# Exercises: Algorithm Complexity and Linear Data Structures

This document defines the **homework assignments** for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1857/data-structures-january-2018). You can submit your code in the SoftUni Judge System - [https://judge.softuni.bg/Contests/Practice/Index/551](https://judge.softuni.bg/Contests/Practice/Index/551#0).

## Sum and Average

Write a program that reads from the console a sequence of integer numbers (on a single line, separated by a space). Calculate and print the **sum** and **average** of the elements of the sequence. Keep the sequence in List<int>. Round the average to second digit after the decimal separator.

|  |  |
| --- | --- |
| **Input** | **Output** |
| 4 5 6 | Sum=15; Average=5.00 |
| 1 1 | Sum=2; Average=1.00 |
|  | Sum=0; Average=0.00 |
| 10 | Sum=10; Average=10.00 |
| 2 2 1 | Sum=5; Average=1.67 |

## Sort Words

Write a program that reads from the console a **sequence of words** (strings on a single line, separated by a space). **Sort** them alphabetically. Keep the sequence in List<string>.

|  |  |
| --- | --- |
| **Input** | **Output** |
| wow softuni alpha | alpha softuni wow |
| hi | hi |
| rakiya beer wine vodka whiskey | beer rakiya vodka whiskey wine |

## Longest Subsequence

Write a method that finds the **longest subsequence of equal numbers** in given List<int> and returns the result as new List<int>. If several sequences has the same longest length, return the leftmost of them. Write a program to test whether the method works correctly.

|  |  |
| --- | --- |
| **Input** | **Output** |
| 12 2 7 4 **3 3** 8 | 3 3 |
| **2 2 2** 3 3 3 | 2 2 2 |
| 4 4 **5 5 5** | 5 5 5 |
| **1** 2 3 | 1 |
| **0** | 0 |
| 4 2 3 **4 4** | 4 4 |

## Remove Odd Occurrences

Write a program that **removes** from given sequence all numbers that occur **odd number of times**.

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 1 **2 3 4** 1 | 1 1 | 2, 3 and 4 occur odd number of times (once). 1 occurs 2 times |
| **1 2** 3 **4 5** 3 **6** 7 **6** 7 **6** | 3 3 7 7 | 1, 2, 4, 5 and 6 occurs odd number of times 🡪 removed |
| 1 2 1 2 1 2 |  | All numbers occur odd number of times 🡪 removed |
| **3** 7 **3 3** 4 **3** 4 **3** 7 | 7 4 4 7 | 3 occurs odd number of times (5) 🡪 removed |
| 1 1 | 1 1 | All numbers occur even number of times 🡪 sequence stays unchanged |

## Count of Occurrences

Write a program that finds in given array of integers **how many times each of them occurs**. The input sequence holds numbers in range [0…1000]. The output should hold all numbers that occur at least once along with their number of occurrences.

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 4 4 2 3 3 4 3 2 | 2 -> 2 times  3 -> 4 times  4 -> 3 times |
| 1000 | 1000 -> 1 times |
| 0 0 0 | 0 -> 3 times |
| 7 6 5 5 6 | 5 -> 2 times  6 -> 2 times  7 -> 1 times |

## Implement the Data Structure ReversedList<T>

Implement a data structure ReversedList<T> that holds a sequence of elements of generic type T. It should hold a **sequence of items in reversed order**. The structure should have some **capacity** that **grows twice** when it is filled, **always starting at 2**. The reversed list should support the following operations:

