Homework: Algorithm Complexity and Linear Data Structures

This document defines the **homework assignments** for the ["Data Structures" course @ Software University](#).

Problem 1. Sum and Average

Write a program that reads from the console a sequence of integer numbers (on a single line, separated by a space). Calculate and print the **sum** and **average** of the elements of the sequence. Keep the sequence in **List<int>**.

Input	Output
4 5 6	Sum=15; Average=5
1 1	Sum=1; Average=1
	Sum=0; Average=0
10	Sum=10; Average=10
2 2 1	Sum=5; Average=1.66666666666667

Problem 2. Sort Words

Write a program that reads from the console a **sequence of words** (strings on a single line, separated by a space). **Sort** them alphabetically. Keep the sequence in **List<string>**.

Input	Output
wow softuni alpha	alpha softuni wow
Hi	hi
rakiya beer wine vodka whiskey	beer rakiya vodka whiskey wine

Problem 3. Longest Subsequence

Write a method that finds the **longest subsequence of equal numbers** in given **List<int>** and returns the result as new **List<int>**. If several sequences has the same longest length, return the leftmost of them. Write a program to test whether the method works correctly.

Input	Output
12 2 7 4 **3 3** 8	3 3
2 2 2 3 3 3	2 2 2
4 4 **5 5 5**	5 5 5
1 2 3	1
0	0

Problem 4. Remove Odd Occurences

Write a program that **removes** from given sequence all numbers that occur **odd number of times**.

Input	Output	Comments
1 2 3 4 **1**	1 1	2, 3 and 4 occur odd number of times (once). 1 occurs 2 times
1 2 3 **4 5** 3 **6** 7 **6** 7 **6**	3 3 7 7	1, 2, 4, 5 and 6 occurs odd number of times → removed

1 2 1 2 1 2		All numbers occur odd number of times → removed
3 7 **3** **3** 4 **3** 4 **3** 7	7 4 4 7	3 occurs odd number of times (5) → removed
1 1	1 1	All numbers occur even number of times → sequence stays unchanged

Problem 5. Count of Occurrences

Write a program that finds in given array of integers **how many times each of them occurs**. The input sequence holds numbers in range [0…1000]. The output should hold all numbers that occur at least once along with their number of occurrences.

Input	Output
3 4 4 2 3 3 4 3 2	2 -> 2 times 3 -> 4 times 4 -> 3 times
1000	1000 -> 1 times
0 0 0	0 -> 3 times
7 6 5 5 6	5 -> 2 times 6 -> 2 times 7 -> 1 times

Problem 6. Implement the Data Structure ReversedList<T>

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 2**. The reversed list should support the following operations:

- **Add(T item)** → adds an element to the sequence (grow twice the underlying array to extend its capacity in case the capacity is full)
- **Count** → returns the number of elements in the structure
- **Capacity** → returns the capacity of the underlying array holding the elements of the structure
- **this[index]** → the indexer should access the elements by **index** (in range **0** … **Count-1**) in the reverse order of adding
- **RemoveAt(index)** → removes an element by **index** (in range **0** … **Count-1**) in the reverse order of adding
- **IEnumerable<T>** → implement an enumerator to allow iterating over the elements in a **foreach** loop in a reversed order of their addition

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

Problem 7. * Distance in Labyrinth

We are given a labyrinth of size N x N. Some of its cells are empty (**0**) and some are full (**x**). We can move from an empty cell to another empty cell if they share common wall. Given a starting position (*****) calculate and fill in the array the minimal distance from this position to any other cell in the array. Use "**u**" for all unreachable cells.

0	0	0	x	0	x
0	x	0	x	0	x
0	*	x	0	x	0
0	x	0	0	0	0
0	0	0	x	x	0
0	0	0	x	0	x

→

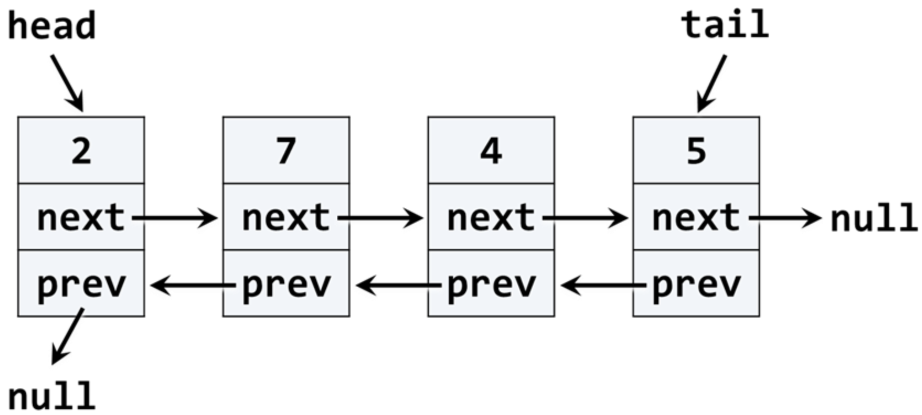
3	4	5	x	u	x
2	x	6	x	u	x
1	*	x	8	x	10
2	x	6	7	8	9
3	4	5	x	x	10
4	5	6	x	u	x

