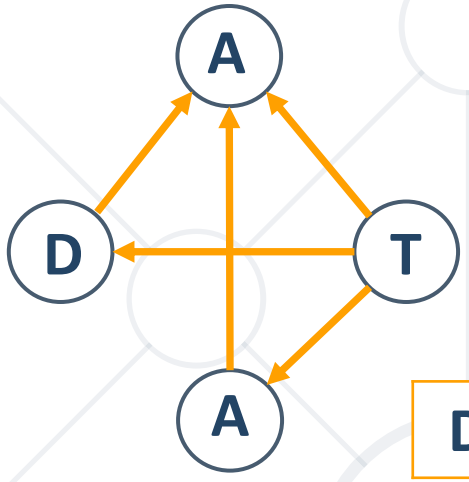
Linear Data Structures

Static and Dynamic Implementation



SoftUni Team

Technical Trainers

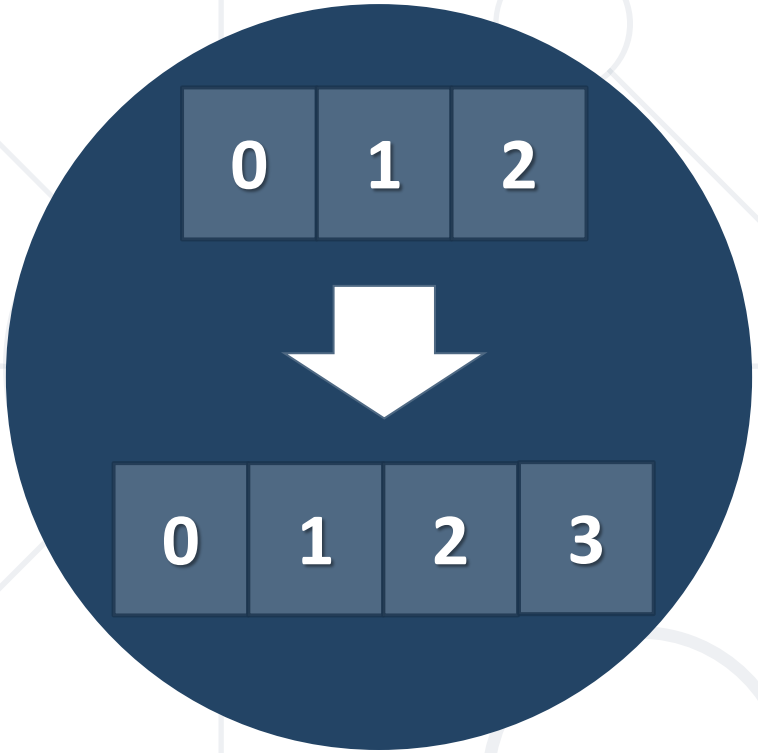
Software University

SoftUni

Software University

https://softuni.bg

Table of Contents



Dynamic Arrays

Lists

Dynamic Arrays – List

- List is the **implementation** of ADS **List**
  - Built **atop an array**, which is able to dynamically **grow** and **shrink** as you **add/remove** elements
- Stores the **elements** inside an array

```
public class List<T> : IAbstractList<T>
{
    private T[] items;
}
```

List – Operations

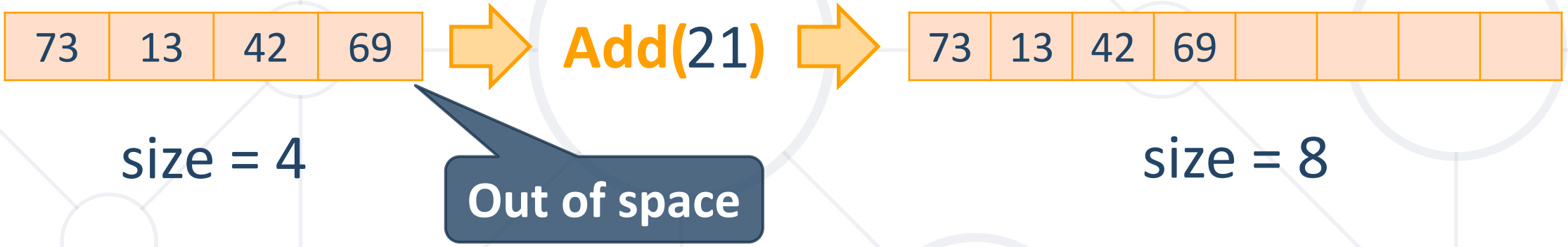
- Supported operations and complexity:
  - **Count**, **Get** and **Set** through **indexer** - **O(1)**
  - **Add(T item)**
    - The operation runs in **amortized constant** time
    - Adding **n** elements requires **O(n)** time

List – Operations (2)

- All of the other operations like:
  - **Insert**(int index, T item)
  - **Contains**(T item)
  - **IndexOf**(T item)
  - **Remove**(T item)
  - **RemoveAt**(int index)
- Run in **linear time O(n)**

List – Add O(1)

- When **adding**, if needed **double** the size



- This approach will copy at **log(n)** → n = 10^9, only ~33 copies – **O(1) amortized**