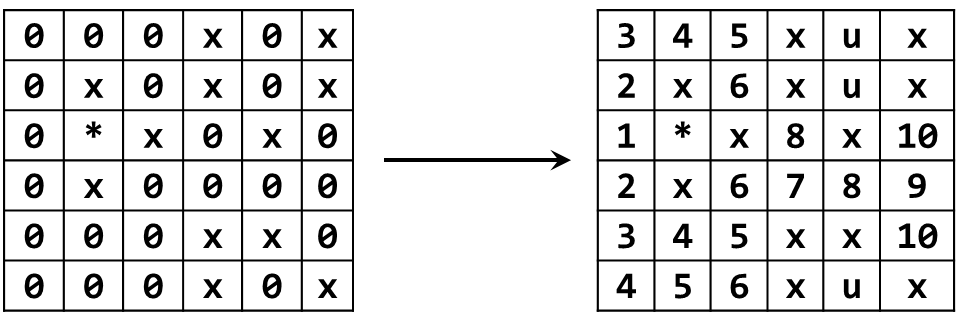
* Add(T item) 🡪 adds an element to the sequence (grow twice the underlying array to extend its capacity in case the capacity is full)
* Count 🡪 returns the number of elements in the structure
* Capacity 🡪 returns the capacity of the underlying array holding the elements of the structure
* this[index] 🡪 the indexer should access the elements by **index** (in range 0 … Count-1) in the reverse order of adding
* RemoveAt(index) 🡪 removes an element by **index** (in range 0 … Count-1) in the reverse order of adding
* IEnumerable<T> 🡪 implement an enumerator to allow iterating over the elements in a foreach loop in a reversed order of their addition

**Note:** For Java, instead of indexer, create methods get(int index) and set(int index, T element). All methods should be with the same names but lowercase.

**Hint:** you can keep the elements in the order of their adding, by access them in reversed order (from end to start).

## \* Distance in Labyrinth

We are given a labyrinth of size N x N. Some of its cells are empty (0) and some are full (x). We can move from an empty cell to another empty cell if they share common wall. Given a starting position (\*) calculate and fill in the array the minimal distance from this position to any other cell in the array. Use "u" for all unreachable cells.



|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  x0  \*x | xu  \*x |
| 3  000  0\*0  000 | 212  1\*1  212 |
| 6  000x0x  0x0x0x  0\*x0x0  0x0000  000xx0  000x0x | 345xux  2x6xux  1\*x8x10  2x6789  345xx10  456xux |

## \*\* Implement a DoublyLinkedList<T>

You are given a project skeleton that contains unit tests for a DoublyLinkedList<T> data structure.

You have to implement a **doubly linked list** in C# or Java – a data structure that holds **nodes**, where each node knows its **next** and **previous** nodes:



Before starting, get familiar with the concept of doubly linked list: <https://en.wikipedia.org/wiki/Doubly_linked_list>.

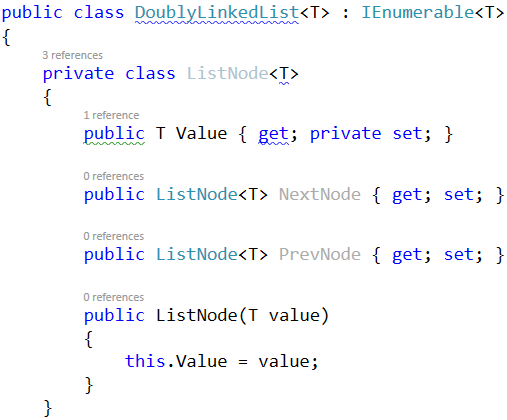
The typical operations over a doubly linked list are **add** / **remove** element at **both ends** and **traverse**. By definition, the doubly linked list has a head (list start) and a tail (list end). Let's start coding!

## Implement ListNode<T>

The first step when implementing a linked / doubly linked list is to understand that we need **two classes**:

* ListNode<T> class to hold a single list node (its value + next node + previous node)
* DoublyLinkedList<T> to hold the entire list (its head + tail + operations)

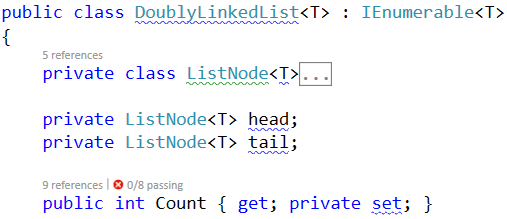
Now, let's write the **list node class**. It should hold a Value and a reference to its previous and next node. It can be inner class, because we will need it only internally from the doubly linked list class:



The class ListNode<T> is called **recursive data structure**, because it references itself recursively. It uses the **generic argument** T to avoid later specialization for any data type, e.g. int, string or DateTime. The **generic classes in C#** work similarly to **templates in C++** and **generic types in Java**.

