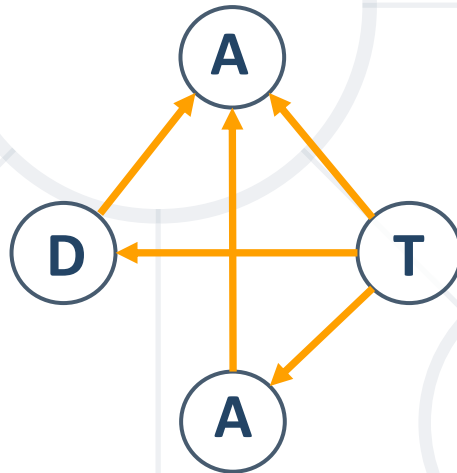


Linear Data Structures

Static and Dynamic Implementation

D	A	T	A
---	---	---	---



SoftUni Team
Technical Trainers



SoftUni



Software University

<https://softuni.bg>

1. Dynamic Arrays

- ArrayList – Static Implementation

2. Nodes

3. Stacks

- Linked/Dynamic Implementation

4. Queues

- Linked/Dynamic Implementation

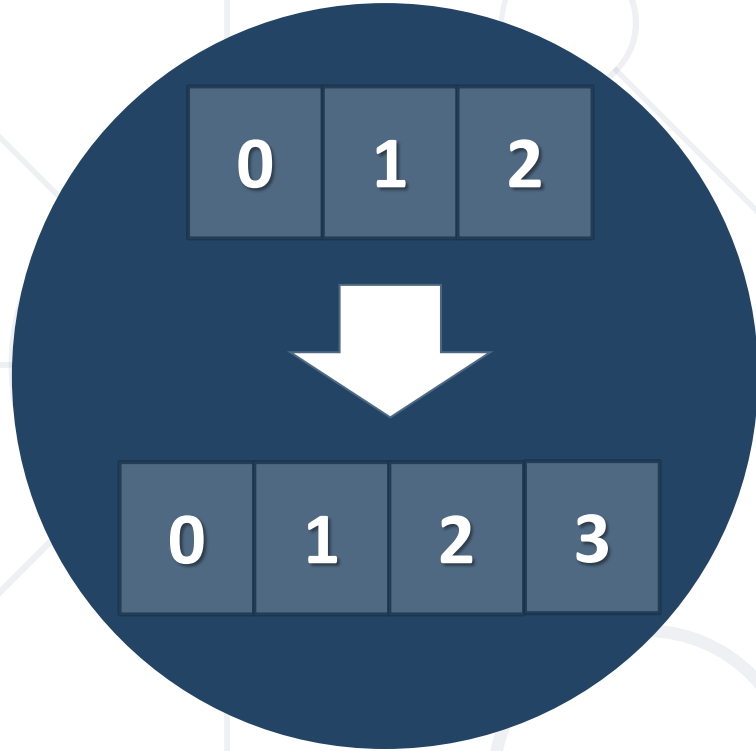
5. Linked Lists

- SinglyLinkedList



sli.do


#ds-java



Dynamic Arrays

ArrayList

Dynamic Arrays – ArrayList

- 
- ArrayList 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 ArrayList<E> implements List<E> {  
    private Object[] elements;  
}
```

- Supported operations and complexity:
 - **size(), isEmpty(), get(), set()** – **$O(1)$**
 - **add()** – the operation runs in **amortized constant** time
 - adding **n** elements requires **$O(n)$** time
 - all of the other operations like: **add(int index, E element), contains(), indexOf(), remove(int index)** etc., run in **linear time $O(n)$** (roughly speaking)

ArrayList – Add $O(n)$

- Implemented **using an array**
- Adding **new item** requires **new array**



- This approach will copy **all the elements** for each add operation – **$O(n)$**

ArrayList – Add $O(1)$

- Implemented **using an array**
- When **adding**, if needed **double** the size



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

Problem: ArrayList

- Create an **ArrayList<E>** data structure, which supports
 - boolean **add**(T element)
 - E **get**(int index)
 - E **set**(int index, E elements)
 - E **remove**(int index)
 - int **size**()
 - int **indexOf**(E element)
 - etc...

ArrayList – Constructor And Fields

- Constructor and fields:

```
public class ArrayList<E> implements List<E> {  
    private static final int DEFAULT_CAPACITY = 4;  
    private Object[] elements;  
    private int size;  
  
    public ArrayList() {  
        this.elements = new Object[DEFAULT_CAPACITY];  
    }  
}
```

- Adds an element after the last element:

```
public boolean add(E element) {  
    if(this.size == this.elements.length) {  
        this.elements = grow();  
    }  
    this.elements[this.size++] = element;  
    return true;  
}
```

- Returns an element at index:

```
public E get(int index) {  
    checkIndex(index);  
    return this.getElement(index);  
}  
  
private E getElement(int index) {  
    return (E) this.elements[index];  
}
```

- Sets an element at index:

```
public E set(int index, E element) {  
    checkIndex(index);  
    E oldElement = this.getElement(index);  
    this.elements[index] = element;  
    return oldElement;  
}
```

- Removes and returns an element at index:

```
public E remove(int index) {  
    this.checkIndex(index);  
    E element = this.getElement(index);  
    this.elements[index] = null;  
    this.size--;  
    shift(index);  
    ensureCapacity();  
    return element;  
}
```

ArrayList – Grow and Shrink

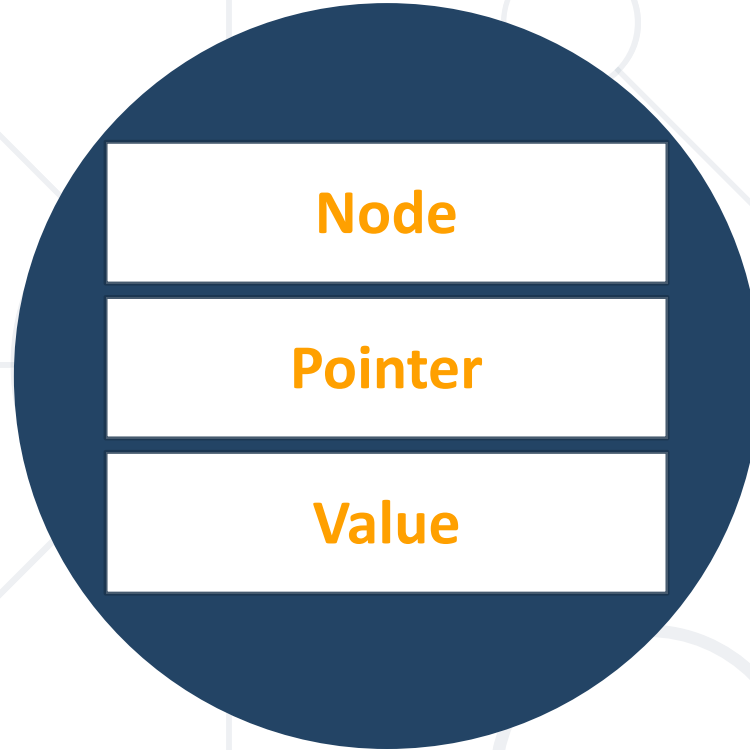
- Helper methods to sustain correct array behaviour:

```
private Object[] grow() {  
    return Arrays.copyOf(this.elements, this.elements.length * 2);  
}  
  
private Object[] shrink() {  
    return Arrays.copyOf(this.elements, this.elements.length / 2);  
}
```

ArrayList – Other Operations

- **indexOf**(E element) – returns the **zero** based index of an element or **-1**
- **contains**(E element) – returns **whether** an element is present
- **size**() – returns the **number** of elements
- **toArray**() – returns the elements **as an array**
- Implement other operations – think about the complexity





Nodes

Building Block

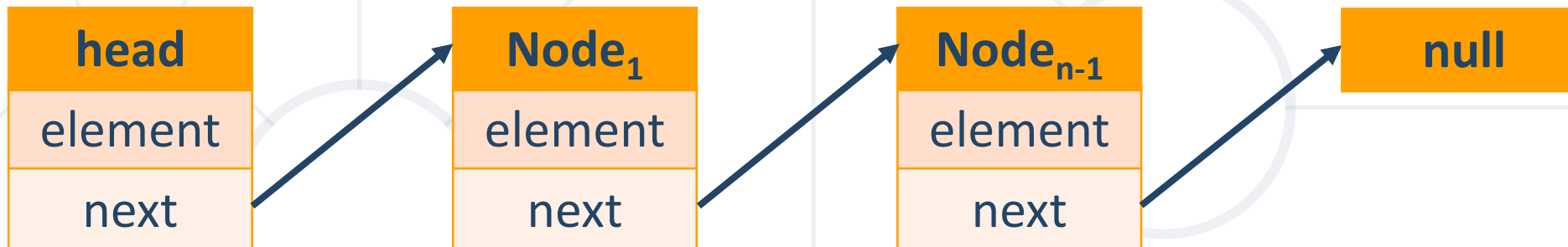
Node Class

- 
- 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**
 - However, we **can store anything else**

```
private static class Node<E> {  
    private E element;           // Must have  
    private Node<E> next;       // Must have  
    private Node<E> previous;   // Additional  
}
```

- Many data structures use **node chaining**

```
public class LinkedList<E> implements Deque<E> {  
    private Node<E> head;  
}
```



Problem: Node

- Create a class **Node<E>**, that has:
 - **E** element
 - **Node<E>** next
 - **Constructor**

```
private static class Node<E> {  
    private E element;  
    private Node<E> next;  
  
    public Node(E value) {  
        this.element = value;  
    }  
}
```



Stacks
Dynamic Implementation

Stack

- Stack is the **implementation** of ADS **LIFO**
Last In First Out
 - Build by using **Node** class or atop an **array**
- Stack example using Node