Input	Output
2 x0 *x	xu *x
3 000 0*0 000	212 1*1 212
6 000x0x 0x0x0x 0*x0x0 0x0000 000xx0 000x0x	345xux 2x6xux 1*x8x10 2x6789 345xx10 456xux

Problem 8. Implement a DoublyLinkedList<T>

You are given a project skeleton that contains unit tests for a **`DoublyLinkedList<T>`** data structure.

You have to implement a **doubly linked list** in C# or Java – a data structure that holds **nodes**, where each node knows its **next** and **previous** nodes:



Before starting, get familiar with the concept of doubly linked list: https://en.wikipedia.org/wiki/Doubly_linked_list.

The typical operations over a doubly linked list are **add** / **remove** element at **both ends** and **traverse**. By definition, the doubly linked list has a **head** (list start) and a **`tail`** (list end). Let's start coding!

Implement ListNode<T>

The first step when implementing a linked / doubly linked list is to understand that we need **two classes**:

- **`ListNode<T>`** class to hold a single list node (its value + next node + previous node)
- **`DoublyLinkedList<T>`** to hold the entire list (its head + tail + operations)

Now, let's write the **list node class**. It should hold a **`Value`** and a reference to its previous and next node. It can be inner class, because we will need it only internally from the doubly linked list class:

```
public class DoublyLinkedList<T> : IEnumerable<T>
{
        3 references
        private class ListNode<T>
        {
            1 reference
            public T Value { get; private set; }

            0 references
            public ListNode<T> NextNode { get; set; }

            0 references
            public ListNode<T> PrevNode { get; set; }

            0 references
            public ListNode(T value)
            {
                this.Value = value;
            }
        }
```

The class **`ListNode<T>`** is called **recursive data structure**, because it references itself recursively. It uses the **generic argument T** to avoid later specialization for any data type, e.g. **`int`**, **`string`** or **`DateTime`**. The **generic classes in C#** work similarly to **templates in C++** and **generic types in Java**.

Implement Head, Tail and Count

Now, let's define the **head** and **`tail`** of the doubly linked list:

```
public class DoublyLinkedList<T> : IEnumerable<T>
{
    5 references
    private class ListNode<T>...

    private ListNode<T> head;
    private ListNode<T> tail;

    9 references | 0/8 passing
    public int Count { get; private set; }
```

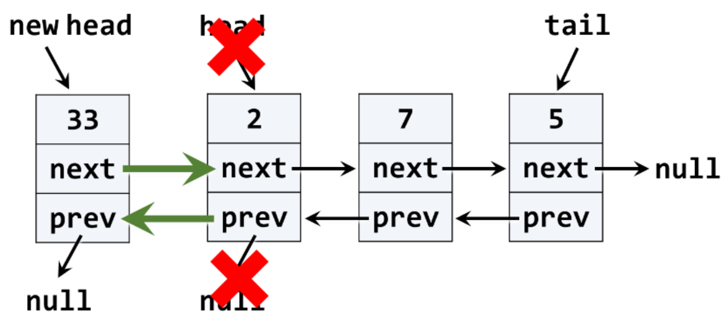
Implement AddFirst(T) Method

Next, implement the **`AddFirst(T element)`** method:

```
public void AddFirst(T element)
{
    if (this.Count == 0)
    {
        this.head = this.tail = new ListNode<T>(element);
    }
    else
    {
        var newHead = new ListNode<T>(element);
        newHead.NextNode = this.head;
        this.head.PrevNode = newHead;
        this.head = newHead;
    }
    this.Count++;
}
```

Adding an element at the start of the list (before its head) has **two scenarios** (considered in the above code):

- **Empty list** → add the new element as **head** and **`tail`** in the same time.
- **Non-empty list** → add the new element as **new head** and redirect the **old head** as second element, just after the new head.



The above graphic visualizes the process of inserting a new node at the start (**head**) of the list. The **red** arrows denote the removed pointers from the old head. The **green** arrows denote the new pointers to the new head.