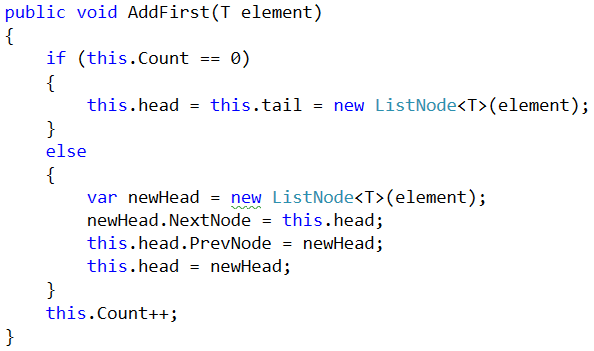
## Implement Head, Tail and Count

Now, let's define the head and tail of the doubly linked list:



## Implement AddFirst(T) Method

Next, implement the AddFirst(T element) method:



Adding an element at the start of the list (before its head) has **two scenarios** (considered in the above code):

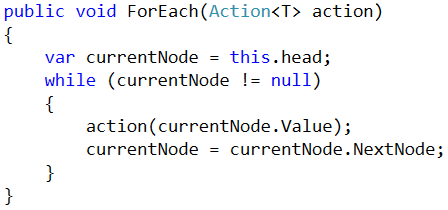
* **Empty list** 🡪 add the new element as head and tail in the same time.
* **Non-empty list** 🡪 add the new element as **new head** and redirect the **old head** as second element, just after the new head.



The above graphic visualizes the process of inserting a new node at the start (head) of the list. The **red** arrows denote the removed pointers from the old head. The **green** arrows denote the new pointers to the new head.

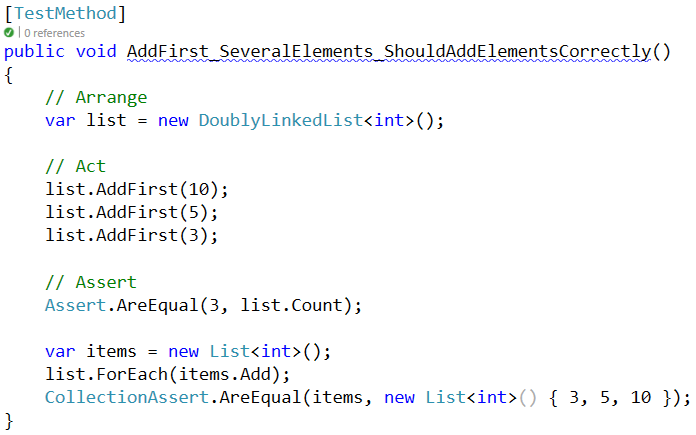
## Implement ForEach(Action) Method

We have a doubly linked list. We can add elements to it. But we cannot see what's inside, because the list still does not have a method to traverse its elements (pass through each of them, one by one). Now let's define the ForEach(Action<T>) method. In programming such a method is known as ["**visitor**" pattern](https://en.wikipedia.org/wiki/Visitor_pattern). It takes as an argument a function (action) to be invoked for each of the elements of the list. The algorithm behind this method is simple: start from head and pass to the next element until the last element is reached (its next element is null). A sample implementation is given below:

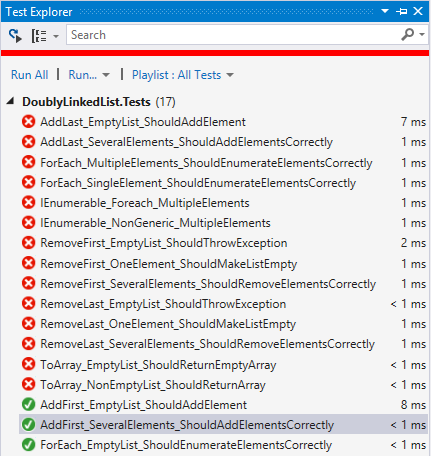


## Run the Unit Tests

Now we have the methods AddFirst(T) and ForEach(Action<T>). We are ready to run the unit tests to ensure they are correctly implemented. Most of the **unit tests** create a doubly linked list, add / remove elements from it and then check whether the elements in the list are as expected. For example, let's examine this unit test:

