**Exercises: Linear Data Structures**

This document defines the lab for the ["Data Structures – Fundamentals (C#)" course @ Software University](https://softuni.bg/trainings/4266/data-structures-fundamentals-with-csharp-september-2023).

Please submit your solutions (source code) of all below-described problems in [Judge](https://judge.softuni.org/Contests/2441/02-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**.

## Circular Queue

Your task is to implement the **ADS** **IAbstractQueue<T>** inside the **Queue<T>** class provided.

You have to implement all the methods to solve the problem, however, you are free to add more methods with any access modifier you want.

### void Enqueue(T item)

* + **Adds** an **element** at the **end** of the **queue** and increases the size.

### T Dequeue()

##### Removes and returns the first element at the queue also decreases the size and performs a check if this method is called upon empty collection.

##### If so throw InvalidOperationException with the message of your choice.

### T Peek()

##### Returns the element at the current front of the queue. If the collection is empty throw InvalidOperationException with the appropriate message.

### T[] ToArray()

##### Returns the queue as array of elements.

### int Count

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

## DoublyLinkedList

Your task is to take the implementation of the **SinglyLinkedList<T>** from the lab and make it a 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 it is such if no then throw InvalidOperationException with the appropriate message.

### T RemoveLast()

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

### T GetFirst()

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

### T GetLast()

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

## \* Reverse Numbers with a Stack

Write a program that reads **N integers** from the console and **reverses them using a stack**. Use the Stack<int> class from .NET Framework. Just put the input numbers in the stack and pop them. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 | 5 4 3 2 1 |
| 1 | 1 |
| (empty) | (empty) |
| 1 -2 | -2 1 |

## \* Calculate Sequence with a Queue

We are given the following sequence of numbers:

* S1 = N
* S2 = S1 + 1
* S3 = 2\*S1 + 1
* S4 = S1 + 2
* S5 = S2 + 1
* S6 = 2\*S2 + 1
* S7 = S2 + 2
* …

Using the Queue<T> class, write a program to print its first 50 members for given N. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2 | 2, 3, 5, 4, 4, 7, 5, 6, 11, 7, 5, 9, 6, … |
| -1 | -1, 0, -1, 1, 1, 1, 2, … |
| 1000 | 1000, 1001, 2001, 1002, 1002, 2003, 1003, … |

## \* Sequence N 🡪 M

We are given numbers **n** and **m**, and the following operations:

1. n 🡪 n + 1
2. n 🡪 n + 2
3. n 🡪 n \* 2

Write a program that **finds the shortest sequence of operations** from the list above that **starts from n and finishes in m**. If several shortest sequences exist, find the first one of them.

Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 10 | 3 -> 5 -> 10 |
| 5 -5 | (no solution) |
| 10 30 | 10 -> 11 -> 13 -> 15 -> 30 |

**Hint**: use a **queue** and the following algorithm:

1. create a queue of numbers
2. queue 🡨 n
3. while (queue not empty)
   1. queue 🡪 e
   2. if (e < m)
      1. queue 🡨 e + 1
      2. queue 🡨 e + 2
      3. queue 🡨 e \* 2
   3. if (e == m) Print-Solution; exit

The above algorithm either will find a solution, or will find that it does not exist. It cannot print the numbers comprising the sequence n 🡪 m.

To print the sequence of steps to reach **m**, starting from **n**, you will need to keep the previous item as well. Instead using a queue of numbers, use a queue of items. Each item will keep a number and a pointer to the previous item. The algorithms changes like this:

**Algorithm Find-Sequence (n, m)**:

1. create a queue of items { value, previous item }
2. queue 🡨 { n, **null** }
3. while (queue not empty)
   1. queue 🡪 item
   2. if (item.value < m)
      1. queue 🡨 { item.value + 1, item }
      2. queue 🡨 { item.value + 2, item }
      3. queue 🡨 { item.value \* 2, item }
   3. if (item.value == m) Print-Solution; exit

**Algorithm Print-Solution (item)**:

1. while (item not null)
   1. print item.value
   2. item = item.previous