Problem: List

- Create a `List<T>` data structure
  - `void Add(T element)`
  - `Get` and `Set` operations through an indexer
  - `int Count { get; }`
  - `bool Remove(T item)`
  - `void RemoveAt(int index)`
  - `int IndexOf(T item)`

List – Constructor and Fields

- Constructor and fields:

```
public class List<T> : IList<T>
{
    private const int DEFAULT_CAPACITY = 4;
    private T[] elements;
    private int size;

    public List()
    {
        this.elements = new T[DEFAULT_CAPACITY];
    }
}
```

List – Add

- Adds an element after the last element:

```
public void Add(T item)
{
    if(this.size == this.elements.Length)
    {
        this.elements = this.Grow();
    }
    this.elements[this.size] = item;
    this.size++;
}
```

Indexer

```
public T this[int index]
{
    get
    {
        this.ValidateIndex(index);
        return this.elements[index];
    }
    set
    {
        this.ValidateIndex(index);
        this.elements[index] = value;
    }
}
```

List – RemoveAt

- Removes an element at the specified:

```
public void RemoveAt(int index)
{
    this.ValidateIndex(index);
    this.Shift(index);
    this.size--;
}
```

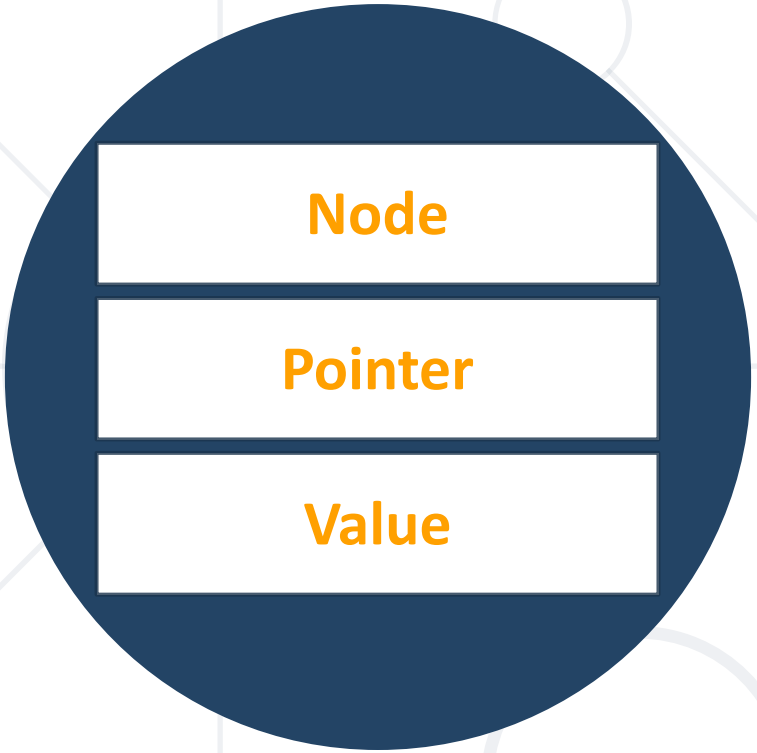
Helper Methods – Grow and Shrink

```
private T[] Grow()
{
    T[] newArray = new T[this.elements.Length * 2];
    Array.Copy(this.elements, newArray, this.elements.length);
    return newArray;
}

private T[] Shrink()
{
    T[] newArray = new T[this.elements.Length / 2];
    // To Do: Implement this on your own
}
```

List – Other Operations

- **IndexOf**(T item)
  - Returns the **zero** based index of an element or **-1**
- **Contains**(T item)
  - Returns **whether** an element is present
- **Count**
  - Returns the **number** of elements
- **ToArray**()
  - Returns the elements **as an array**



Nodes

Building Block

Node Class

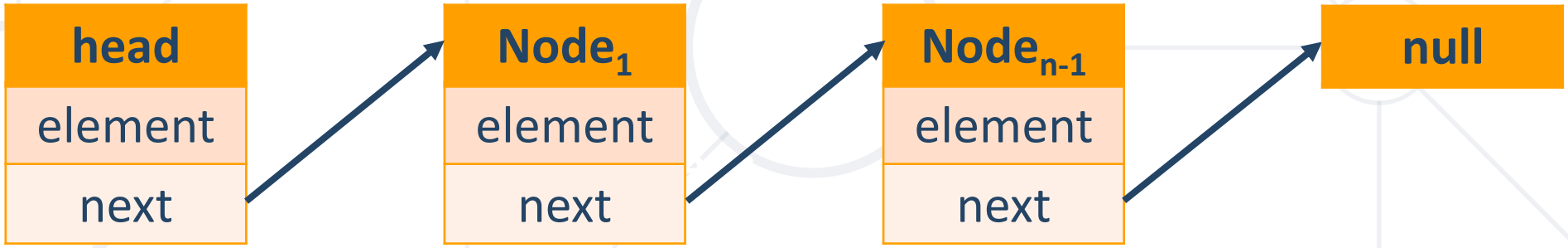
- The **Node** class is the **build block** for many data structures
- Inside Node object we store **an element and pointer to the next node at least**

```
public class Node<T>
{
    public T Element { get; set; }
    public Node<T> Next { get; set; }
}
```