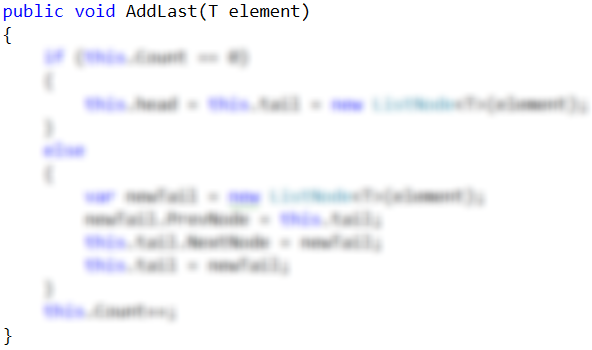


If we **run the unit tests**, some of them will now pass:

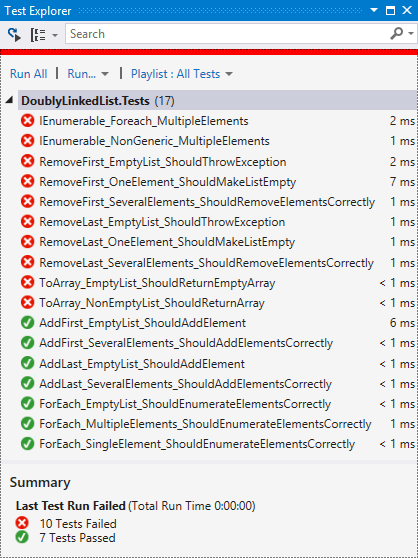


## Implement AddLast(T) Method

Next, implement the AddLast(T element) method for appending a new element as the list tail. It should be very similar to the AddFirst(T element) method. The logic inside it exactly the same, but we append the new element at the tail instead of at the head. The code below is intentionally blurred. Write it yourself!



Now **run the unit tests** again. You should have several more passed (green) tests:

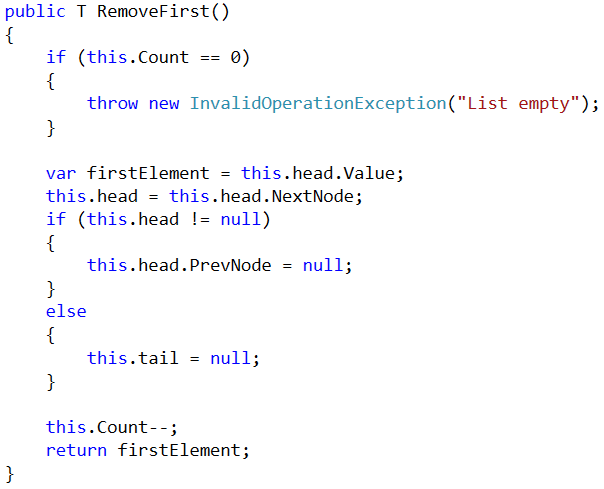


## Implement RemoveFirst() Method

Next, let's implement the method RemoveFirst() 🡪 T. It should **remove the first element** from the list and move its head to point to the second element. The removed element should be returned as a result from the method. In case of empty list, the method should throw an exception. We have to consider the following three cases:

* **Empty list** 🡪 throw and exception.
* **Single element in the list** 🡪 make the list empty (head == tail == null).
* **Multiple elements in the list** 🡪 remove the first element and redirect the head to point to the second element (head = head.NextNode).

