# Упражнения: Опашки

## Казвано ли е дадено число?

Напишете програма, която чете от конзолата **брой N** и после **последователност от N цели числа**, всяко на отделен ред 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”**.

## По-големи от средното

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

## Числата в диапазона

Напишете програма, която чете от конзолата цяло число за **долна** и **горна граница** на диапазон (всяко на отделен ред), после **брой N** и после **последователност от N цели числа**, всяко на отделен ред. Да се изведат **числата** от последователността, **които са в диапазона.** Гарантирано е, че долната граница е по-малка или равна на горната. Да се използва възможно най-малко памет.

## Вмъкване на число

Напишете програма, която чете от конзолата **брой N** и после **възходяща** **последователност от цели числа**, всяко на отделен ред и накрая **число,** което трябва да **се вмъкне в редицата** на такава позиция, че **новополучената редица отново да е възходящо подредена**. **Изведете новополучената редица**. Опитайте се да използвате **възможно най-малко памет**.

## Вмъкване на число и сравнение

Напишете програма, която чете от конзолата **брой N** и после **възходяща** **последователност от цели числа**, всяко на отделен ред и накрая **число,** което трябва да **се вмъкне в редицата** на такава позиция, че **новополучената редица отново да е възходящо подредена**. **Изведете двете редици** – тази от преди вмъкването и другата – след вмъкването на числото.

## Отделно положителните, отделно отрицателните

Напишете програма, която чете от конзолата **редица от цели числа.** Числата са подадени на един ред, разделени с интервал. Трябва да се изведат два реда - на първия всички положителни числа от горната редица, а на втория всички отрицателни числа - в реда, в който са подадени, отново разделени с интервал.

## Сортиране чрез опашка

Напишете програма, която чете от конзолата **редица от цели числа.** Числата са подадени на един ред, разделени с интервал. Трябва да се изведе на един ред **същата редица, подредена във възходящ ред**, числата да отново отделени с интервал. Задачата да бъде решена, като за подредената редица бъде използвана само структура от данни опашка. Масиви и списъци не е разрешено да се ползват за сортирането, а само за съхраняване на входните данни.

## Обединяване на подредени редици

Напишете програма, която чете от конзолата **брой N** и после **последователност от N двойки цели числа**, всяка на отделен ред. Гарантирано е, че числата от първата колонка са във възходящ ред. За втората важи същото, но първото число от всяка двойка може да е по-голямо от второто, равно или по-малко от него. Да се обединят всички числа така, че **новополучената редица отново да е възходящо подредена**. **Изведете новополучената редица** на един ред, числата да са отделени с интервали.

## Числова редица

Дадена е следната последователност от числа:

* S1 = N
* S2 = S1 + 1
* S3 = 2\*S1 + 1
* S4 = S1 + 2
* S5 = S2 + 1
* S6 = 2\*S2 + 1
* S7 = S2 + 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, … |

## \* Редица N 🡪 M

Дадени са числата n и m и следните операции:

1. n 🡪 n + 1
2. n 🡪 n + 2
3. n 🡪 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 🡨 n
3. докато (опашката не е празна)
   1. опашка 🡪 e
   2. ако (e < m)
      1. опашка 🡨 e + 1
      2. опашка 🡨 e + 2
      3. опашка 🡨 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 🡪 m.

За да отпечатате редицата от стъпки, за да достигне м, 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)**:

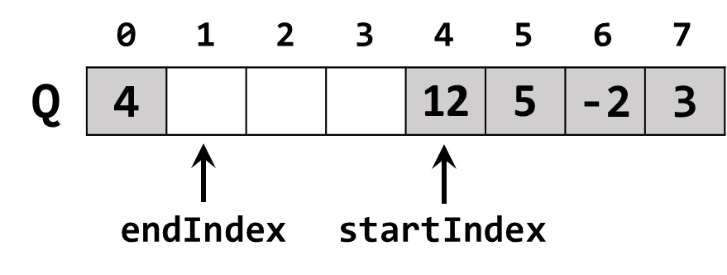
1. създайте опашка от елемент {стойност, предходен\_елемент }
2. опашка 🡨 { n, **null** }
3. докато (опашката не е празна)
   1. опашка 🡪 елемент
   2. ако (елемент.стойност < m)
      1. опашка 🡨 { елемент.стойност + 1, елемент}
      2. queue 🡨 { елемент.стойност + 2, елемент }
      3. queue 🡨 { елемент.стойност \* 2, елемент }
   3. ако (елемент.стойност == m) Print-Solution; **край**

**Алгоритъм Print-Solution (item)**:

1. докато (елемента не е null)
   1. отпечатай елемент.стойност
   2. елемент=елемент.предходен\_елемент

## \* Имплементиране на кръгова 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.

### CircularQueue<T>

Използвайте следният скелет за класа:

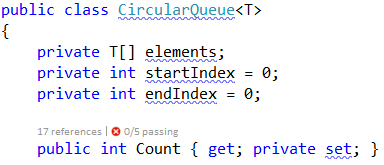
|  |
| --- |
| public class CircularQueue<T>  {  private const int DefaultCapacity = 4;  public int Count { get; private set; }  public CircularQueue(int capacity = DefaultCapacity) { … }  public void Enqueue(T element) { … }  public T Dequeue() { … }  public T[] ToArray() { … }  } |

### 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:



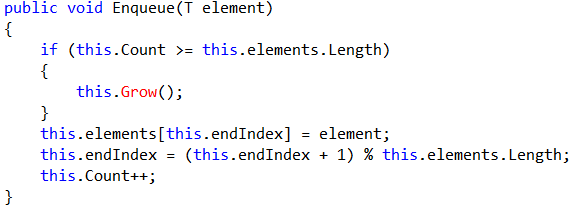
### Направете конструктор

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

### Имплементиране на Enqueue(…) метод

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



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.