Node – Application

- Many data structures use **node chaining**

```
public class LinkedList<T> : IAbstractList<T>
{
    private Node<E> head;
}
```



Problem: Node

- Create a class **Node<T>**, that has:
  - **T** Element
  - **Node**<T> Next
  - **Constructor**

```
public class Node<T> {
    public T Element { get; set; }
    public Node<T> Next { get; set; }

    public Node(T value) {
        this.Element = value;
    }
}
```

Stacks

Dynamic Implementation

Stack

- Stack is the **implementation** of ADS **LIFO**
  - **L**ast **I**n **F**irst **O**ut
  - Build by using **Node** class or atop an **array**
- Stack example using Node

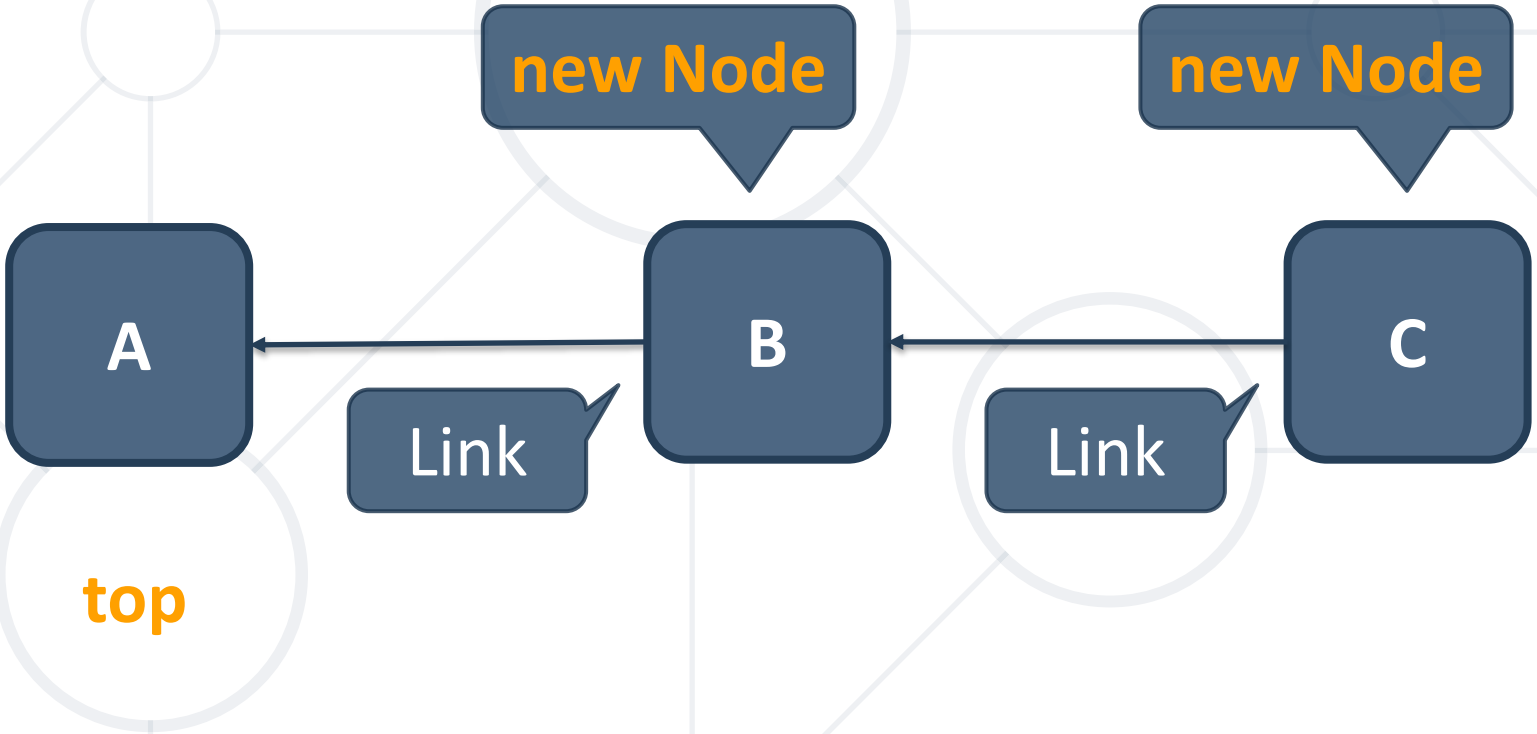
```
public class Stack<T> : IAbstractStack<T>
{
    private Node<T> top;
    private int size;
}
```

Stack – Operations

- Supported operations and complexity:
  - **Count**, **Push**(T item), **Pop**(), **Peek**() – **O(1)**
  - All other operations run in linear time – **O(n)**
    - **CopyTo**(T[] array, int arrayIndex)
    - **Contains**(T item)
    - etc…

Stack – Push

- Chain the nodes by using the **top** field:



