# Workshop

# Custom Doubly Linked List

## Overview

In this workshop, we are going to create another custom data structure, which has similar functionalities as the C# doubly linked list. Just like the structures from the previous workshop, our custom doubly linked list will work only with integers. It will have the following functionalities:

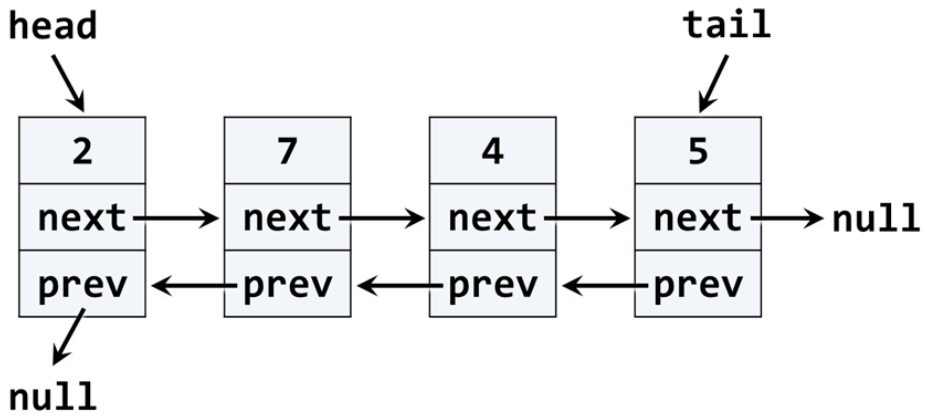
- **`void AddFirst(int element)`** – adds an element at the beginning of the collection
- **`void AddLast(int element)`** – adds an element at the end of the collection
- **`int RemoveFirst()`** – removes the element at the beginning of the collection
- **`int RemoveLast()`** – removes the element at the end of the collection
- **`void ForEach()`** – goes through the collection and executes a given action
- **`int[] ToArray()`** – returns the collection as an array

Feel free to implement your own functionality or to write the methods by yourself.

## 1. Implement the CustomDoublyLinkedList class

### Details about the structure

The **doubly linked list** is a structure that resembles a list, but has different functionalities. Each element in it "knows" about the previous one, if there is such, and the next one, again, if there is such. This is possible, because the doubly linked list has **nodes** and each node has **two reference properties** pointing to other nodes and **a value property,** which contains some kind of data. By definition, the doubly linked list has a **head** (list start) and a **`tail`** (list end). The typical operations over a doubly linked list are **add** / **remove** an element at **both of the endings** and **traverse**. If you are interested, you can find more detailed information here: https://en.wikipedia.org/wiki/Doubly_linked_list. This figure shows how the structure looks:




Now, that we are somewhat familiar with the doubly linked list, we can proceed to the implementation of our own custom doubly linked list. We will try to implement the main functionalities, but you are free to add other ones if you are interested.

### Implementation

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

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

Now, let's create the **`ListNode`** class. It should hold a **`Value`** and a reference to its previous and next node. We can do that inside the **`DoublyLinkedList`** class, because we will use it only internally inside it. Here is how the class should look:

```
public class DoublyLinkedList
{
    3 references
    private class ListNode
    {
        1 reference
        public int Value { get; set; }
        0 references
        public ListNode NextNode { get; set; }
        0 references
        public ListNode PreviousNode { get; set; }
        0 references
        public ListNode(int value)
        {
            this.Value = value;
        }
    }
}
```

The class **`ListNode`** is called a **recursive data structure**, because it references itself recursively. In this case our nodes' **`Value`** property will be of type **`int`.** At some point throughout the next course from this module, we will be able to change that and make the structure **generic**, which means it will be able to work with any type.

## Implement Head, Tail and Count

Now, let's define the **`head`** and **`tail`** of the doubly linked list. They will be of type **`ListNode`**:

```
public class DoublyLinkedList
{
    5 references
    private class ListNode...
    private ListNode head;
    private ListNode tail;

    0 references
    public int Count { get; private set; }
}
```

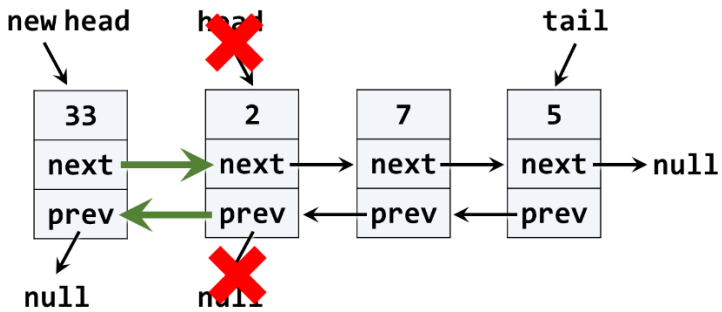
## Implement AddFirst(int) Method

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

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

Adding an element at the beginning 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 AddLast(int) Method

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

```
public void AddLast(int element)
{
    if (this.Count == 0)
    {
        this.head = this.tail = new ListNode(element);
    }
    else
    {
        var newTail = new ListNode(element);
        newTail.PreviousNode = this.tail;
        this.tail.NextNode = newTail;
        this.tail = newTail;
    }
    this.Count++;
}
```

## Implement RemoveFirst() Method

Next, let's implement the method **`RemoveFirst()` → `int`**. 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 an 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 int RemoveFirst()
{
    if (this.Count == 0)
    {
        throw new InvalidOperationException("The list is empty");
    }

    var firstElement = this.head.Value;
    this.head = this.head.NextNode;
    if(this.head != null)
    {
        this.head.PreviousNode = null;
    }
    else
    {
        this.tail = null;
    }
    this.Count--;
    return firstElement;
}
```

## Implement RemoveLast() Method

Next, let's implement the method **`RemoveLast()` → `int`**. 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()`**.

```
public int RemoveLast()
{
    if(this.Count == 0)
    {
        throw new InvalidOperationException("The list is empty");
    }
    var lastElement = this.tail.Value;
    this.tail = this.tail.PreviousNode;
    if(this.tail != null)
    {
        this.tail.NextNode = null;
    }
    else
    {
        this.head = null;
    }
    this.Count--;
    return lastElement;
}
```

## 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<int>)`** method. In programming, such a method is known as a **"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<int> action)
{
    var currentNode = this.head;
    while(currentNode != null)
    {
        action(currentNode.Value);
        currentNode = currentNode.NextNode;
    }
}
```

For example if you want to print all of the elements you can use the following code:

```
list.ForEach(n => Console.WriteLine(n));
```

Where **list** is **DoublyLinkedList** type object.

## Implement ToArray() Method

Now, implement the next method: **`ToArray()` → `int[]`**. 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 **`int[]`** of size **`this.Count`**.
- Pass through all elements of the list and fill them to **`int[0]`**, **`int[1]`**, …, **`int[Count-1]`**.
- Return the array as result.

```
public int[] ToArray()
{
    int[] array = new int[this.Count];
    int counter = 0;
    var currentNode = this.head;
    while(currentNode != null)
    {
        array[counter] = currentNode.Value;
        currentNode = currentNode.NextNode;
        counter++;
    }
    return array;
}
```

Congratulations! You have implemented your doubly linked list.