

Упражнения: Опашки

1. Казвано ли е дадено число?

Напишете програма, която чете от конзолата **брой N** и после **последователност от N цели числа**, всяко един ред, разделени с интервал и накрая число, което се проверява дали съществува в първата група числа. Ако числото е сред тях, се извежда “{number} Exists in the List”, в противен случай - “{Number} Not exists in the List”.

2. По-големи от средното

Напишете програма, която чете от конзолата **брой N** и после **последователност от N цели числа**, всяко един ред, разделени с интервал. Да се изведат **числата от първия списък, които са по-големи от средното аритметично** на всички въведени числа.

3. Вмъкване на число

Напишете програма, която чете от конзолата **възходяща последователност от цели числа** на един ред, разделени с интервал и на втори ред **число, което се вмъква** на такава позиция, че **новополучената редица отново да е възходяща подредена**. Изведете **новополучената редица**. Опитайте се да използвате възможно най-малко памет.

4. Вмъкване на число и сравнение

Напишете програма, която чете от конзолата **възходяща последователност от цели числа** на един ред, разделени с интервал и на втори ред **число, което се вмъква** на такава позиция, че **новополучената редица отново да е възходяща подредена**. Изведете **двете редици** – тази от преди вмъкването и другата – след вмъкването на числото.

5. Изчислете редицата с опашка

Дадена е следната последователност от числа:

- $S_1 = N$
- $S_2 = S_1 + 1$
- $S_3 = 2 * S_1 + 1$
- $S_4 = S_1 + 2$
- $S_5 = S_2 + 1$
- $S_6 = 2 * S_2 + 1$
- $S_7 = S_2 + 2$
- ...

Използвайте класа **Queue<T>** и напишете програма, която извежда първите 50 члена за даденото N

Примери:

Вход	Изход
2	2, 3, 5, 4, 4, 7, 5, 6, 11, 7, 5, 9, 6, ...

-1	-1, 0, -1, 1, 1, 1, 2, ...
1000	1000, 1001, 2001, 1002, 1002, 2003, 1003, ...

6. * Редица $N \rightarrow M$

Дадени са числата n и m и следните операции:

- a) $n \rightarrow n + 1$
- b) $n \rightarrow n + 2$
- c) $n \rightarrow n * 2$

Напишете програма, която **намира най-късата редица от операции** от списъка по-долу, който **започва от n и завършва в m** . Ако съществуват няколко най-къси редици, намерете първата от тях.

Примери:

Вход	Изход
3 10	3 -> 5 -> 10
5 -5	(няма решение)
10 30	10 -> 11 -> 13 -> 15 -> 30

Подсказка: използвайте **опашка** и следващия алгоритъм:

1. създайте опашка от числа
2. опашка $\leftarrow n$
3. докато (опашката не е празна)
 1. опашка $\rightarrow e$
 2. ако ($e < m$)
 - i. опашка $\leftarrow e + 1$
 - ii. опашка $\leftarrow e + 2$
 - iii. опашка $\leftarrow e * 2$
 3. ако ($e == m$) Print-Solution; край

С по-горния алгоритъм ще намерите решение, или ще откриете, че то не съществува. Той не може да отпечата числата, включващи редицата $n \rightarrow m$.

За да отпечатате редицата от стъпки, за да достигне m , започвайки от n , ще трябва да запазите също и предишния елемент. Вместо с опашка от числа, използвайте опашка от елементи. Всеки елемент ще запази число и указател към предишния елемент. Промените в алгоритъма са примерно такива:

Алгоритъм Find-Sequence (n, m):

1. създайте опашка от елемент {стойност, предходен_елемент}
2. опашка $\leftarrow \{ n, \text{null} \}$
3. докато (опашката не е празна)
 1. опашка \rightarrow елемент
 2. ако (елемент.стойност $< m$)
 - i. опашка $\leftarrow \{ \text{елемент.стойност} + 1, \text{елемент} \}$

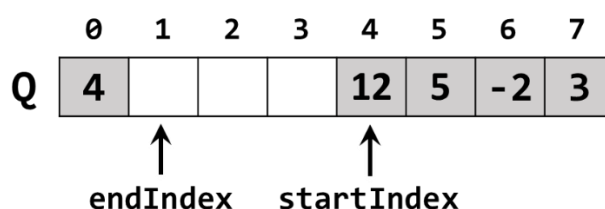
- ii. `queue ← { элемент.стойност + 2, элемент }`
- iii. `queue ← { элемент.стойност * 2, элемент }`
- 3. ако `(элемент.стойност == m)` Print-Solution; **край**

Алгоритъм Print-Solution (item):

1. докато (елемента не е null)
 1. отпечатай элемент.стойност
 2. элемент=элемент.предходен_элемент

7. * Имплементиране на кръгова опашка

Имплементирайте кръгова опашка, базирана на масив в C# – структура от данни, която съдържа елементи и следва принципа FIFO (First In, First Out – 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.