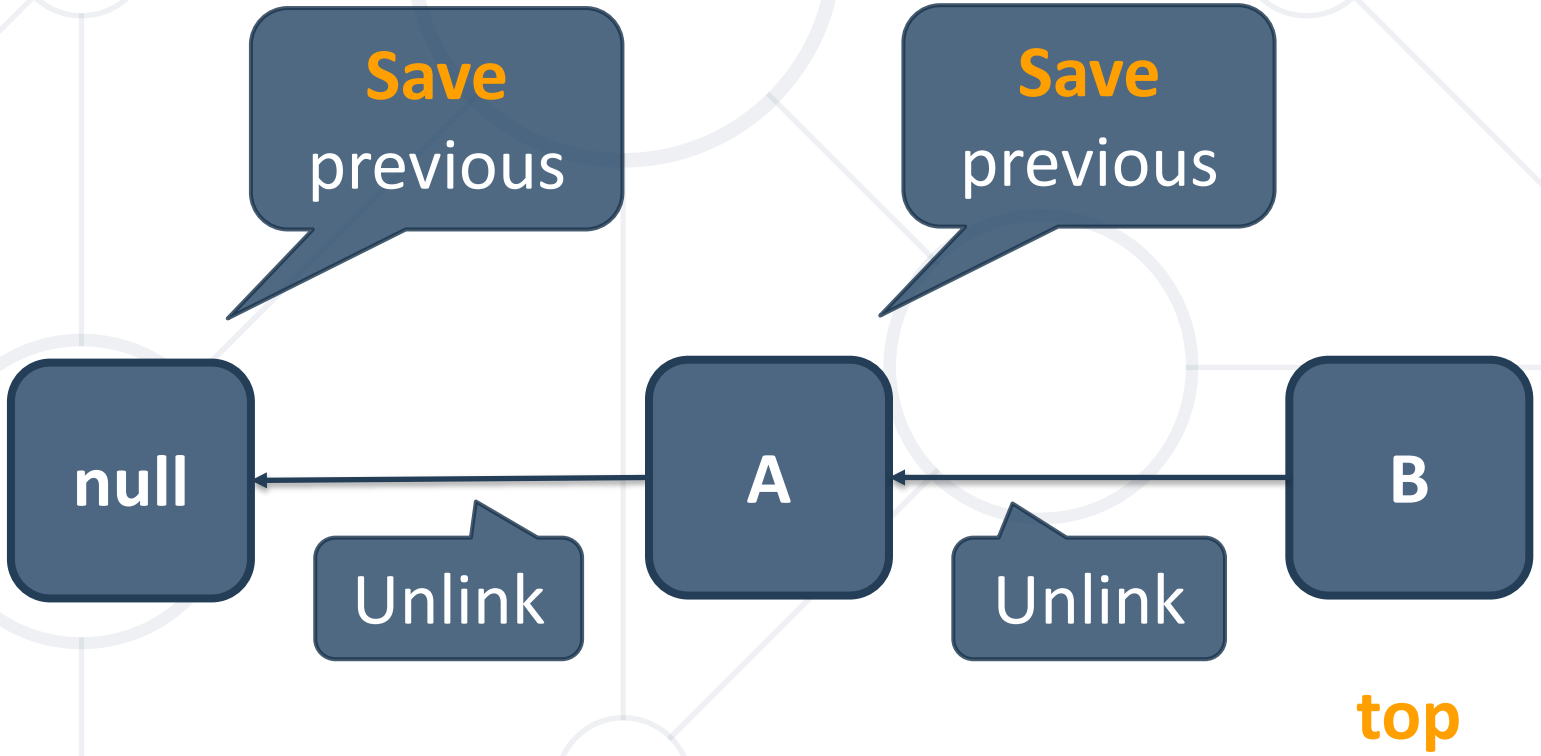
Stack – Push

- Add element at the top
  - **Link** the nodes and **increment** size

```
public void Push(T element)
{
    var newNode = new Node<T>(element);
    newNode.Next = top;
    top = newNode;
    this.size++;
}
```

Stack – Pop

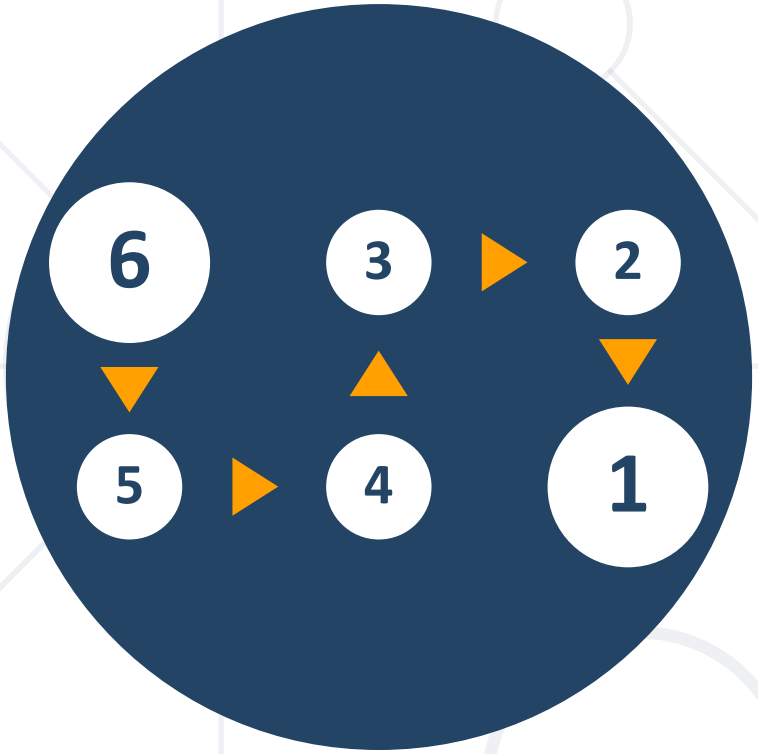
- Remove the **top** Node and return the element
  - **Unlink** the nodes and **decrease** size



