Lab: Linear Data Structures

This document defines the **in-class exercises** assignments the "Data Structures" course @ Software University. You can submit your code in the SoftUni Judge System - https://judge.softuni.bg/Contests/574/Linear-Data-Structures-Stacks-and-Queues-Lab.

1. LinkedList<T>

Implement a data structure **`LinekdList<T>`** that holds a sequence of elements of generic type **T**. It should hold a **sequence of items in a sequence of linked nodes**. The list should support the following operations:

- **`int Count`** → returns the number of elements in the structure
- **`void AddFirst(T item)`** → adds an element to the start of the sequence
- **`void AddLast(T item)`** → adds an element to the end of the sequence
- **`T RemoveFirst()`** → removes an element from the start of the sequence and returns the element
- **`T RemoveLast()`** → removes an element from the end of the sequence and returns the element
- **`IEnumerable<T>`** → implement interface

`RemoveFirst()` and **`RemoveLast()`** methods should throw **`InvalidOperationException`** if the list is empty

Examples

```
static void Main(string[] args)
{
    LinkedList<int> list = new LinkedList<int>();

    list.AddFirst(1);
    list.AddLast(2);

    foreach (var item in list)
    {
        Console.WriteLine(item);
    }
}
```

Solution

Start by defining the class **`LinkedList<T>`**, you can define the Node class inside (in which case it doesn't need to be generic, as you will use **T** from the **`LinkedList`**)

Also, note that we are going to keep a reference to both the head and the tail of the list

```
public class LinkedList<T>
{
    public class Node
    {
        public Node(T value)
        {
            this.Value = value;
        }

        public T Value { get; set; }
        public Node Next { get; set; }
    }

    public Node Head { get; private set; }
    public Node Tail { get; private set; }
    public int Count { get; private set; }
}
```

First, implement **`AddFirst()`**. The utility method **`IsEmpty()`** checks if the Count is 0

```
public void AddFirst(T item)
{
    Node old = Head;

    this.Head = new Node(item);
    this.Head.Next = old;

    if (IsEmpty())
    {
        Tail = Head;
    }

    Count++;
}
```

`AddLast()` is almost the same

```
public void AddLast(T item)
{
    Node old = this.Tail;
    this.Tail = new Node(item);

    if (IsEmpty())
    {
        this.Head = this.Tail;
    }
    else
    {
        old.Next = this.Tail;
    }

    this.Count++;
}
```

Remove methods are a little bit more complicated. Start with **`RemoveFirst()`**

```
public T RemoveFirst()
{
    if (this.IsEmpty())
    {
        throw new InvalidOperationException();
    }

    T item = this.head.Value;

    this.head = this.head.Next;

    this.Count--;

    if (this.IsEmpty())
    {
        this.tail = null;
    }

    return item;
}
```

Method **`RemoveLast()`** uses a utility method **`GetSecondToLast()`** which gets the second to last element. Try to implement it yourself

```
public T RemoveLast()
{
    if (this.IsEmpty())
    {
        throw new InvalidOperationException();
    }

    T item = this.tail.Value;

    if (this.Count == 1)
    {
        this.head = this.tail = null;
    }
    else
    {
        Node newTail = this.GetSecondToLast();
        newTail.Next = null;
        this.tail = newTail;
    }

    this.Count--;

    return item;
}
```

The last thing to do is to implement **`IEnumerable<T>`**

```
public class LinkedList<T> : IEnumerable<T>
```

You need to implement two methods. The first is the actual that will do the work. The second one calls the first and you need it only for compatibility reasons

```
public IEnumerator<T> GetEnumerator()
{
    Node current = this.Head;
    while (current != null)
    {
        yield return current.Value;
        current = current.Next;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return this.GetEnumerator();
}
```