**Exercises: Linear Data Structures**

This document defines the lab for ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/3112/data-structures-fundamentals-with-csharp-september-2020).

Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/Contests/2441/Linear-Data-Structures-Exercise).

Write C# code for solving the tasks on the following pages.

Any code files that are part of the task are provided as **Skeleton**.

**Do not change the names of the provided projects, interfaces, classes and methods. You are free to create new ones as long as you follow the previously described rule.**

Some **tests may be provided** within the skeleton – use those for local **testing and debugging**, however there **is no guarantee that there are no hidden tests added inside Judge**.

Some of the problem will have simple **Benchmark** **tests** inside the skeleton. You can try to run those with **different** **values** and **different** **implementations** in order to **observe** behaviour. However **keep** in mind that the result comes **only as numbers** and this data may be **misleading** in some situations. Also the tests are not started from the command prompt which may **influence** the **accuracy** of the results. Those tests are only added as an **example** of **different** **data** **structures** **performance** on their **common** operations.

The Benchmark tool we are using is **BenchmarkDotNet**. You can find **additional** **information** here: [Getting started guide](https://benchmarkdotnet.org/articles/guides/getting-started.html) and there are other examples over the **Internet**.

## Faster Queue

You have the basic implementation of the **IAbstractQueue<T>** data structure from the lecture lab.

The task is simple - you have to modify the structure so now we can reduce the complexity when adding an element to a **constant factor**.

* **Enqueue(T item)**
  + Modify thisoperationso you can **perform it in constant time**, also modify anything required to achieve that.

**Hint**: you can add **additional node** that points to the end of the queue. But now you have to modify everything that somehow relates to the node chaining.

Here the tests are hidden so you have to figure out how to solve the problem above. Remember you can use the **benchmark** **tests** to observe the **performance**.

## DoublyLinkedList

Your task is to take the implementation of the **SinglyLinkedList<T>** form the lab and make it doubly linked list.

This means that you have to add two things:

* Add additional field **Node<T> tail** that will always **point to the last** element of the linked list.
* Add new property **Node<T>** **Previous** to the **Node class** this should point to the **previous node**.

Do the changes above the methods below should remain with unchanged erasure, use the tests provided to ensure that.

### AddFirst(T item)

### Adds an element in front of the collection and increases the size.

### AddLast(T item)

### Adds an element after the last element of the collection and increases the size.

### T RemoveFirst()

### Removes and returns the first element of the collection if there is such if no then throw InvalidOperationException with appropriate message.

### T RemoveLast()

### Removes and returns the last element of the collection if there is such if no then throw InvalidOperationException with appropriate message.

### T GetFirst()

### Returns but does not remove the first element of the collection if there is such if no then throw InvalidOperationException with appropriate message.

### T GetLast()

### Returns but does not remove the last element of the collection if there is such if no then throw InvalidOperationException with appropriate message.

### int Count

### Returns the number of elements inside the collection.

## ReversedList

Implement a data structure ReversedList<T> that holds a sequence of elements of generic type T.

It should hold a **sequence of items in reversed order**. The structure should have some **capacity** that **grows twice** when it is filled, **always starting at four**.

The reversed list should support the same operations that the **List** we have developed in the Lab implements but in a reversed order of adding elements.

**Hint:** you can keep the elements in the order of their adding but **access** them in **reversed order** (from end to start).

## Balanced Parentheses

Inside the **skeleton** you are given class **BalancedParentheses** and **BalancedParenthesesTest**.

Your task is to **implement** the **method** **Solve()** – which **performs** **analysis** of the **parentheses** filed and returns **true** or **false** whether the **parentheses** are **balanced** or **not**.

A sequence of parentheses **is balanced if** every open parenthesis can be paired uniquely with a closed parenthesis that occurs after the former.

You will be given three types of parentheses: (, {, and [.

**{[()]}** - This is a balanced parenthesis.

**{[(])}** - This is not a balanced parenthesis.