Stack – Pop

- Remove and return element at the top:

```
public T Pop()
{
    // To Do: Implement on your own
}
```



Queues

Dynamic Implementation

Queue

- Queue is the **implementation** of ADS **FIFO**
  - **F**irst **I**n **F**irst **O**ut
  - Build by using **Node** class or atop an **array**
- Queue example using Node

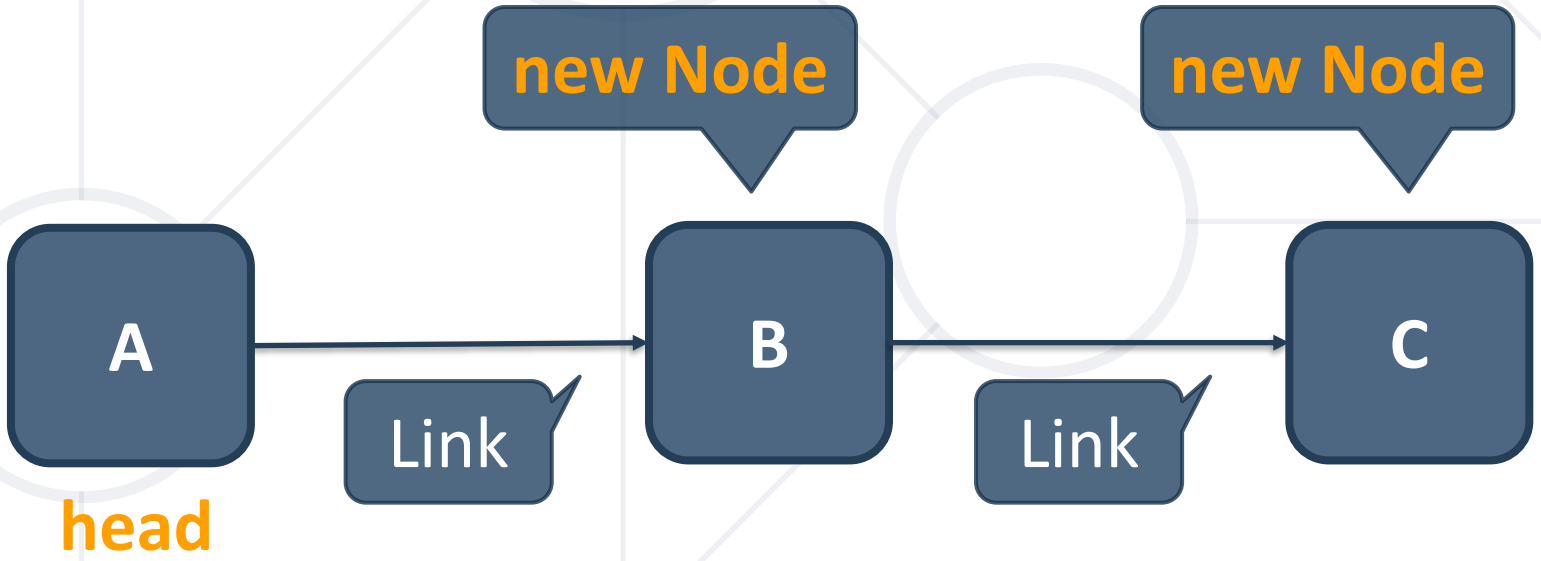
```
public class Queue<T> : IAbstractQueue<T>
{
    private Node<T> head;
    private int size;
}
```

Queue – Operations

- Supported operations and complexity:
  - **Count**, **Dequeue**(), **Peek**() – **O(1)**
  - **Enqueue**(T item):
    - If we keep the reference to the that node – **O(1)**
    - If we have to chase pointers to that node – **O(n)**

Queue – Enqueue

- Head == null => head = new Node
- Size > 0 => chain the nodes by adding new Node after the last one - the so-called tail:



