algorithms Basics" course from the official "Applied Programmer" curriculum.

You can check your solutions here: https://judge.softuni.bg/Contests/2930/Intro-to-Data-Structures-Exercises

1. Events in Given Date Range

Write a program that reads a set of **events** in format "**{Event name} | Date and time**" and a date range **a** < **b** and prints all events within the range [a … b] inclusively (ordered by date; for duplicated dates preserve the order of appearance).

Use **Ordered Multi-Dictionary**. Note that you should first install **`SoftUni.Wintellect.PowerCollections`** package from NuGet Packages.

Examples

Input	Output
5 C# Course - Group II | 15-Aug-2015 14:00 Data Structures Course | 13-Aug-2015 18:00 C# Course - Group I | 15-Aug-2015 10:00 Seminar for Java Developers | 18-Aug-2015 19:00 Game Development Seminar | 15-Aug-2015 10:00 15-Aug-2015 10:00 | 1-Sep-2015 0:00	{C# Course - Group I,Game Development Seminar} | 15-Aug-2015 10:00 {C# Course - Group II} | 15-Aug-2015 02:00 {Seminar for Java Developers} | 18-Aug-2015 07:00

Read All Events in Ordered Multi-Dictionary

Read all events in an **ordered multi-dictionary**:

```
Thread.CurrentThread.CurrentCulture = CultureInfo.InvariantCulture;

var events = new OrderedMultiDictionary<DateTime, string>(true);

int n = int.Parse(Console.ReadLine());
for (int i = 0; i < n; i++)
{
    string[] eventTokens = Console.ReadLine().Split(" | ");
    string eventName = eventTokens[0];
    DateTime eventDate = DateTime.Parse(eventTokens[1]);
    events.Add(eventDate, eventName);
}
```

Initially, we **reset the current culture** (locate) to ensure the system locale in the operating system regional settings does not affect the date and time format (we want to use the neutral date and time format).

Then, we **create an ordered multi-dictionary**: **`OrderedMultiDictionary<DateTime, string>`**. It maps the event dates and times to event names.

Finally, we read the input line by line and put the events from each line into the multi-dictionary.

Find the Events in the Given Dates Range

Write some code to efficiently take a subrange from the ordered multi-dictionary. Read the start and end dates from the console and then use **`.Range(startDate, true, endDate, true)`** method:

```
string[] datesTokens = Console.ReadLine().Split(" | ");
DateTime startDate = DateTime.Parse(datesTokens[0]);
DateTime endDate = DateTime.Parse(datesTokens[1]);
var eventsInRange = events.Range(startDate, true, endDate, true);
```

Print the Results

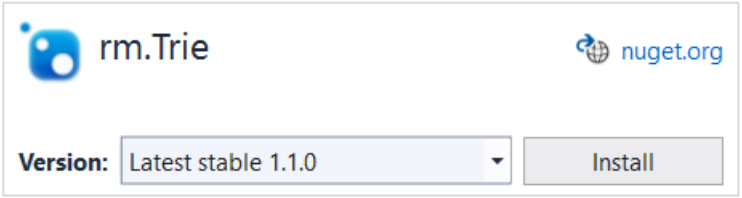
Finally, print the expected output. Use the **`ToString()`** method to **format** the date and time:

```
foreach (var e in eventsInRange)
{
    Console.WriteLine($"{e.Value} | {e.Key.ToString("dd-MMM-yyyy hh:mm")}");
}
```

2. Words with Prefix

`Trie` (Prefix Tree) are an extremely special and useful data structure that is based on the **prefix** of a string. That's why you can **quickly look up** prefixes of keys, enumerate all entries with a given prefix, etc. Also, **deletion** is very fast because it is straightforward.

First, to use the **`Trie`** data structure, you should install the **`rm.Trie`** NuGet package.



Get familiar with the **`Trie`** structure and methods from here: https://github.com/rmandvikar/csharp-trie. It is a good idea to look at the examples.

First, accept a **text** through the console and, on a new line, a **prefix string**. Then, do the following tasks:

- Print the **count** of **all words** in the text
- Print the **count** of **unique** words in the text
- Print **all words**, separated by ", "
- Print words, starting with the given **prefix**, separated by ", "
- **Remove** all words with the **prefix** from the trie
- Print all **remained** words, separated by ", "

Note that operations over the **`Trie`** are very **fast** because of its special tree structure.