Implement ForEach(Action) Method

We have a doubly linked list. We can add elements to it. But we cannot see what's inside, because the list still does not have a method to traverse its elements (pass through each of them, one by one). Now let's define the **`ForEach(Action<T>)`** method. In programming such a method is known as ["**visitor**" pattern](). It takes as an argument a function (action) to be invoked for each of the elements of the list. The algorithm behind this method is simple: start from **head** and pass to the next element until the last element is reached (its next element is **`null`**). A sample implementation is given below:

```
public void ForEach(Action<T> action)
{
    var currentNode = this.head;
    while (currentNode != null)
    {
        action(currentNode.Value);
        currentNode = currentNode.NextNode;
    }
}
```

Problem 9. Run the Unit Tests

Now we have the methods **`AddFirst(T)`** and **`ForEach(Action<T>)`**. We are ready to run the unit tests to ensure they are correctly implemented. Most of the **unit tests** create a doubly linked list, add / remove elements from it and then check whether the elements in the list are as expected. For example, let's examine this unit test:

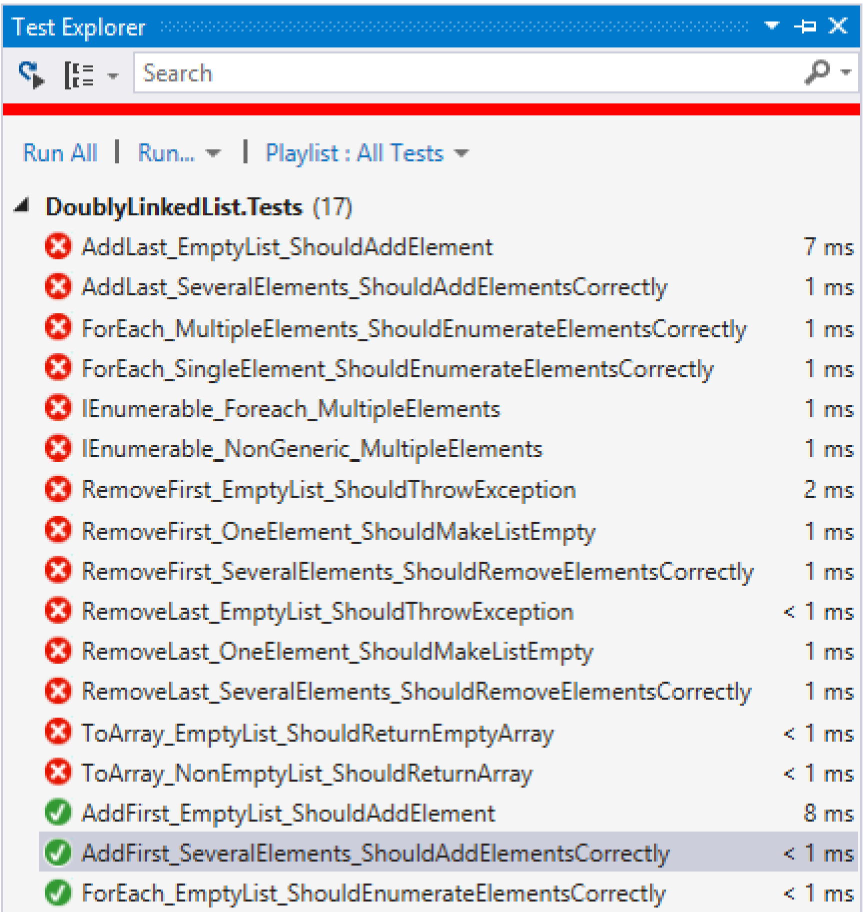
```
[TestMethod]
0 references
public void AddFirst_SeveralElements_ShouldAddElementsCorrectly()
{
    // Arrange
    var list = new DoublyLinkedList<int>();

    // Act
    list.AddFirst(10);
    list.AddFirst(5);
    list.AddFirst(3);

    // Assert
    Assert.AreEqual(3, list.Count);

    var items = new List<int>();
    list.ForEach(items.Add);
    CollectionAssert.AreEqual(items, new List<int>() { 3, 5, 10 });
}
```

If we **run the unit tests**, some of them will now pass:

