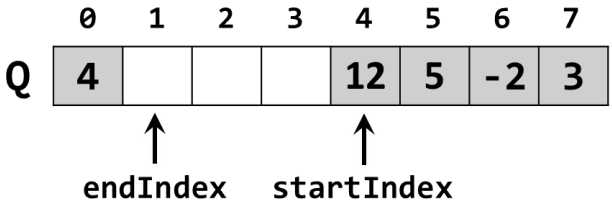
Упражнения: Имплементация на опашка

Имплементирайте кръгова опашка, базирана на масив в C# – структура от данни, която съдържа елементи и следва принципа FIFO (**F**irst **I**n, **F**irst **O**ut – първи вътре, първи вън), като използвате фиксиран вътрешен **капацитет**, който се удвоява, когато се запълни:



На фигурата по-горе, елементите {12, 5, -2, 3, 4} стоят в масив с фиксиран капацитет от 8 елемента. Капацитета на опашката е 8, броят на елементите е 5, а 3 клетки стоят празни. **`startIndex`** ни показва първият непразен елемент в опашката. **`endIndex`** ни показва мястото точно след последния непразен елемент в опашката – мястото, където следващият елемент ще бъде добавен към опашката. Забележете, че опашката е **кръгова**: след елемента на последна позиция 7 идва елемент на позиция 0.

1. CircularQueue<T>

Използвайте следният скелет за класа:

```
public class CircularQueue<T>
{
    private const int DefaultCapacity = 4;
    public int Count { get; private set; }
    public CircularQueue(int capacity = DefaultCapacity) { … }
    public void Enqueue(T element) { … }
    public T Dequeue() { … }
    public T[] ToArray() { … }
}
```

2. Създайте вътрешната информация за опашката

Първата стъпка е да създадете вътрешна информация, която пази елементите, както и началният+крайният индекс:

- **`T[] elements`** – масив, който държи елементите на опашката
  - Непразните клетки пазят елементите
  - Празните клетки са свободни за добавяне на нови елементи
  - Дължината на масива (**Length)** пази капацитета на опашката
- **`int startIndex`** – пази началния индекс (индекса на първия влезнал елемент в опашката)

- **`int endIndex`** – пази крайния индекс (индекса в масива, който е непосредствено след последния добавен елемент)
- **`int Count`** – пази информация за броя елементи в опашката

Кодът би изглеждал по подобен начин:

```
public class CircularQueue<T>
{
    private T[] elements;
    private int startIndex = 0;
    private int endIndex = 0;

    17 references | 0/5 passing
    public int Count { get; private set; }
```

3. Направете конструктор

Сега, нека да имплементираме конструктор. Негоава цел е да заделя място за масива в рамките на **`CircularQueue<T>`** класа. Ще имаме два констуктора:

- Конструктор без параметри – трябва да задели 16 елемента (16 е капацитета по подразбиране в началото за опашката)
- Констуктор с параметър **`capacity`** – заделя масива с конкретен капацитет

4. Имплементиране на Enqueue(…) метод

Нека да имплементираме **`Enqueue(element)`** метода, който добавя нов елемент в края на опашката:

```
public void Enqueue(T element)
{
    if (this.Count >= this.elements.Length)
    {
        this.Grow();
    }
    this.elements[this.endIndex] = element;
    this.endIndex = (this.endIndex + 1) % this.elements.Length;
    this.Count++;
}
```

Как работи? Първо, ако опашката е пълна, **увеличава** я (т.е. нейния капацитет става двойно по-голям). След това, добавя новият елемент на позиция **`endIndex`** (индексът, който е точно след последния елемент), а след това премества индекса с една позиция надясно, както и увеличава вътрешния брояч **`Count`**.