Examples

Input	Output
She is a wonderful woman wo	5 5 She, is, a, wonderful, woman wonderful, woman She, is, a

My conscience hath a thousand several tongues and every tongue brings in a several tale and every tale condemns me for a villain t	23 17 My, conscience, condemns, hath, a, and, thousand, tongue, tongues, tale, several, every, brings, in, me, for, villain thousand, tongue, tongues, tale My, conscience, condemns, hath, a, and, several, every, brings, in, me, for, villain

3. Shopping Center

A **shopping center** keeps a set of **products**. Each product has **name**, **price** and **producer**. Your task is to model the shopping center and design a **data structure holding the products**. Write a program that executes **N** commands, given in the input (a single command at a line):

- **`AddProduct name;price;producer`** – adds a product by given name, price and producer. If a product with the same name / producer/ price already exists, the newly added product does not affect the existing ones (duplicates are allowed). As a result, the command prints "**Product added**".
- **`DeleteProducts producer`** – deletes all products matching given producer. As a result, the command prints "**X products deleted**" where **X** is the number of deleted products or "**No products found**" if no such products exist.
- **`DeleteProducts name;producer`** – deletes all products matching given product name and producer. As a result, the command prints "**X products deleted**" where **X** is the number of deleted products or "**No products found**" if no such products exist.
- **`FindProductsByName name`** – finds all products by given product name. As a result, the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. Print each product on a separate line. If no products exist with the specified name, the command prints "**No products found**".
- **`FindProductsByProducer producer`** – finds all products by given producer. As a result, the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. You should print each product on a single line**.** If no products exist by the specified producer, the command prints "**No products found**".
- **`FindProductsByPriceRange fromPrice;toPrice`** – finds all products whose price is greater or equal than **fromPrice** and less or equal than **toPrice**. As a result, the command prints a list of products in format **{name;producer;price}**, ordered by name, producer and price. You should print each product on a separate line. If no products exist within the specified price range, the command prints "**No products found**".

All string matching operations are **case-sensetive**.

Input

The input data should be read from the console.

- At the first line you will be given the number **N** of the commands.
- At each of the next **N** lines you will be given a command in the format described above.

The input data will always be valid and in the described format. There is no need to check it explicitly.

Output

The output data should be printed on the console.

The output should contain the output from each command from the input.

Constraints

- **N** will be between 1 and 50 000, inclusive.

- All strings specified in the commands (e.g. product names and producers) consist of alphabetical characters, numbers and spaces. Strings are case-sensitive.
- Prices are given as real numbers with up to 2 digits after the decimal point, (e.g. 133.58, 320.3, or 10)
- The '**.**' symbol is used as decimal separator.
- Prices should be printed with exactly **2 digits** after the decimal point (e.g. 320.30 instead of 320.3).
- Allowed working time for your program: **1.00 seconds** (at the judge environment).
- Allowed memory: **32 MB**.

Examples

Input	Output
17 AddProduct IdeaPad Z560;1536.50;Lenovo AddProduct ThinkPad T410;3000;Lenovo AddProduct VAIO Z13;4099.99;Sony AddProduct CLS 63 AMG;200000;Mercedes FindProductsByName CLS 63 AMG FindProductsByName CLS 63 FindProductsByName cls 63 amg AddProduct 320i;10000;BMW FindProductsByName 320i AddProduct G560;999;Lenovo FindProductsByProducer Lenovo DeleteProducts Lenovo FindProductsByProducer Lenovo FindProductsByPriceRange 100000;200000 DeleteProducts Beer;Ariana DeleteProducts CLS 63 AMG;Mercedes FindProductsByName CLS 63 AMG	Product added Product added Product added Product added {CLS 63 AMG;Mercedes;200000.00} No products found No products found Product added {320i;BMW;10000.00} Product added {G560;Lenovo;999.00} {IdeaPad Z560;Lenovo;1536.50} {ThinkPad T410;Lenovo;3000.00} 3 products deleted No products found {CLS 63 AMG;Mercedes;200000.00} No products found 1 products deleted No products found

Hints

In the provided skeleton, you are given all the **classes** you will need. However, you need to fill in some of the methods but first let's see what we are given.

First, examine the **`Product`** class in the **`Product.cs`** file. We have implemented the class structure for you- it has all needed **properties**, a **`CompareTo(product)`** method for comparing products and an overridden **`ToString()`** method for printing data.

Next, note that in the **`ProductList`** class we have combined different data structures to create suitable **collections** for our data:

```
private readonly Dictionary<string, OrderedBag<Product>> _byName;
private readonly Dictionary<string, OrderedBag<Product>> _byProducer;
private readonly OrderedDictionary<decimal, Bag<Product>> _byPrice;
private readonly Dictionary<string, Bag<Product>> _byNameAndProducer;
```

For example, we have used **`Dictionary<string, OrderedBag<Product>>`** for keeping **names**, because we want to find names fast (that's why we use a **dictionary** and names as a **key**) and, at the same time, be able to get products with that name without looping through all products and to have our products **sorted**. The use of **`OrderedBag<Product>`** is possible due to the **`CompareTo(product)`** method we implemented in the **`Product`** class.

Also, in addition to properties, we have implemented some **methods** for you to use, as they make work with complex data structures easier. **Examine** the methods carefully, as you are going to use them.

However, for even more simplicity when using data structures and for more clear code, we have implemented additional **dictionary** methods in the **`DictionaryExtensions`** class. **Examine** them, as well.

Finally, in the **`StartUp`** class you are given a **`ProductList`** and some **methods**. **Implement** the missing methods, using the methods from the **`ProductList`** class, which you already examined.