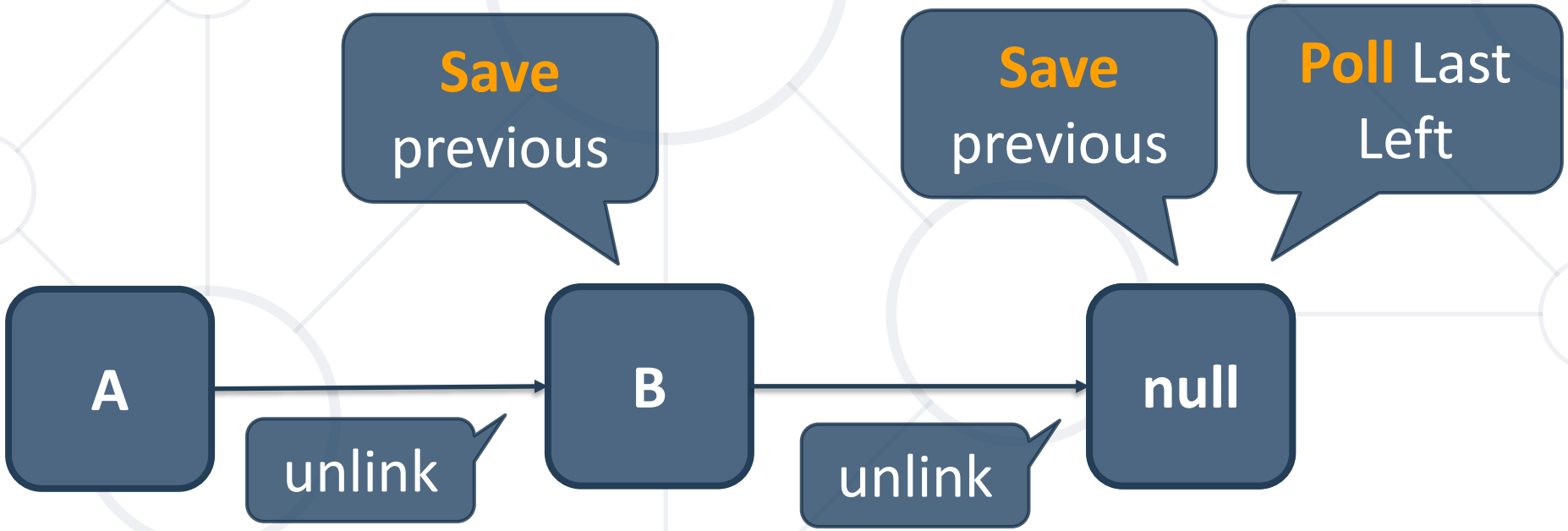
Queue – Enqueue

- Add element at the end – **link** the nodes and **increase** size

```
public void Enqueue(T element)
{
    // To Do: Implement on your own
}
```

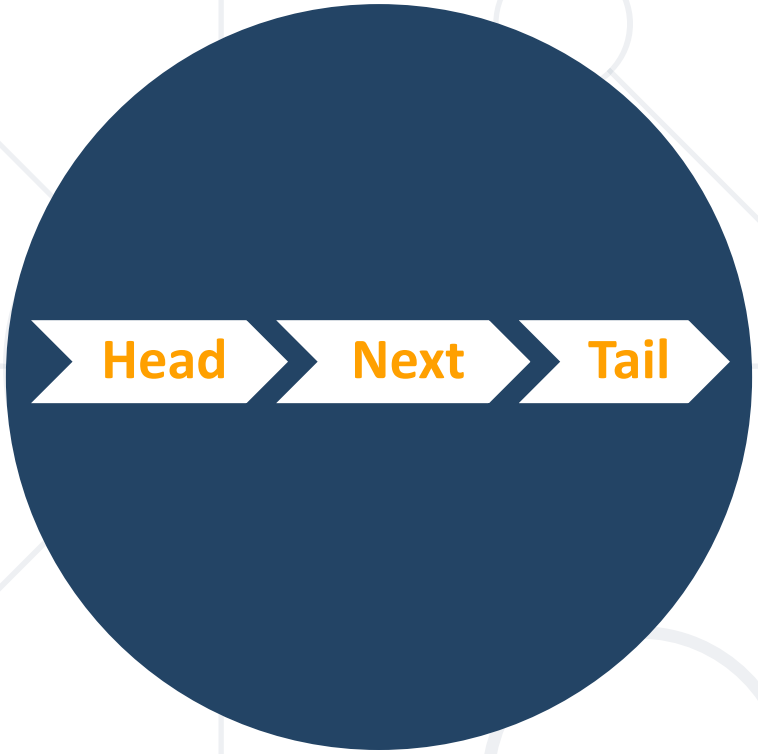
Queue – Dequeue

- Remove the **head** Node and return the element
- **Unlink** the node and **decrease** size



Stack / Queue – Real-World Applications

- Stack
  - **Undo** operations
    - Browser history
    - Chess game progress
  - **Math expression** evaluation
  - Implementation of **function** (method) **calls**
  - Tree-like structures traversal (**DFS** algorithm)
- Queue
  - Operation system **process scheduling**
  - Resource sharing
    - Printer document queue
    - **Server requests** queue
  - Tree-like structures traversal (**BFS** algorithm)