```
public class Stack<E> implements AbstractStack<E> {  
    private Node<E> top;  
    private int size;  
}
```

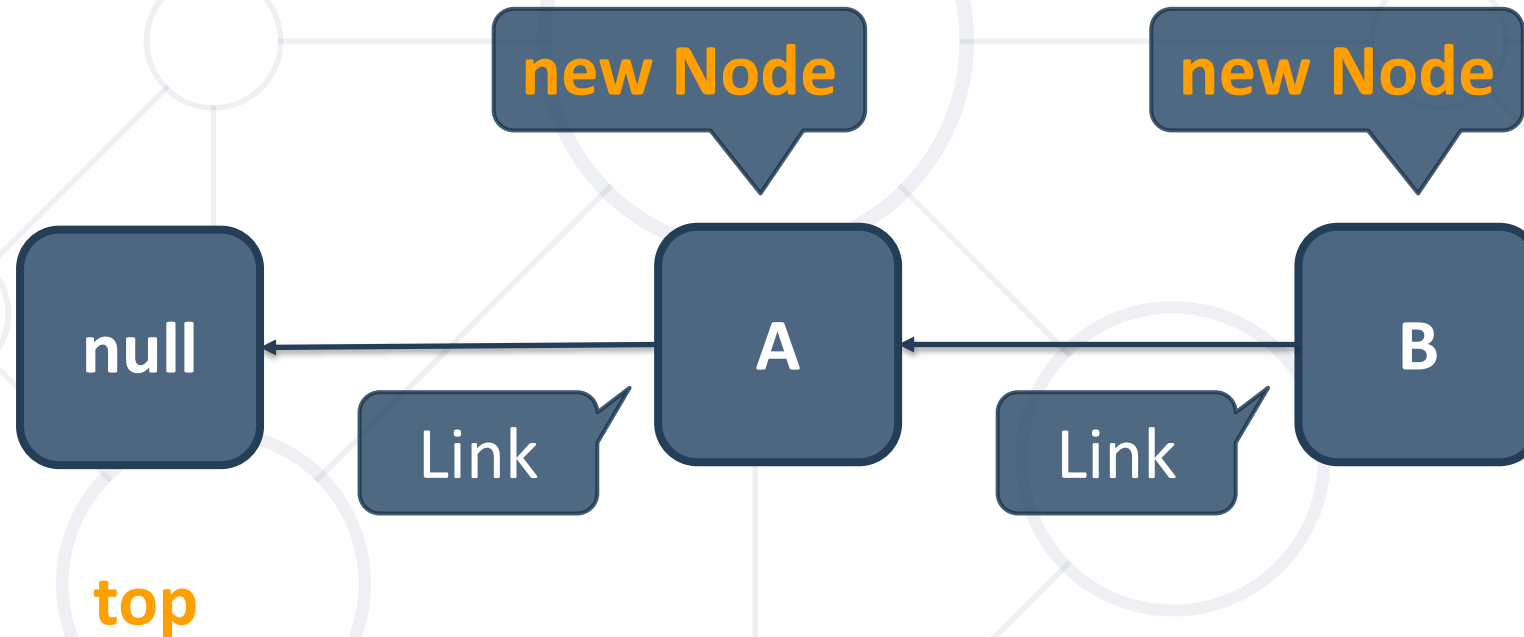


- Supported operations and complexity:
 - **size(), isEmpty(), push(), pop(), peek() – $O(1)$**
 - all of the other operations run in linear time (roughly speaking):
 - **forEach()**
 - **contains()**
 - etc...

- Constructor and fields:

```
public class Stack<E> implements AbstractStack<E> {  
    private Node<E> top;  
    private int size;  
  
