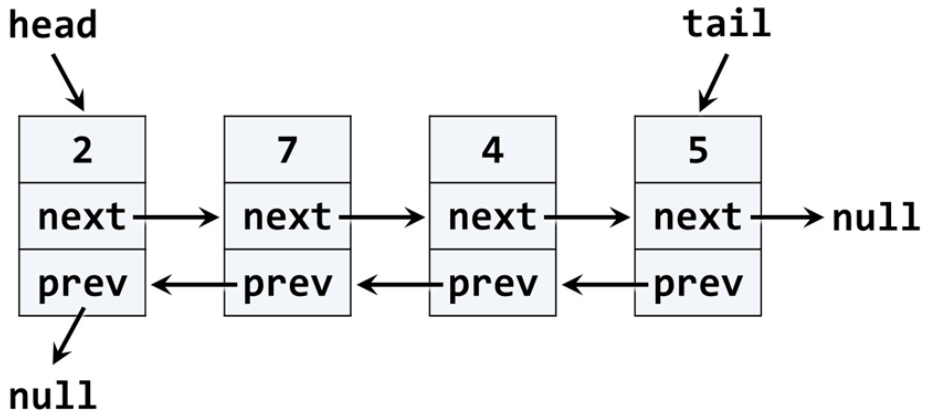
Упражнение: Реализация на DoublyLinkedList<T>

Трябва да реализирате **двусвързан списък** в C# – структура от данни, която има възли (**nodes**), където всеки възел съдържа информация, както за **следващия,** така и за **предишния** възел:



Операциите върху двусвързан списък са добавяне (**add)** / премахване (**remove)** на елемент от **двата края** и **обхождане**. По дефиниция, двусвързания списък има **head** (начало на списъка) и **`tail`** (край на списъка). Да започваме!

1. Реализация на ListNode<T>

Първата стъпка при имеплемтиране на свързан / двусвързан списък е да разберем, че ни трябват **два класа**:

- **`ListNode<T>`** клас, който съдържа един възел (стойност + следващ възел + преден възел)
- **`DoublyLinkedList<T>`** клас, който съдържа целия списък

Сега, нека да напишем класа за **възела**. Той трябва да пази **`Value`** и информация за неговия предишен и следващ възел. Може да бъде и вътрешен клас, защото ще ни трябва само вътрешно от другия:

```
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;
        }
    }
```

`ListNode<T>` се нарича **рекурсивна структура от данни**, защото „сочи“ сам към себе си рекурсивно. Той използва **шаблонен аргумент T** за да избегне по-късна конкретика за даден тип данни.

2. Имплементирайте Head, Tail и Count

Сега нека си дефинираме **head** и **`tail`** на двусвързания списък:

```
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; }
```

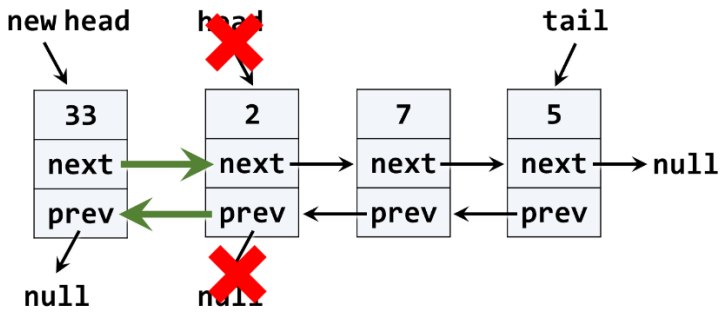
3. Имплементиране на AddFirst(T) метода

Сега, имплементираме **`AddFirst(T element)`** метода:

```
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++;
}
```

Добавянето на елемент към началото на списъка има **два случая**:

- **Празен списък** → добавя нов елемент като свой **head** и **`tail`** едновременно.
- **Непразен списък** → Добавя нов елемент като нов **head** и задава **старият head** като втори елемент, след head.



Графиката визуализира процеса на добавянето на нов възел в началото (**head**) на списъка. **Червените** стрелки отбелязват премахнатите указатели от старият head. **Зелените** стрелки отбелязват новите указатели към новият head.

4. Имплементирайте ForEach(Action) метод

Ние имаме двусвързан списък. Можем да добавяме елементи към него, но не можем да видим какво има вътре, защото списъкът все още няма метод за обхождане на елементите му. Сега, нека да дефинираме **`ForEach(Action<T>)`** метод. Той взема за параметър функция (действие, action), която ще бъде извикана

за всеки един от елементите в списъка. Алгоритъмът зад този метод е прост: започвайки от **head** и преминавайки напред към следващия елемент, докато не стигнем до последният елемент (неговият следващ елемент е **null**). Ето и проста имплементация:

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

5. Имплементиране на AddLast(T) метод

Сега, имплементирайте **AddLast(T element)** метод, за добавяне на нов елемент като **tail**. Това трябва да бъде много подобно на **AddFirst(T element)** метода. Логиката вътре в него е точно същата, но елемента се добавяне към **tail,** вместо към **head**. Кодът по-долу е нарочно замъглен. Опитайте се сами!

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

6. Имплементирайте RemoveFirst() метода

Сега, нека да имплементираме **RemoveFirst() → T**. Той трябва да премахва първия елемент и да премества **head** към втория елемент. Премахнятията елемент трябва да бъде върнат като резултат от метода. В случай на празен списък, методът трябва да хвърли изключение. Трябва да разгледаме следните случаи:

- **Празен списък** → хвърляме изключение.
- **Единствен елемент в списъка** → прави списъка празен (**head** == **tail** == **null**).
- **Множество елементи в списъка** → премахваме първия елемент и задаваме head да сочи към втория елемент (**head = head.NextNode**).

Примерна реализация на **RemoveFirst()** метода е дадена по-долу:

```
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;
}
```

7. Имплементация на RemoveLast() метода

Сега, нека добавим и метод **`RemoveLast()` → `T`**. Той трябва да **махне последния елемент** от списъка и да промени неговия **`tail`**, така щото да сочи към елемента преди последния. Идеята е мнгоо подобна на **`RemoveFirst()`**, така че се опитайте сами. Кодът по-долу нарочно е замъглен:

8. Имплементация на ToArray() метод

Сега, имплементирайте следващия метод: **`ToArray()` → `T[]`**. Той трябва да копира всички елементи на свързания списък към масив със същия размер. Може да използвате следните стъпки, за да реализирате този метод:

- Заделете място за масив **`T[]`** с размер **`this.Count`**.
- Обходете всички елементи в списъка (от **`head`** към **`tail`**) и ги присвоете в **`T[0]`**, **`T[1]`**, …, **`T[Count-1]`**.
- Върнете масива като резултат.

Напишете сами кодът за **`ToArray()`**:

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

9. Имплементирайте IEnumerable<T>

Класовете за колекции в C# и .NET Framework (като масиви, списъци и т.н.) имплементират системния интерфейс (повече за интерфейси в курсът за ООП) **`IEnumerable<T>`**. Това се прави, за да може да направим обхождане с **`foreach`** по елементите. В C# ключовата дума **`foreach`** извиква вътрешно следния метод:

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

Този метод връща **`IEnumerator<T>`**, който може да мине към следващия елемент и да прочете текущия елемент. Това е известно още и като итератор (**enumerator**).

Ще използване [`"yield return" от C#`], за да опростим имплементацията на итератора:

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

Кодът по-горе ще ви позволи да използвате **`DoublyLinkedList<T>`** и **`foreach`** цикли.

Последният неимплементиран метод е следния:

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