# Homework: Linear Data Structures – Stacks and Queues

This document defines the **homework assignments** for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1147/Data-Structures-June-2015). Please submit a single zip / rar / 7z archive holding the solutions (source code) of all below described problems.

## Reverse Numbers with a Stack

Write a program that reads **N integers** from the console and **reverses them using a stack**. Use the Stack<int> class from .NET Framework. Just put the input numbers in the stack and pop them. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 1 2 3 4 5 | 5 4 3 2 1 |
| 1 | 1 |
| (empty) | (empty) |
| 1 -2 | -2 1 |

## Calculate Sequence with a Queue

We are given the following sequence of numbers:

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

Using the Queue<T> class, write a program to print its first 50 members for given N. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 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, … |

## Implement an Array-Based Stack

Implement the "**stack**" data structure Stack<T> that holds its elements in an array:

|  |
| --- |
| public class ArrayStack<T>  {  private T[] elements;  public int Count { get; private set; }  private const int InitialCapacity = 16;  public ArrayStack(int capacity = InitialCapacity) { … }  public void Push(T element) { … }  public T Pop() { … }  public T[] ToArray() { … }  private void Grow() { … }  } |

Follow the concepts from the CircularQueue<T> class from the exercises in class. The stack is simpler than the circular queue, so you will need to follow the same logic, but more simplified. Some hints:

* The stack **capacity** is this.elements.Length
* Keep the stack **size** (number of elements) in this.Count
* Push(element) just saves the element in elements[this.Count] and increases this.Count
* Push(element) should invoke Grow() in case of this.Count == this.elements.Length
* Pop() decreases this.Count and returns this.elements[this.Count]
* Grow() allocates a new array newElements of size 2 \* this.elements.Length and copies the first this.Count elements from this.elements to newElements. Finally, assign this.elements = newElements
* ToArray() just creates and returns a [**sub-array**](http://stackoverflow.com/questions/943635/c-sharp-arrays-getting-a-sub-array-from-an-existing-array) of this.elements[0…this.Count-1]

## Array-Based Stack: Unit Tests

Write **unit tests** to ensure your array-based stack implementation works correctly. Test the following scenarios:

* **Push / pop element**: create a stack of numbers; assert Count == 0; push element; assert Count == 1; pop element; assert the element is the same like the pushed element; assert Count == 0.
* **Push / pop 1000 elements**: create a stack of strings; assert Count == 0; repeat 1000 times: { push element; assert the Count is correct; }; repeat 1000 times: { pop an element; assert the element is correct; assert the Count is correct }. Pushing 1000 elements will indirectly test the auto-grow functionality several times.
* **Pop from empty stack**: create a stack; pop an element; expect an exception;
* **Push / pop with initial capacity 1**: create a stack of numbers with initial capacity of 1; assert Count == 0; push element; assert Count == 1; push another element; assert Count == 2; pop element; assert the element is correct; assert Count == 1; pop element; assert the element is correct; assert Count == 0.
* **ToArray()**: create a stack of numbers; push a few numbers, e.g. { 3, 5, -2, 7 }; convert the stack to array; assert the results holds the pushed numbers in reversed order, e.g. { 7, -2, 5, 3 }.
* **Empty stack ToArray()**: create a stack of dates; convert the stack to array; expect empty array.

Use as reference the unit tests for the circular queue from the exercises.

## Linked Stack

Implement a stack by a "**linked list**" as underlying data structure:



Use the following code as start:

|  |
| --- |
| public class LinkedStack<T>  {  private Node<T> firstNode;  public int Count { get; private set; }  public void Push(T element) { … }  public T Pop() { … }  public T[] ToArray() { … }  private void Grow() { … }  private class Node<T>  {  private T value;  public Node<T> NextNode { set; set; }  public Node(T value, Node<T> nextNode = null) { … }  }  } |

The Push(element) operation should create a new Node<T> and put it as firstNode, followed by the old value of the firstNode, e.g. this.firstNode = new Node<T>(element, this.firstNode).

The Pop() operation should return the firstNode and replace it with firstNode.NextNode.

## Linked Stack: Unit Tests

Write **unit tests** to ensure your linked stack implementation works correctly. Adjust the array-based stack unit tests.

## Linked Queue

Implement a queue by a "**doubly-linked list**" as underlying data structure:



Use the following code as start:

|  |
| --- |
| 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; }  }  } |

You may modify and adjust the code from the DoublyLinkedList<T> class from few lessons ago.

## Linked Queue: Unit Tests

Write **unit tests** to ensure your linked queue is implemented correctly. Adjust the unit tests from the linked stack.

## \* Sequence N 🡪 M

We are given numbers **n** and **m**, and the following operations:

1. n 🡪 n + 1
2. n 🡪 n + 2
3. n 🡪 n \* 2

Write a program that **finds the shortest sequence of operations** from the list above that **starts from n and finishes in m**. If several shortest sequences exist, find one of them. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 10 | 3 -> 5 -> 10 |
| 5 -5 | (no solution) |
| 10 30 | 10 -> 12 -> 14 -> 28 -> 30 |

**Hint**: use a **queue** and the following algorithm:

1. create a queue of numbers
2. queue 🡨 n
3. while (queue not empty)
   1. queue 🡪 e
   2. if (e < m)
      1. queue 🡨 e + 1
      2. queue 🡨 e + 2
      3. queue 🡨 e \* 2
   3. if (e == m) Print-Solution; exit

The above algorithm either will find a solution, or will find that it does not exist. It cannot print the numbers comprising the sequence n 🡪 m.

To print the sequence of steps to reach **m**, starting from **n**, you will need to keep the previous item as well. Instead using a queue of numbers, use a queue of items. Each item will keep a number and a pointer to the previous item. The algorithms changes like this:

**Algorithm Find-Sequence (n, m)**:

1. create a queue of items { value, previous item }
2. queue 🡨 { n, **null** }
3. while (queue not empty)
   1. queue 🡪 item
   2. if (item.value < m)
      1. queue 🡨 { item.value + 1, item }
      2. queue 🡨 { item.value + 2, item }
      3. queue 🡨 { item.value \* 2, item }
   3. if (item.value == m) Print-Solution; exit

**Algorithm Print-Solution (item)**:

1. while (item not null)
   1. print item.value
   2. item = item.previous