A sample implementation of RemoveFirst() method is given below:



Run the **unit tests** to ensure the method is correctly implemented:

## Implement RemoveLast() Method

Next, let's implement the method RemoveLast() 🡪 T. It should **remove the last element** from the list and move its tail to point to the element before the last. It is very similar to the method RemoveFirst(), so you are free to implement it yourself. The code below is intentionally blurred:

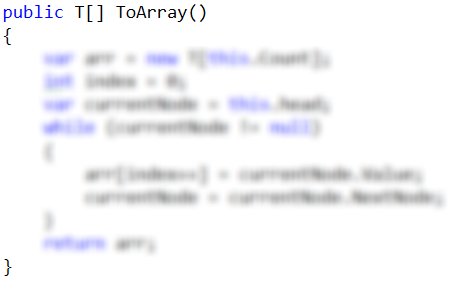


## Implement ToArray() Method

Now, implement the next method: ToArray() 🡪 T[]. It should copy all elements of the linked list to an array of the same size. You could use the following steps to implement this method:

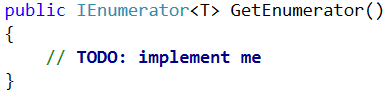
* Allocate an array T[] of size this.Count.
* Pass through all elements of the list (from head to tail) and fill them to T[0], T[1], …, T[Count-1].
* Return the array as result.

Write yourself the blurred code in the method ToArray():



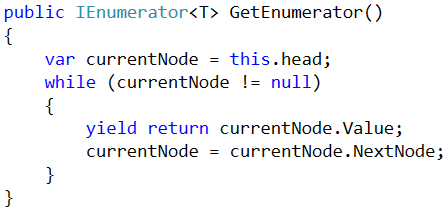
## Implement IEnumerable<T>

Collection classes in C# and .NET Framework (like arrays, lists and sets) implement the system interface IEnumerable<T> to enable the foreach iteration over their elements. The C# keyword foreach calls internally the following method:



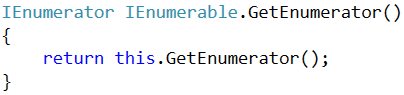
This method returns IEnumerator<T>, which can move to the next element and read the current element. In programming, this is known [as "**iterator**" pattern](https://en.wikipedia.org/wiki/Iterator_pattern) (**enumerator**).

We will use [the "**yield return**" C# statement](https://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx) to simplify the implementation of the iterator:

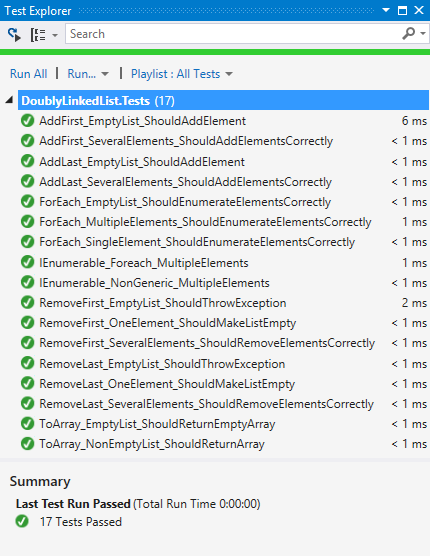


The above code will enable using the DoublyLinkedList<T> in foreach loops.

The last unimplemented method is the **non-generic enumerator**:



Finally, **run the unit tests** to ensure all of them pass correctly:



Congratulations! You have implemented your doubly linked list.

# Exercises: Linear Data Structures – Stacks and Queues

This document defines the **exercise assignments** for the ["Data Structures" course @ Software University](https://softuni.bg/trainings/1857/data-structures-january-2018). You can submit your code in the SoftUni Judge System - <https://judge.softuni.bg/Contests/567/Linear-Data-Structures-Stacks-and-Queues-CSharp-Exercise>.

## 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]
* Pop() should throw InvalidOperationException (or IllegalArgumentException) if the stack is empty

## 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 class Node<T>  {  private T value;  public Node<T> NextNode { get; 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. If the stack is empty, it should throw InvalidOperationException.

## 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 the last lesson. If the queue is empty, the Dequeue() should throw InvalidOperationException.

## \* 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 the first one of them. Examples:

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3 10 | 3 -> 5 -> 10 |
| 5 -5 | (no solution) |
| 10 30 | 10 -> 11 -> 13 -> 15 -> 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