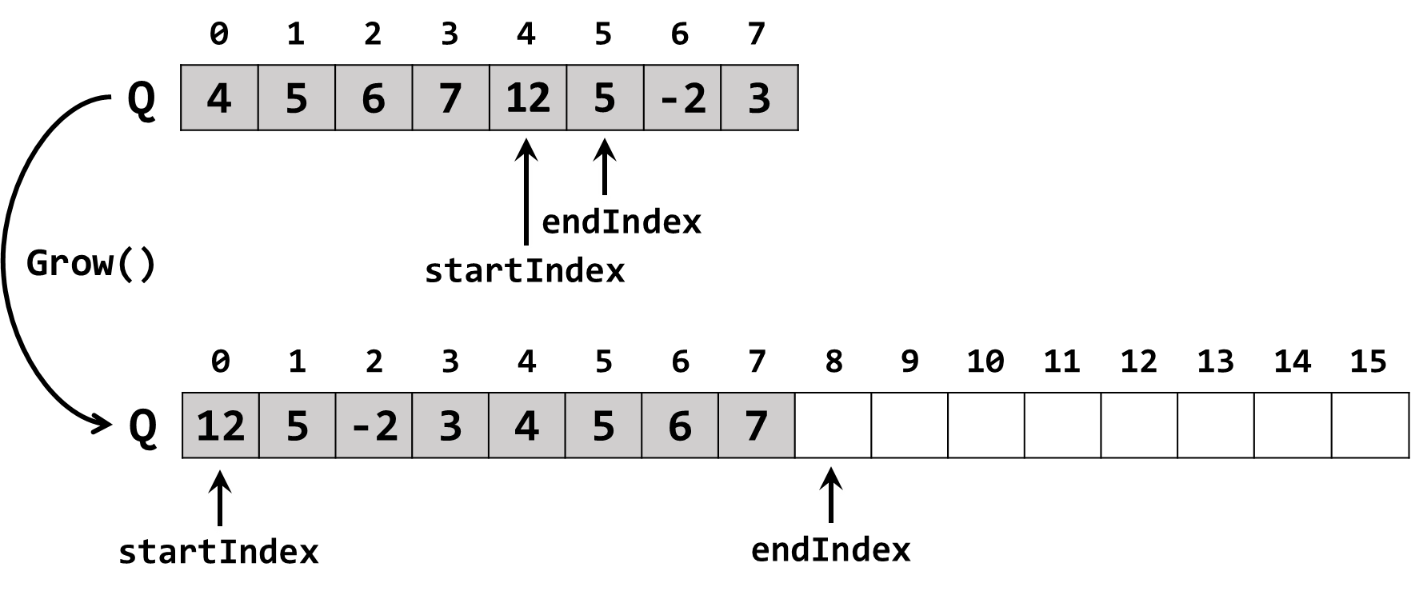
Забележете, че опашката е кръгова, така че елемента след последния елемент (this.elements.Length-1) е 0.

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

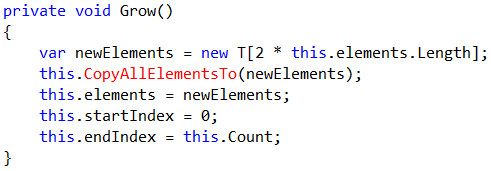
(this.endIndex + 1) % this.elements.Length

### Имплементиране на Grow() метод

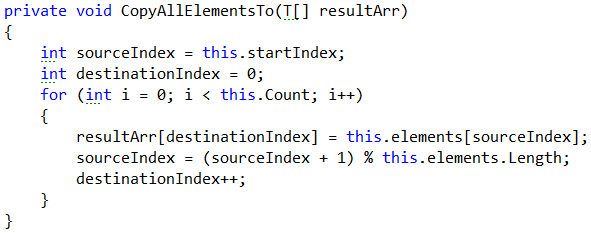
Grow() методът се извиква, когато опашката е със запълнен капацитет (capacity == Count) и искаме да добавим нов елемент. Grow() методът трябва да задели нов масив с **удвоен капацитет** и да премести всички елементи от стария масив в новия масив:



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

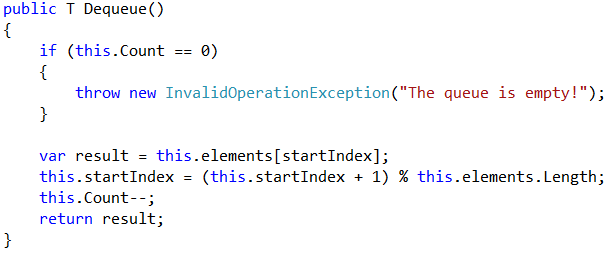


Важна част от "**уголемяването**" е да се **копират елементите от стария масив в новия**. Това може да се случу ето така:



### Имплементиране на Dequeue() метод

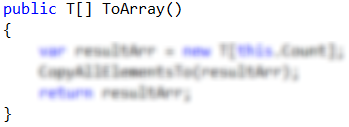
Сега е ред на Dequeue() метода. Неговата цел е да се върне и да се премахне от опашката първият добавен елемент (той се намира на позиция startIndex). Кодът е както следва:



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.

## 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.



## \* 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.