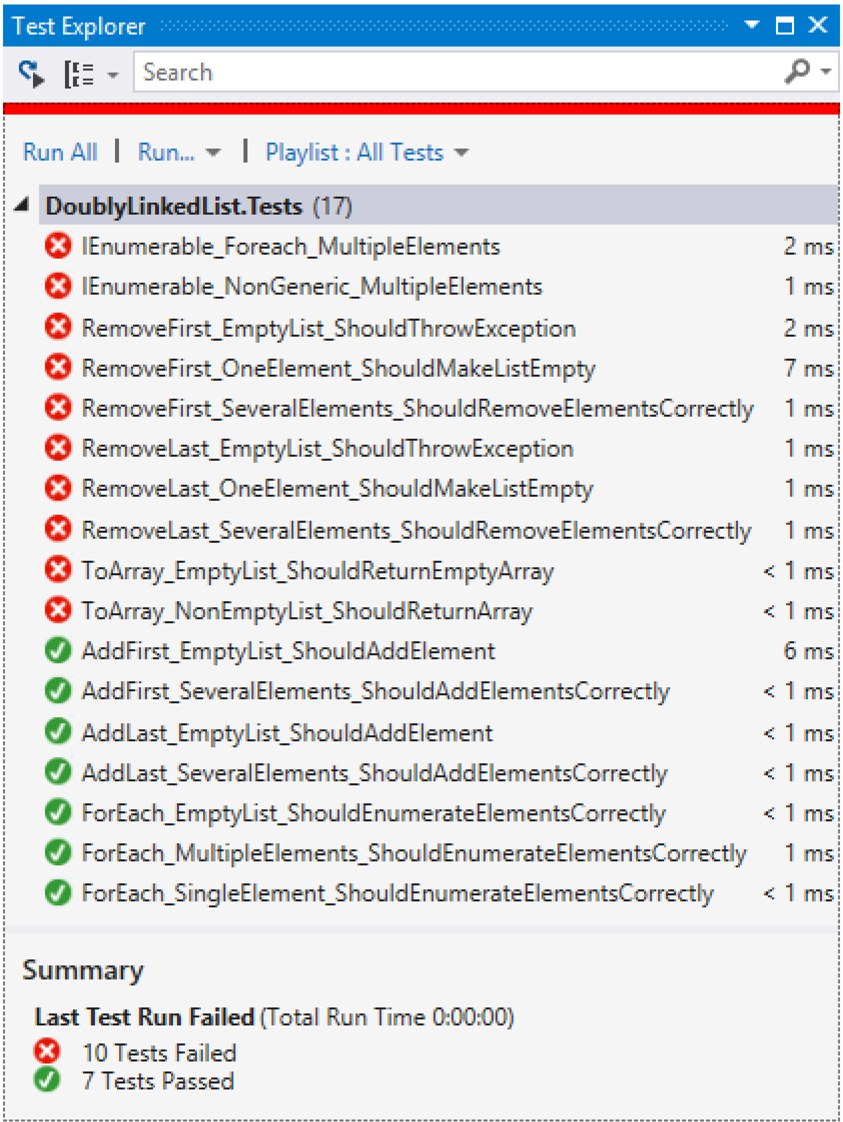


Implement AddLast(T) Method

Next, implement the **`AddLast(T element)`** method for appending a new element as the list **`tail`**. It should be very similar to the **`AddFirst(T element)`** method. The logic inside it exactly the same, but we append the new element at the **`tail`** instead of at the **head**. The code below is intentionally blurred. Write it yourself!

```
public void AddLast(T element)
{
}
```

Now **run the unit tests** again. You should have several more passed (green) tests:



Implement RemoveFirst() Method

Next, let's implement the method **`RemoveFirst()` → T**. It should **remove the first element** from the list and move its **head** to point to the second element. The removed element should be returned as a result from the method. In case of empty list, the method should throw an exception. We have to consider the following three cases:

- **Empty list** → throw and exception.
- **Single element in the list** → make the list empty (**head** == **`tail`** == **`null`**).
- **Multiple elements in the list** → remove the first element and redirect the head to point to the second element (**head = `head.NextNode`**).

A sample implementation of **`RemoveFirst()`** method is given below:

```
public T RemoveFirst()
{
    if (this.Count == 0)
    {
        throw new InvalidOperationException("List empty");
    }

    var firstElement = this.head.Value;
    this.head = this.head.NextNode;
    if (this.head != null)
    {
        this.head.PrevNode = null;
    }
    else
    {
        this.tail = null;
    }

    this.Count--;
    return firstElement;
}
```

Run the **unit tests** to ensure the method is correctly implemented:

Problem 10. Implement RemoveLast() Method

Next, let's implement the method **`RemoveLast()` → T**. It should **remove the last element** from the list and move its **`tail`** to point to the element before the last. It is very similar to the method **`RemoveFirst()`**, so you are free to implement it yourself. The code below is intentionally blurred:

```
public T RemoveLast()
{
}
```

Problem 11. Implement ToArray() Method

Now, implement the next method: **`ToArray()`** → **`T[]`**. It should copy all elements of the linked list to an array of the same size. You could use the following steps to implement this method:

- Allocate an array **`T[]`** of size **`this.Count`**.
- Pass through all elements of the list (from **head** to **`tail`**) and fill them to **`T[0]`**, **`T[1]`**, …, **`T[Count-1]`**.
- Return the array as result.