Стъпка 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() { ... }
}
```

Стъпка 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; }
```

Стъпка 3. Направете конструктор

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

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

Стъпка 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**.

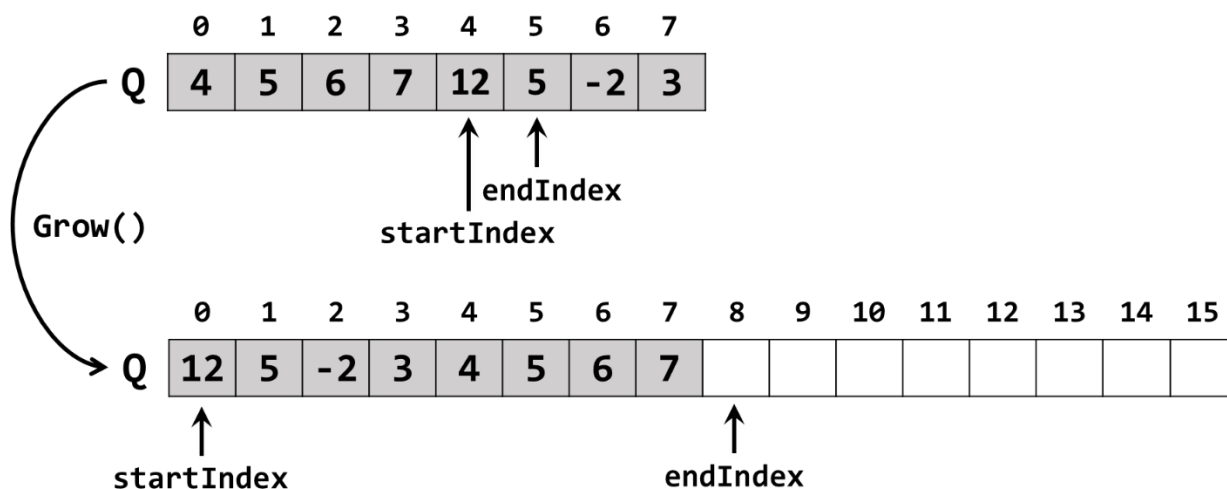
Забележете, че опашката е кръгова, така че елемента след последния елемент (**this.elements.Length - 1**) е **0**.

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

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

Стъпка 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 случи 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++;
    }
}
```

Стъпка 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.

Стъпка 7. Имплементиране на ToArray() Method

Сега нека си направим и **ToArray()** метод. Той трябва да заделва 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.

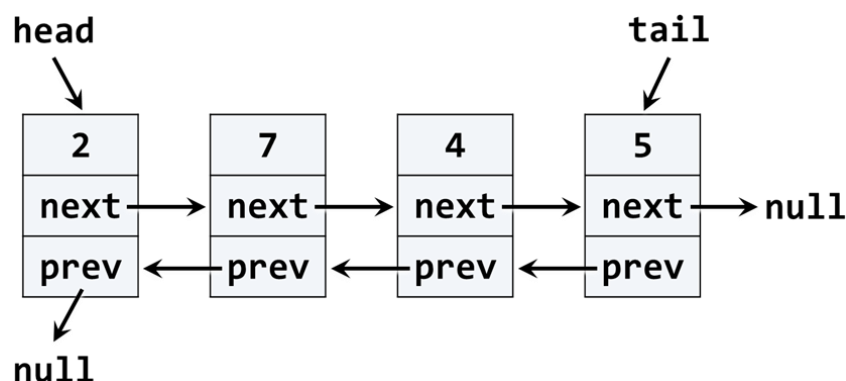
```

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

```

8. * Имплементиране на свързана опашка

Имплементирайте опашката използвайки "двусвързан списък":



Използвайте този код като скелет:

```

public class LinkedQueue<T>
{
    public int Count { get; private set; }
}

```

```
public void Enqueue(T element) { ... }  
public T Dequeue() { ... }  
public T[] ToArray() { ... }  
  
private class QueueNode<T>  
{  
    public T Value { get; private set; }  
    public QueueNode<T> NextNode { get; set; }  
    public QueueNode<T> PrevNode { get; set; }  
}  
}
```

Разгледайте и модифицирайте кода за **DoublyLinkedList<T>** класа. Ако опашката е празна, **Dequeue()** трябва да хвърля **InvalidOperationException**.