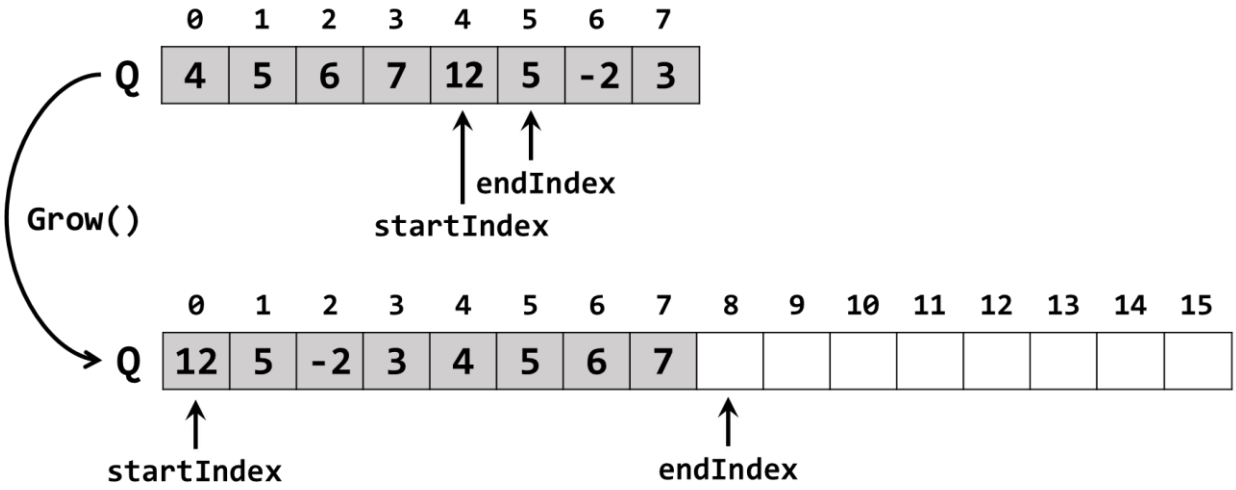
Забележете, че опашката е кръгова, така че елемента след последния елемент (**`this.elements.Length-1`**) е **`0`**.

Така стигаме до **формула**: Елементът следващ **`p`** е на позиция **`(p + 1) % capacity`**. В кода имаме:

```
(this.endIndex + 1) % this.elements.Length
```

5. Имплементиране на Grow() метод

`Grow()` методът се извиква, когато опашката е със запълнен капацитет (**`capacity == Count`**) и искаме да добавим нов елемент. **`Grow()`** методът трябва да задели нов масив с **удвоен капацитет** и да премести всички елементи от стария масив в новия масив:



Кодът за увеличаване на капацитета може да изглежда по подобен начин:

```
private void Grow()
{
    var newElements = new T[2 * this.elements.Length];
    this.CopyAllElementsTo(newElements);
    this.elements = newElements;
    this.startIndex = 0;
    this.endIndex = this.Count;
}
```

Важна част от "**уголемяването**" е да се **копират елементите от стария масив в новия**. Това може да се случу ето така:

```
private void CopyAllElementsTo(T[] resultArr)
{
    int sourceIndex = this.startIndex;
    int destinationIndex = 0;
    for (int i = 0; i < this.Count; i++)
    {
        resultArr[destinationIndex] = this.elements[sourceIndex];
        sourceIndex = (sourceIndex + 1) % this.elements.Length;
        destinationIndex++;
    }
}
```

6. Имплементиране на Dequeue() метод

Сега е ред на **`Dequeue()`** метода. Неговата цел е да се върне и да се премахне от опашката първият добавен елемент (той се намира на позиция **`startIndex`**). Кодът е както следва:

```
public T Dequeue()
{
    if (this.Count == 0)
    {
        throw new InvalidOperationException("The queue is empty!");
    }

    var result = this.elements[startIndex];
    this.startIndex = (this.startIndex + 1) % this.elements.Length;
    this.Count--;
    return result;
}
```

Как работи? Ако опашката е празна, се хвърля изключение. В противен случай, първият елемент от опашката се взема; **`startIndex`** се отмества нататък; **`Count`** се намаля.

7. Имплементиране на ToArray() Method

Сега нека си направим и **`ToArray()`** метод. Той трябва да заделя масив с размер **`this.Count`** и да **копира всички елементи от опашката** в него. Ние вече имаме метод за копиране на елементите, така че този път ще се справим по-лесно и кратко. Кодът е замъглен нарочно. Опитайте се сами.

```
public T[] ToArray()
{
    var resultArr = new T[this.Count];
    CopyAllElementsTo(resultArr);
    return resultArr;
}
```