Write yourself the blurred code in the method **`ToArray()`**:

```
public T[] ToArray()
{

}
```

Implement IEnumerable<T>

Collection classes in C# and .NET Framework (like arrays, lists and sets) implement the system interface **`IEnumerable<T>`** to enable the **`foreach`** iteration over their elements. The C# keyword **`foreach`** calls internally the following method:

```
public IEnumerator<T> GetEnumerator()
{
    // TODO: implement me
}
```

This method returns **`IEnumerator<T>`**, which can move to the next element and read the current element. In programming, this is known [as "**iterator**" pattern]() (**enumerator**).

We will use [the "**`yield return`**" C# statement]() to simplify the implementation of the iterator:

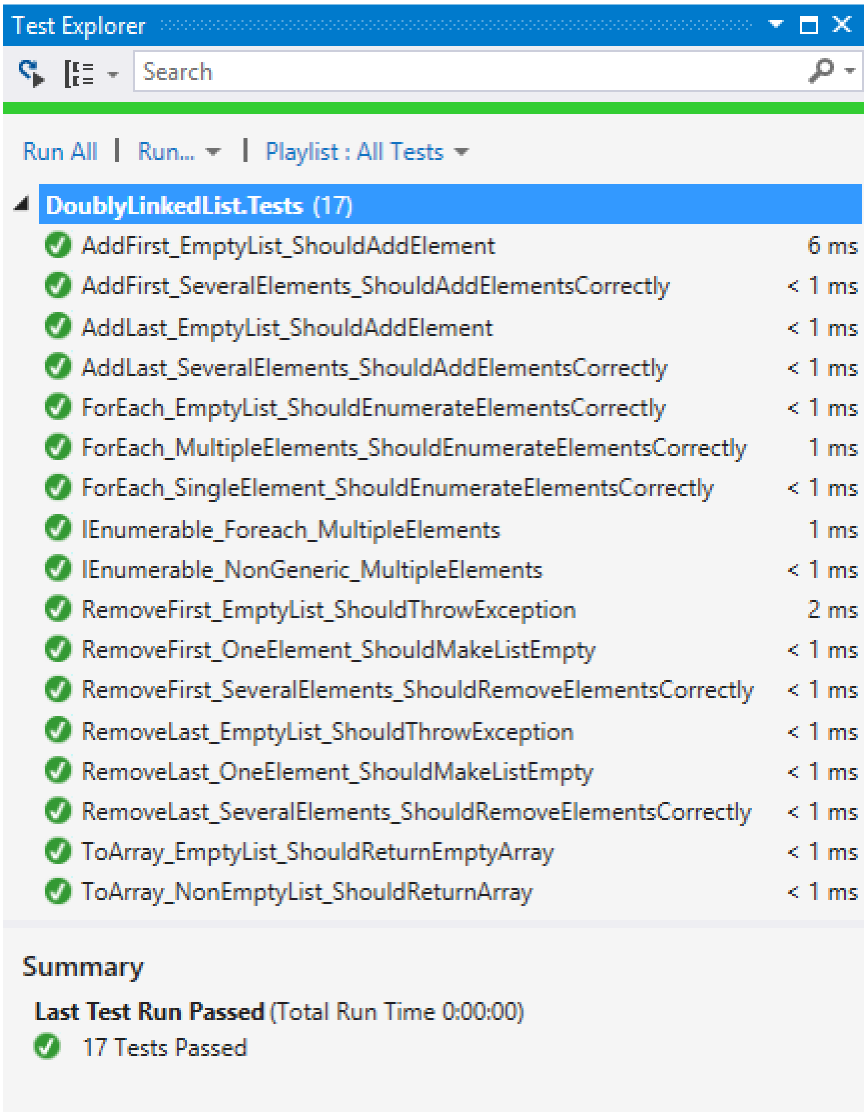
```
public IEnumerator<T> GetEnumerator()
{
    var currentNode = this.head;
    while (currentNode != null)
    {
        yield return currentNode.Value;
        currentNode = currentNode.NextNode;
    }
}
```

The above code will enable using the **`DoublyLinkedList<T>`** in **`foreach`** loops.

The last unimplemented method is the **non-generic enumerator**:

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

Finally, **run the unit tests** to ensure all of them pass correctly:



Congratulations! You have implemented your doubly linked list.