Linked Lists

SinglyLinkedList

SinglyLinkedLists

- Linear data structure where each **element** is a **separate object** – **Node**
- The elements are **not** stored at **contiguous** memory
- The entry point is commonly the **head** of the list

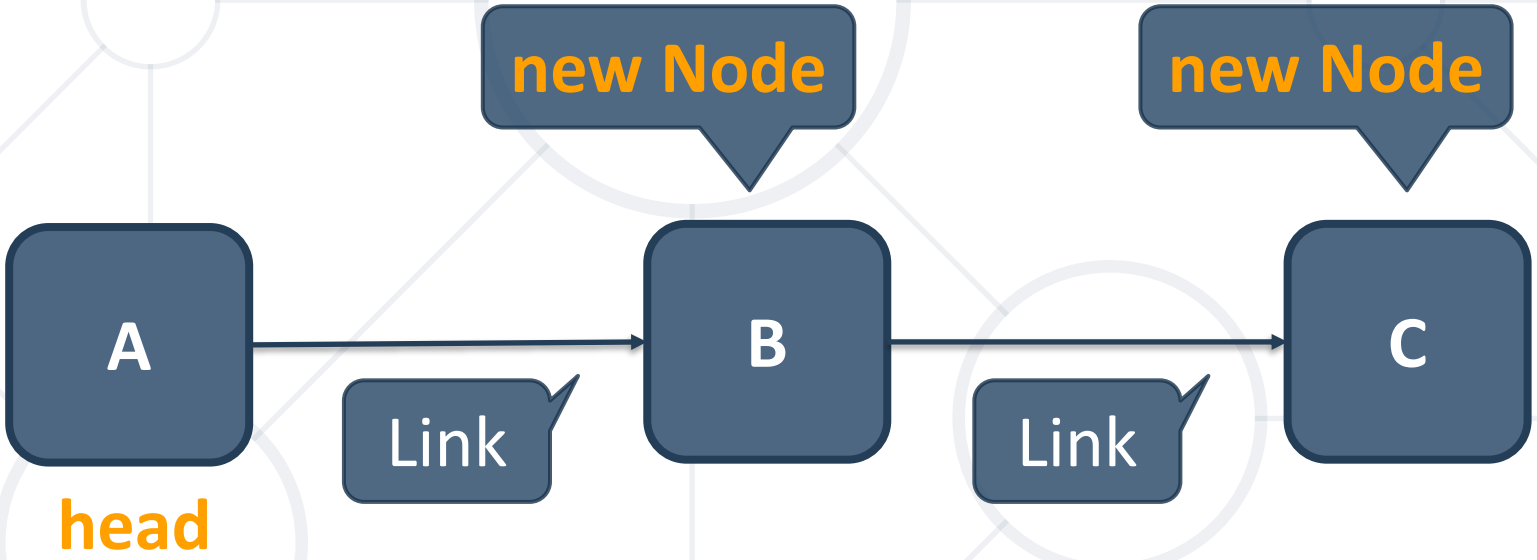
```
public class SinglyLinkedList<T> : IAbstractLinkedList<T>
{
    private Node<T> head;
    private int size;
}
```

Singly Linked List – Operations

- Supported operations and complexity:
  - **AddFirst**(T item), **RemoveFirst**(), **GetFirst**(), **Count** – **O(1)**
  - How about operations on the **last element**?
    - **AddLast**(), **RemoveLast**(), **GetLast**()
      - Depends if we keep the **reference** to the **last node (DoublyLinkedList)** or not can be constant – **O(1)** or linear – **O(n)**
  - Operations that **index** into the list will run in **linear time O(n)**