    private static class Node<E> {...}    // Node class  
  
    public Stack() {  
    }  
}
```

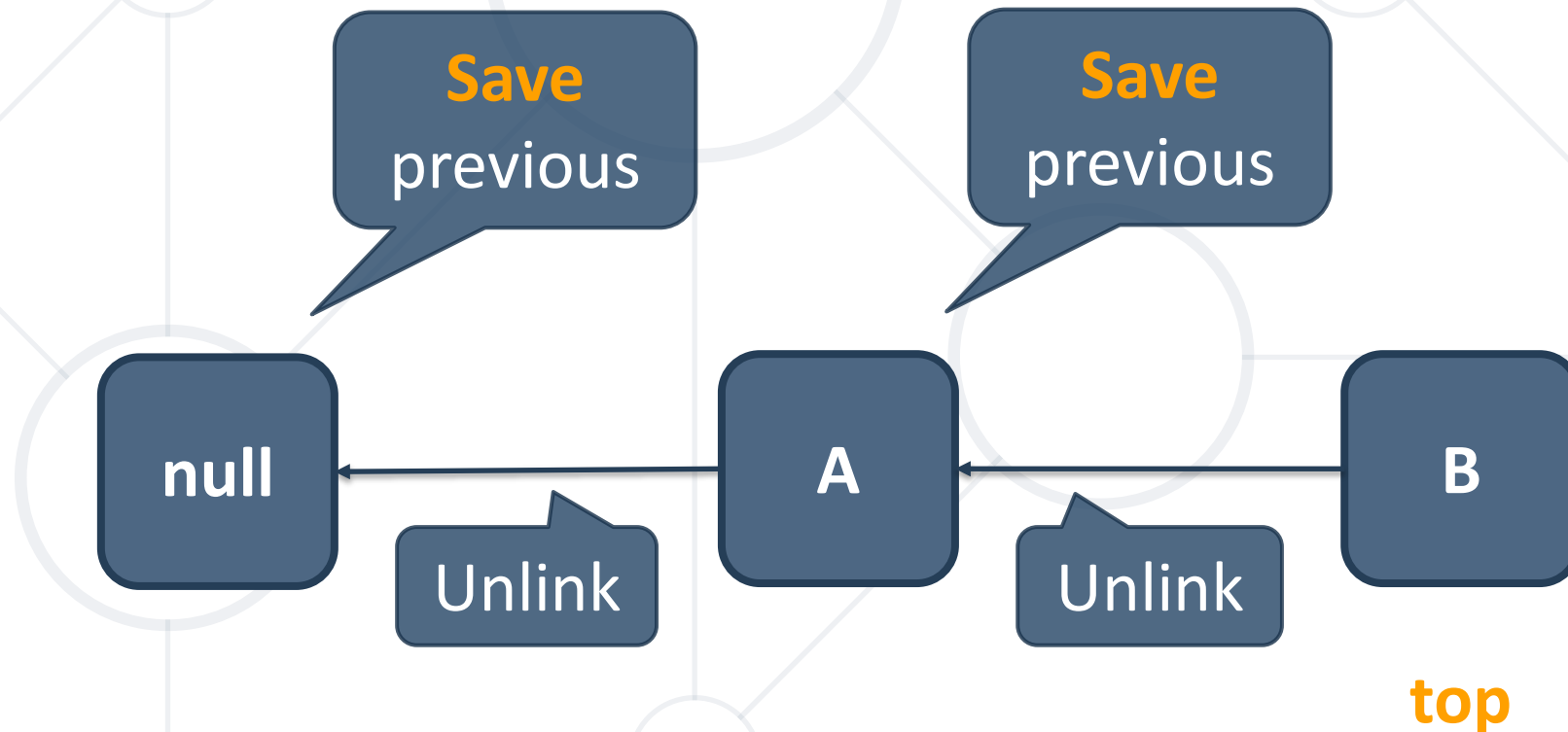

- Chain the nodes by using the **top** field:



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

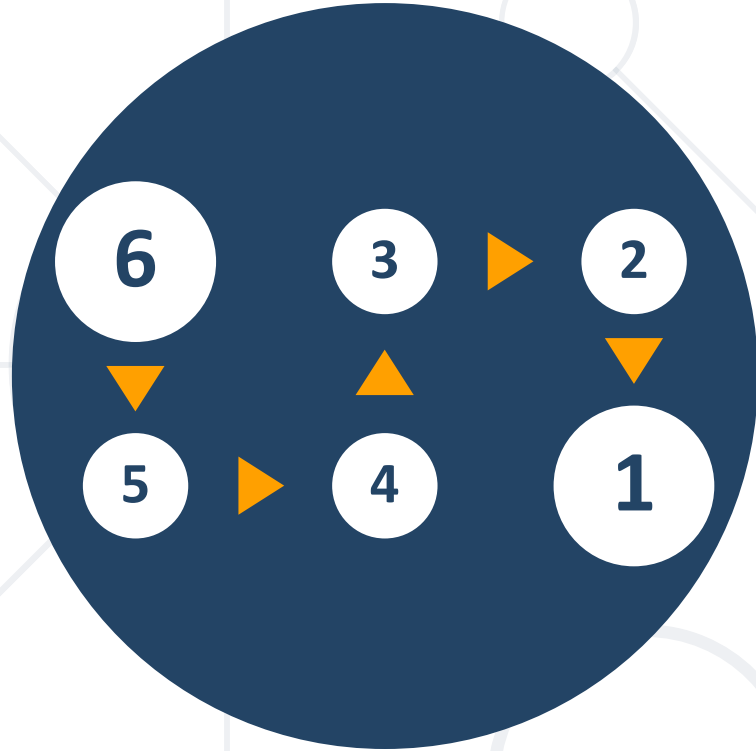
```
public void push(E element) {  
    Node<E> newNode = new Node<>(element);  
    newNode.previous = top;  
    top = newNode;  
    this.size++;  
}
```

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



- Remove and return element at the top:

```
public E pop() {  
    ensureNonEmpty();  
    E element = this.top.element;  
    Node<E> temp = this.top.previous;  
    this.top.previous = null;  
    this.top = temp;  
    this.size--;  
    return element;  
}
```



Queues

Dynamic Implementation

Queue

- Queue is the **implementation** of ADS **FIFO**
First In First Out
 - Build by using **Node** class or atop an **array**
- Queue example using Node

```
public class Queue<E> implements AbstractQueue<E> {  
    private Node<E> head;  
    private int size;  
}
```



- Supported operations and complexity:
 - **size(), isEmpty(), poll(), peek() – $O(1)$**
 - **offer():**
 - if we keep the reference to the that node – **$O(1)$**
 - If we have to chase pointers to that node – **$O(n)$**
 - all of the other operations run in linear time (roughly speaking):
 - **forEach(), contains(), etc...**

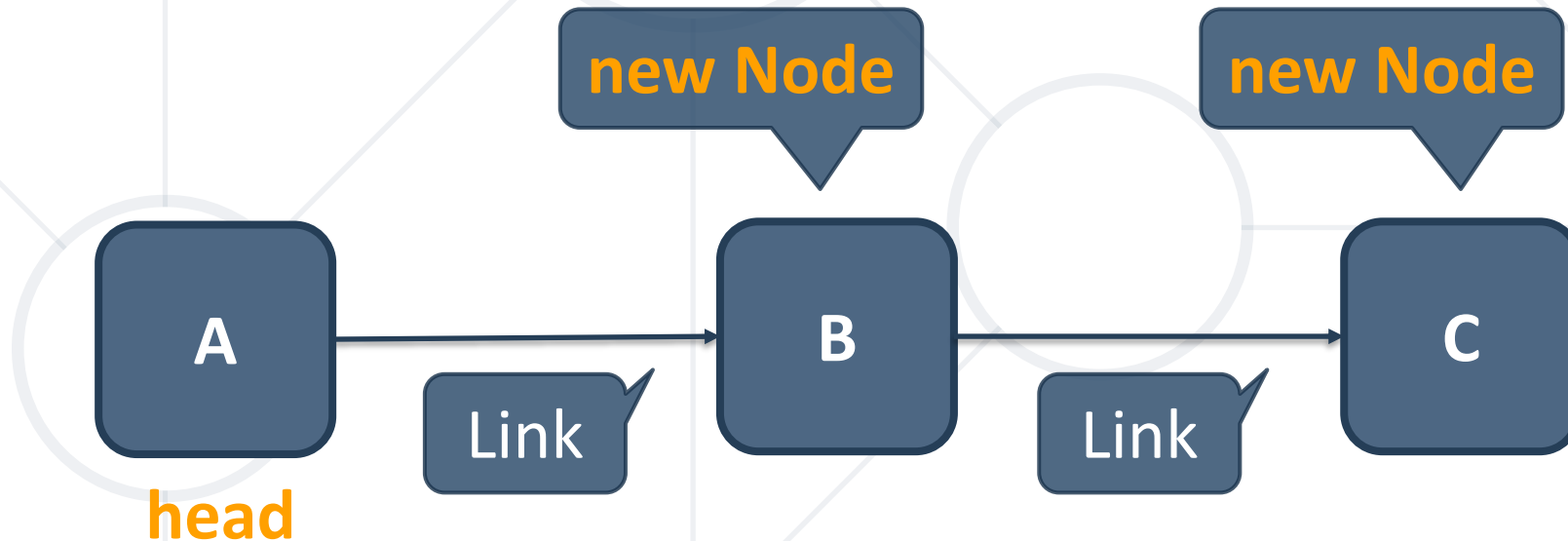
Queue – Constructor And Fields

- Constructor and fields:

```
public class Queue<E> implements AbstractQueue<E> {  
    private Node<E> head;  
    private int size;  
  
    private static class Node<E> {...}    // Node class  
  
    public SimpleQueue() {  
    }  
}
```


Queue – Offer

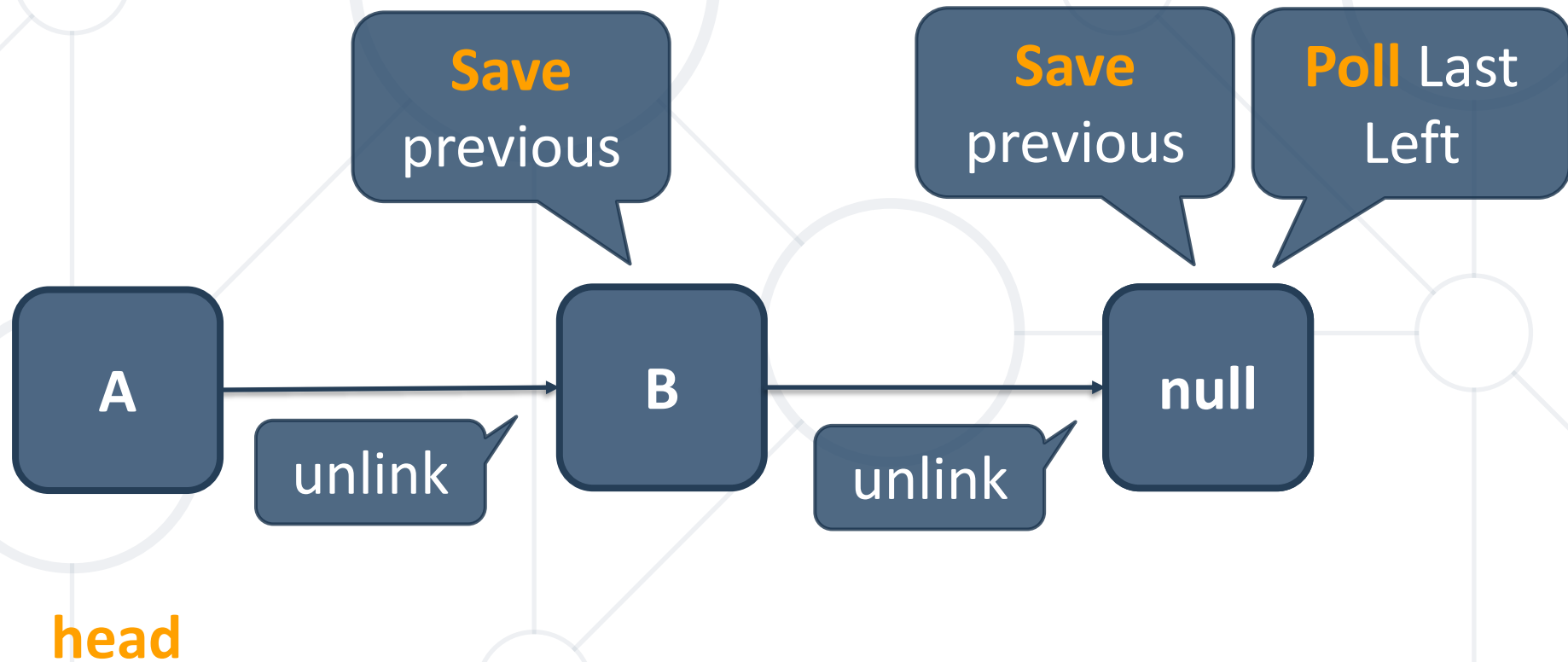
- Head == null \rightarrow head = new Node
- Size > 0 \rightarrow chain the nodes by adding new Node after the last one the so called tail:



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

```
public void offer(E element) {  
    Node<E> newNode = new Node<>(element);  
    if (this.head == null) {  
        this.head = newNode;  
    } else {  
        Node<E> current = this.head;  
        while (current.next != null) {  
            current = current.next;  
        }  
        current.next = newNode;  
    }  
    this.size++;  
}
```

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



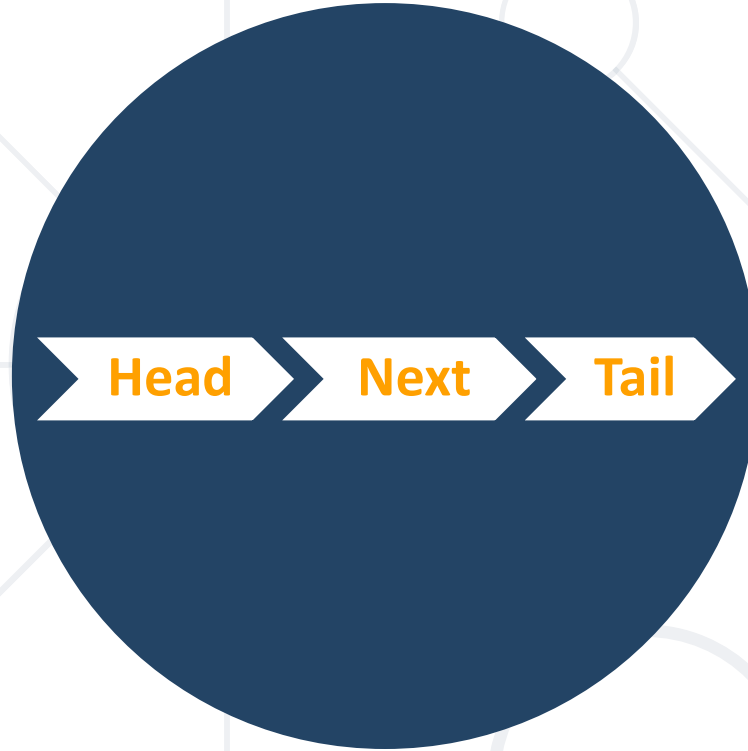
■ Stack

- 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, e.g.:
 - 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
 - However we define what is the entry point

```
public class SinglyLinkedList<E> implements LinkedList<E> {  
    private Node<E> head;  
    private int size;  
}
```

Singly Linked List – Operations

- Supported operations and complexity:
 - **addFirst(), removeFirst(), getFirst(), size() – $O(1)$**
 - How about operations on the **last element**?
 - **addLast(), removeLast(), getLast()** – again depends if we keep the reference to the last node or no can be constant – **$O(1)$** or linear – **$O(n)$**
 - operations that **index** into the list will run in **linear time $O(n)$** (roughly speaking)

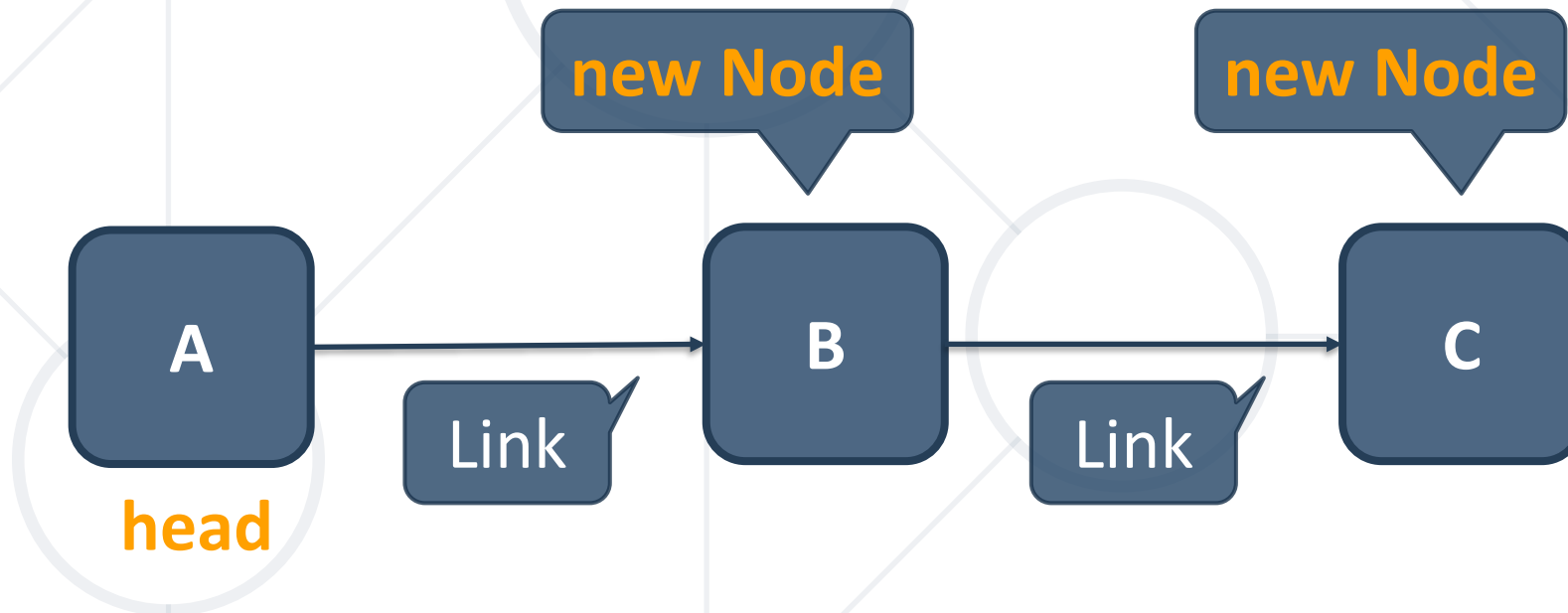
Singly LinkedList – Constructor And Fields

- Constructor and fields:

```
public class SinglyLinkedList<E> implements LinkedList<E> {  
    private Node<E> head;  
    private int size;  
  
    private static class Node<E> {...}    // Node class  
  
    public LinkedList() {  
    }  
}
```


Singly Linked List – Adding Last

- Head == null \rightarrow head = new Node
- Size > 0 \rightarrow

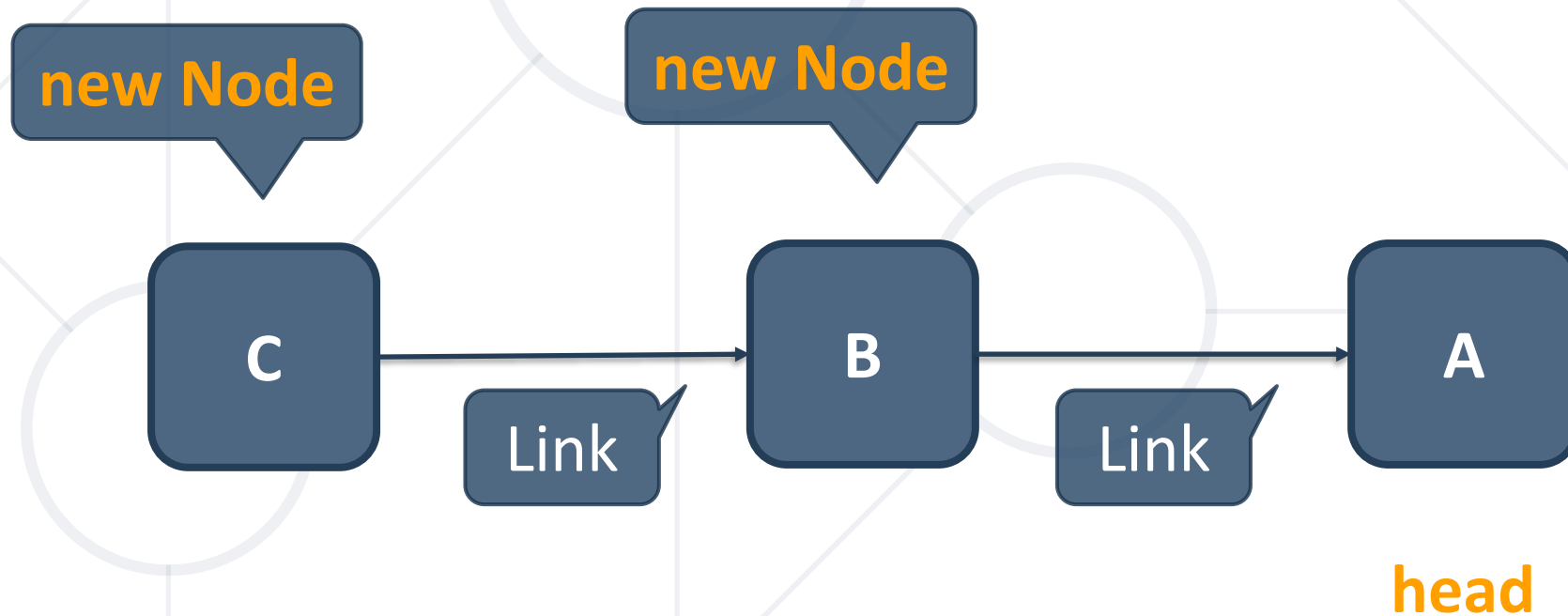


- Add element at the end:

```
public void addLast(E element) {  
    Node<E> newNode = new Node<>(element);  
    if (this.head == null) {  
        this.head = newNode;  
    } else {  
        Node<E> current = this.head;  
        while (current.next != null) {  
            current = current.next;  
        }  
        current.next = newNode;  
    }  
    this.size++;  
}
```

Singly Linked List – Adding First

- Head == null \rightarrow head = new Node
- Size > 0 \rightarrow

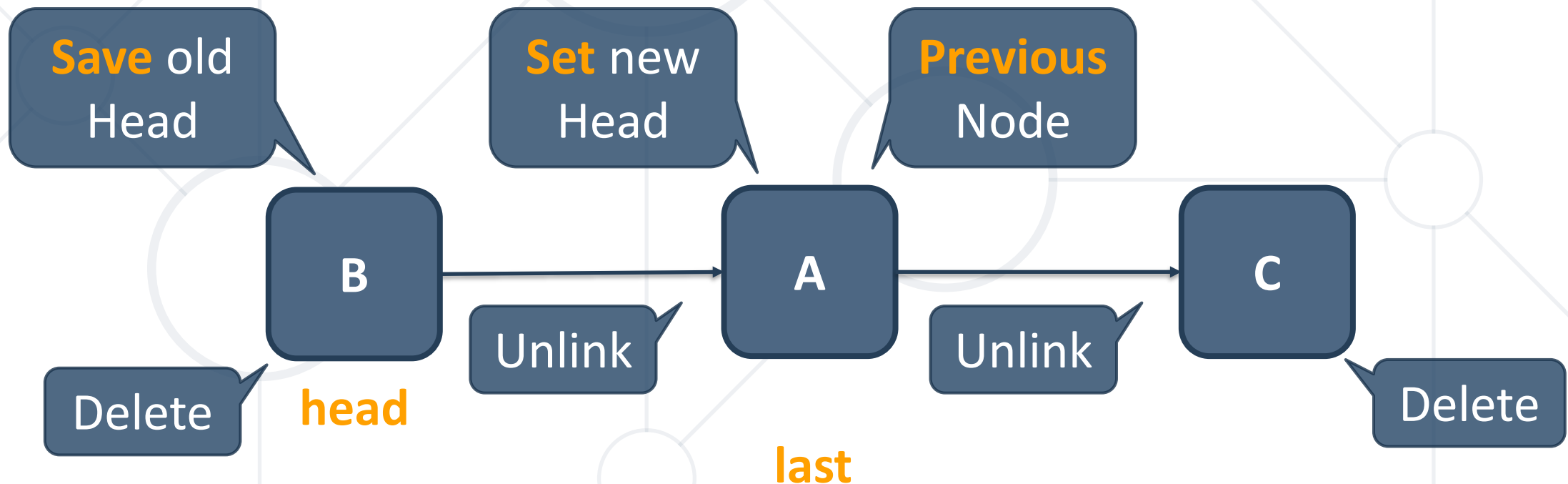


- Add element at the beginning:

```
public void addFirst(E element) {  
    Node<E> newNode = new Node<>(element);  
    if (this.head != null) {  
        newNode.next = this.head;  
    }  
    this.head = newNode;  
    this.size++;  
}
```

Linked List – Removing First/Last

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



Node Implementation

- So far we have implemented some Data Structures by using the **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.



- 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



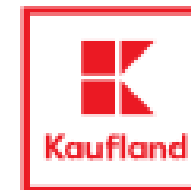
Questions?



SoftUni Diamond Partners

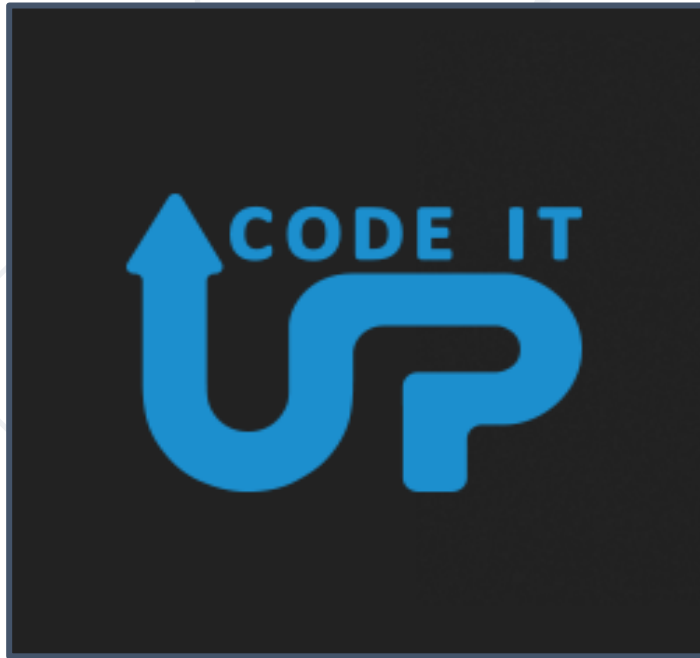


SCHWARZ



**SUPER
HOSTING
.BG**





VIRTUAL RACING SCHOOL



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

- Software University Forums

- forum.softuni.bg



- 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>