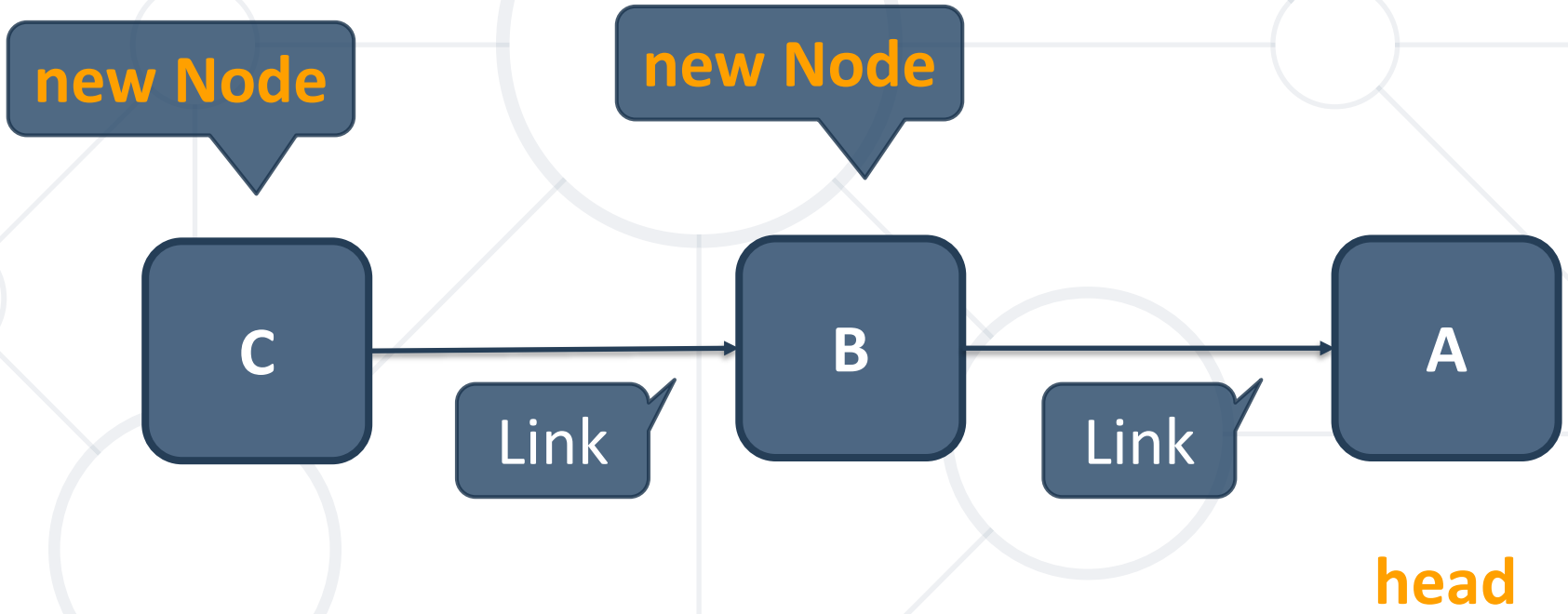
Singly Linked List – Adding Last

- Same as **Queue** – Enqueue



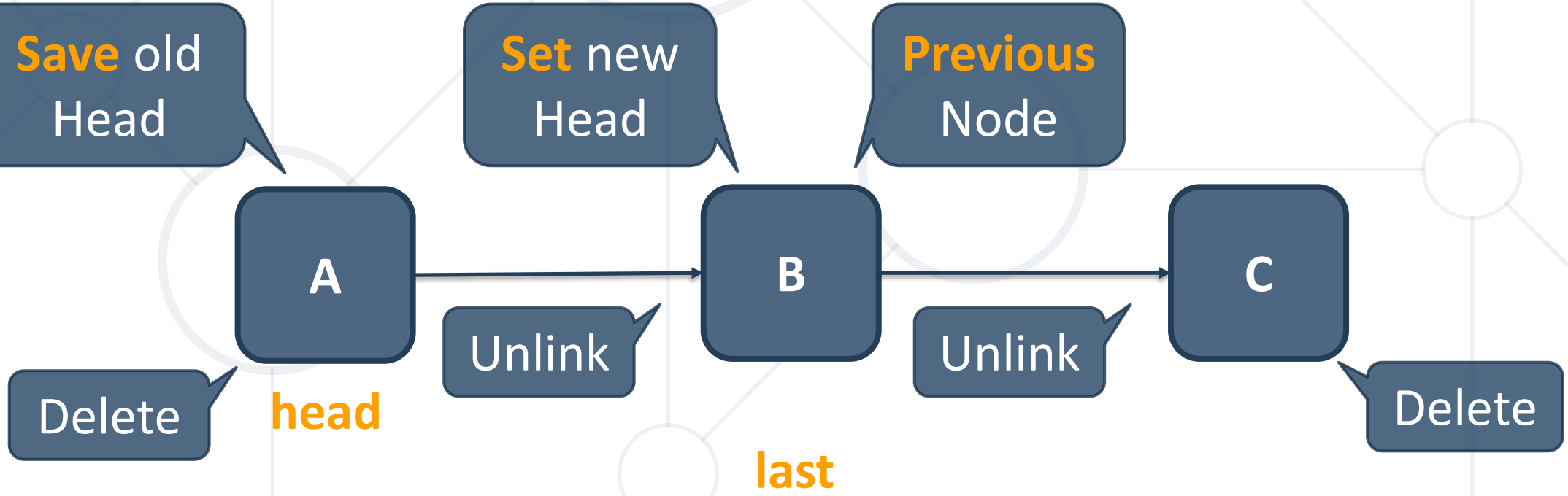
Singly Linked List – Adding First

- Same as **Stack** – Push



Linked List – Removing First/Last

- Size == 0 → Do Nothing / Throw Exception
- Size == 1 → head = null
- Size > 1



Node Implementation

- We have implemented some Data Structures
  - **Node** class properties
- However the way we did it **introduces** some **performance problems** when **chaining** nodes.
- **Can** we **solve** them?
- Add/Remove/Get in **constant time**?
- We will try to **understand** and **solve** those **problems** at the exercise.

Summary

- Stack is **LIFO** structure (**L**ast **I**n **F**irst **O**ut)
  - Linked implementation is pointer-based
- Queue is **FIFO** (**F**irst **I**n **F**irst **O**ut) structure
  - Linked implementation is pointer-based
- SinglyLinkedList
  - Linked implementation is pointer-based

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter™ International

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank
Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg