

# Kreisrunde Bäume — Halin Graphen?

*Datastructures going wild*

# Kreisrunde Bäume — Halin Graphen?

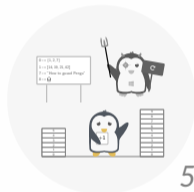
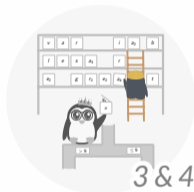
*Datastructures going wild*



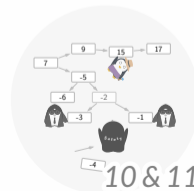
## Theorie



## Grundlagen



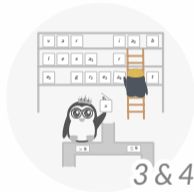
## Vertiefungen



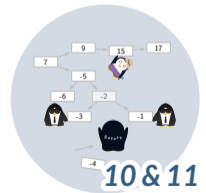
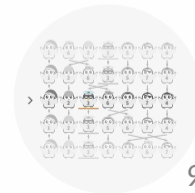
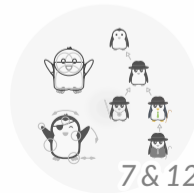
## Theorie



## Grundlagen



## Vertiefungen





---

Laufzeit

---

best

worst

Ansatz

---

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2), \mathcal{O}(n)$	$\mathcal{O}(n^2)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
Quick	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
Quick	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	Pivot → Ende, l sol. <, r sol. ≥. Treffen → tausche Pivot.

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
Quick	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	Pivot $\rightarrow$ Ende, l sol. $<$ , r sol. $\geq$ . Treffen $\rightarrow$ tausche Pivot.
Heap	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	

	Laufzeit		Ansatz
	best	worst	
Bubble	$\mathcal{O}(n^2)$ , $\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Vertausche benachbarte El., solange unsortiert.
Insertion	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
Selection	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
Merge	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
Quick	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	Pivot $\rightarrow$ Ende, l sol. $<$ , r sol. $\geq$ . Treffen $\rightarrow$ tausche Pivot.
Heap	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	Baue Heap, entferne wiederholt Wurzel, heapify.

		Laufzeit		Ansatz
		best	worst	
iterativ	Bubble	$O(n^2)$ , $O(n)$	$O(n^2)$	Vertausche benachbarte El., solange unsortiert.
	Insertion	$O(n)$	$O(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
	Selection	$O(n^2)$	$O(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
	Merge	$O(n \log n)$	$O(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
	Quick	$O(n \log n)$	$O(n^2)$	Pivot → Ende, l sol. <, r sol. ≥. Treffen → tausche Pivot.
	Heap	$O(n \log n)$	$O(n \log n)$	Baue Heap, entferne wiederholt Wurzel, heapify.

		Laufzeit		Ansatz
		best	worst	
iterativ	Bubble	$O(n^2)$ , $O(n)$	$O(n^2)$	Vertausche benachbarte El., solange unsortiert.
	Insertion	$O(n)$	$O(n^2)$	Sortiere 1. unsortiertes El. in sortierten Teil ein.
	Selection	$O(n^2)$	$O(n^2)$	Kleinstes unsortiertes El. an Ende des sortierten Teils.
rekursiv	Merge	$O(n \log n)$	$O(n \log n)$	Aufteilen bis einel., wiederholtes mergen sortierter Teill.
	Quick	$O(n \log n)$	$O(n^2)$	Pivot → Ende, l sol. <, r sol. ≥. Treffen → tausche Pivot.
	Heap	$O(n \log n)$	$O(n \log n)$	Baue Heap, entferne wiederholt Wurzel, heapify.



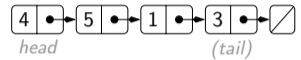
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

- Wir kennen eine Reihe an dynamischen Datenstrukturen:

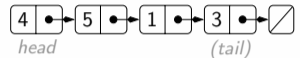
## Einfach verkettete Liste:



- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

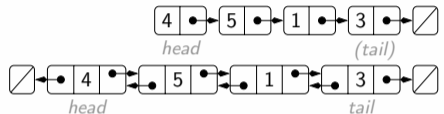
**Doppelt verkettete Liste:**



- > Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**

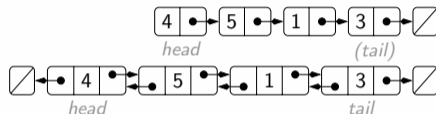


- > Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**

**Einfach verketteter (Binär-)Baum:**

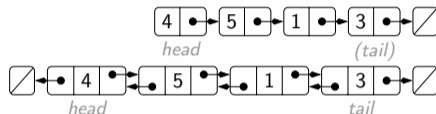
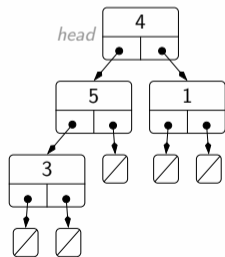


- › Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**

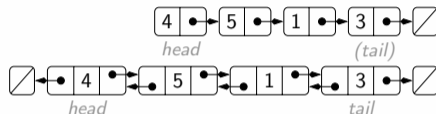
**Einfach verketteter (Binär-)Baum:**



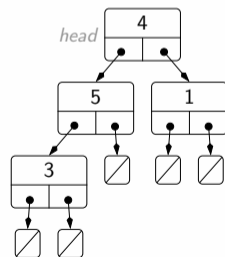
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

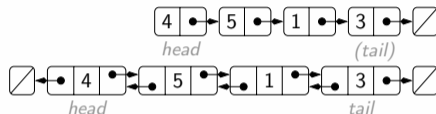


- Diese erlauben es, abstrakter Datentypen umsetzen:

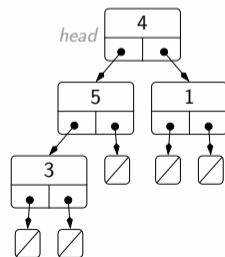
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

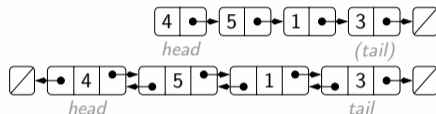


- Diese erlauben es, abstrakter Datentypen umsetzen:  
**Liste:**

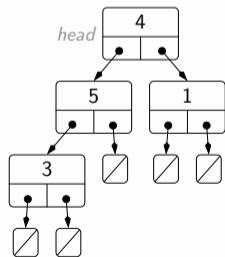
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

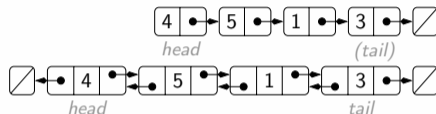


- Diese erlauben es, abstrakter Datentypen umsetzen:  
**Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das  $i$ -te Element, ist die Liste leer?, ...

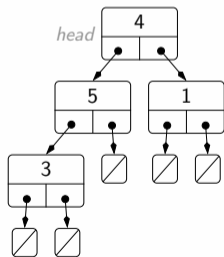
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

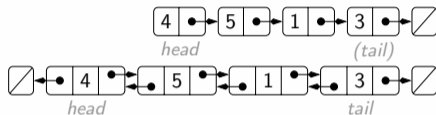


- Diese erlauben es, abstrakter Datentypen umsetzen:  
**Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das  $i$ -te Element, ist die Liste leer?, ...  
**Queue (FIFO):**

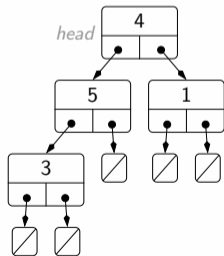
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

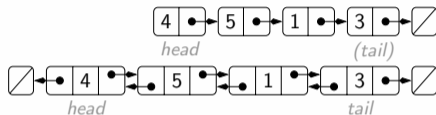


- Diese erlauben es, abstrakter Datentypen umsetzen:  
**Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das  $i$ -te Element, ist die Liste leer?, ...  
**Queue (FIFO):** Elemente hinzufügen und entfernen (in Einfügereihenfolge), ist die Queue leer?, ...

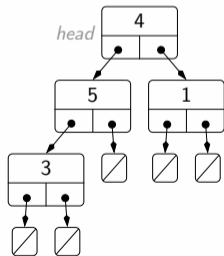
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

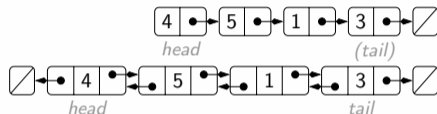


- Diese erlauben es, abstrakter Datentypen umsetzen:
  - Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das i-te Element, ist die Liste leer?, ...
  - Queue (FIFO):** Elemente hinzufügen und entfernen (in Einfügereihenfolge), ist die Queue leer?, ...
  - Stack (LIFO):**

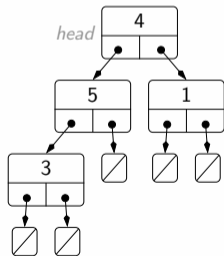
- > Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**

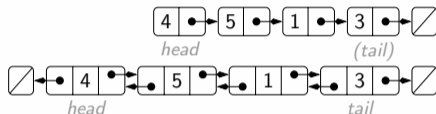


- > Diese erlauben es, abstrakter Datentypen umsetzen:  
**Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das i-te Element, ist die Liste leer?, ...  
**Queue (FIFO):** Elemente hinzufügen und entfernen (in Einfügereihenfolge), ist die Queue leer?, ...  
**Stack (LIFO):** Elemente hinzufügen und entfernen (umgekehrte Einfügereihenfolge), ist der Stack leer?, ...

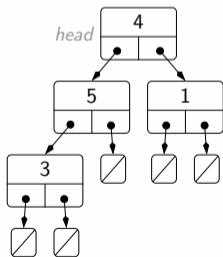
- Wir kennen eine Reihe an dynamischen Datenstrukturen:

**Einfach verkettete Liste:**

**Doppelt verkettete Liste:**



**Einfach verketteter (Binär-)Baum:**



- Diese erlauben es, abstrakter Datentypen umsetzen:
  - Liste:** Elemente vorne und hinten hinzufügen, Zugriff auf das i-te Element, ist die Liste leer?, ...
  - Queue (FIFO):** Elemente hinzufügen und entfernen (in Einfügereihenfolge), ist die Queue leer?, ...
  - Stack (LIFO):** Elemente hinzufügen und entfernen (umgekehrte Einfügereihenfolge), ist der Stack leer?, ...
- All diese abstrakten Datentypen haben ein klares (meist mathematisch definiertes) Verhalten.

# Präsenzaufgabe

1

Traverse My-Tree

In dieser Aufgabe sollen Sie binären Bäume aus Arrays wachsen lassen und diese anschließend in einem Breitendurchlauf traversieren. Hierfür finden Sie auf Moodle neben diesem Übungsblatt weitere Java Files (`BinaryTree.java`, `BinaryTreeMain.java`, `IntegerNode.java`), die Sie für Ihre Implementierung verwenden dürfen. Verwenden Sie ansonsten keine vorgefertigten dynamischen Datenstrukturen. Für die Darstellung der Knoten können Sie die vorgegebene Klasse `IntegerNode` verwenden, welche der einfach verketteten Version aus der Vorlesung entspricht.

Sie sollen die Methode `public void breadthFirstTraversal()` implementieren, welche den Baum in der Breite durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie die vorgegebene Klasse `Queue` benötigen, welche `IntegerNode` Elemente einreihen kann.

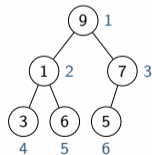
1

Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode `public void breadthFirstTraversal()`, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie `Queue` benötigen.

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode **public void breadthFirstTraversal()**, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie **Queue** benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:

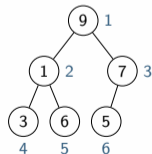


## 1

## Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode **public void breadthFirstTraversal()**, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie **Queue** benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:

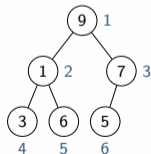


```
public class BinaryTree {  
    private IntegerNode root;  
    public BinaryTree(int[] items) { ... }  
    public void breadthFirstTraversal() { ★ }  
}
```

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode `public void breadthFirstTraversal()`, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie `Queue` benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:

```
public class BinaryTree {  
    private IntegerNode root;  
    public BinaryTree(int[] items) { ... }  
    public void breadthFirstTraversal() { ★ }  
}  
  
public class Queue {
```



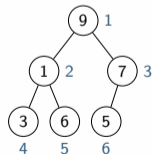
}

## 1

## Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode **public void breadthFirstTraversal()**, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie **Queue** benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:



```
public class BinaryTree {  
    private IntegerNode root;  
    public BinaryTree(int[] items) { ... }  
    public void breadthFirstTraversal() { ★ }  
}
```

```
public class Queue {  
    class Element {  
        public Element(IntegerNode node) { ... }  
        public void setNextElement(Element n) { ... }  
        public Element getNextElement() { ... }  
        public IntegerNode getNode() { ... }  
    }  
}
```

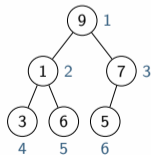
}

## 1

## Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode **public void breadthFirstTraversal()**, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie **Queue** benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:



```
public class BinaryTree {
    private IntegerNode root;
    public BinaryTree(int[] items) { ... }
    public void breadthFirstTraversal() { ★ }
}
```

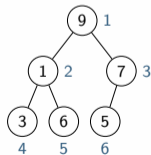
```
public class Queue {
    class Element {
        public Element(IntegerNode node) { ... }
        public void setNextElement(Element n) { ... }
        public Element getNextElement() { ... }
        public IntegerNode getNode() { ... }
    }
    public Queue() { ... }
    public void enqueue(IntegerNode node) { ... }
    public IntegerNode dequeue() { ... }
    public int getLength() { ... }
    public boolean isEmpty() { ... }
}
```

## 1

## Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode **public void breadthFirstTraversal()**, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie **Queue** benötigen.

**Zahlenbeispiel** Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:



```
public class BinaryTree {
    private IntegerNode root;
    public BinaryTree(int[] items) { ... }
    public void breadthFirstTraversal() { ★ }
}
```

```
public class IntegerNode {
    public IntegerNode(int value) { ... }
    public void setLeftChild(IntegerNode l) { ... }
    public IntegerNode getLeftChild() { ... }
    public void setValue(int value) { ... }
    public int getValue() { ... }
    ...
}
```

```
public class Queue {
    class Element {
        public Element(IntegerNode node) { ... }
        public void setNextElement(Element n) { ... }
        public Element getNextElement() { ... }
        public IntegerNode getNode() { ... }
    }
    public Queue() { ... }
    public void enqueue(IntegerNode node) { ... }
    public IntegerNode dequeue() { ... }
    public int getLength() { ... }
    public boolean isEmpty() { ... }
}
```



## TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

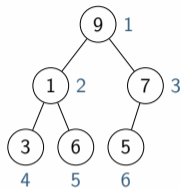
13. Juli 2022  
SP, Universität Ulm





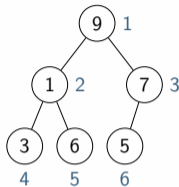
```
public void breadthFirstTraversal() {
```

```
}
```



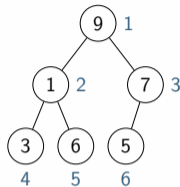
# Let it be code

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
}
```



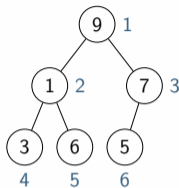
# Let it be code

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
}
```



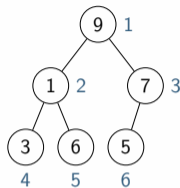
# Let it be code

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
  
    }  
}
```



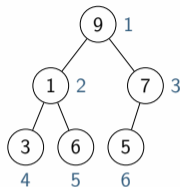
# Let it be code

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
  
    }  
}
```

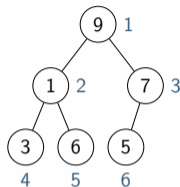


# Let it be code

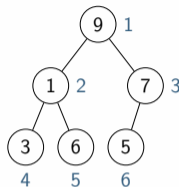
```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node.getLeftChild() != null)  
            queue.enqueue(node.getLeftChild());  
        if (node.getRightChild() != null)  
            queue.enqueue(node.getRightChild());  
    }  
}
```



```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node.getLeftChild() != null)  
            queue.enqueue(node.getLeftChild());  
    }  
}
```

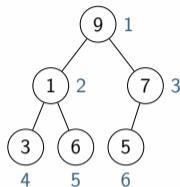


```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node.getLeftChild() != null)  
            queue.enqueue(node.getLeftChild());  
        if (node.getRightChild() != null)  
            queue.enqueue(node.getRightChild());  
    }  
}
```



# Let it be code

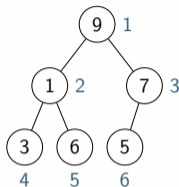
```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node.getLeftChild() != null)  
            queue.enqueue(node.getLeftChild());  
        if (node.getRightChild() != null)  
            queue.enqueue(node.getRightChild());  
        System.out.println(node.getValue());  
    }  
}
```



# Let it be code

 [BinaryTree.java](#), [BinaryTreeMain.java](#), [IntegerNode.java](#), [Queue.java](#)

```
public void breadthFirstTraversal() {  
    Queue queue = new Queue();  
    queue.enqueue(root);  
  
    while (!queue.isEmpty()) {  
        IntegerNode node = queue.dequeue();  
        if (node.getLeftChild() != null)  
            queue.enqueue(node.getLeftChild());  
        if (node.getRightChild() != null)  
            queue.enqueue(node.getRightChild());  
        System.out.println(node.getValue());  
    }  
}
```

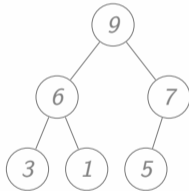


# Übungsblatt 10

# Aufgabe 1: Bäume

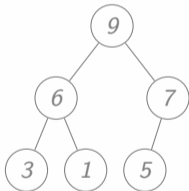
# Aufgabe 1: Bäume

Betrachten Sie den folgenden Binärbaum:

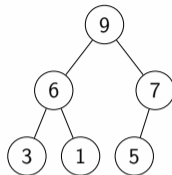


# Aufgabe 1: Bäume

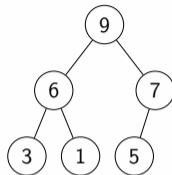
Betrachten Sie den folgenden Binärbaum:



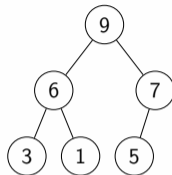
- a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.
- b) Welche Tiefe hat der Baum?
- c) Wie viele Knoten hat der Baum und wie viele davon sind Blätter?
- d) Formt der Baum einen Max-Heap? Begründen Sie Ihre Antwort kurz.
- e) In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?
- f) In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?



a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.

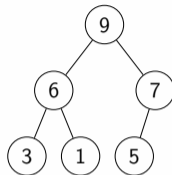


- a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.  
2, alle Knoten haben *höchstens* zwei Kinder.



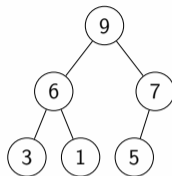
# Grundlagen Fragen

- a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.  
2, alle Knoten haben *höchstens* zwei Kinder.
- b) Welche Tiefe hat der Baum?



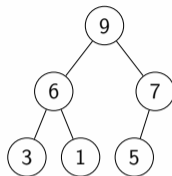
# Grundlagen Fragen

- a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.  
2, alle Knoten haben *höchstens* zwei Kinder.
- b) Welche Tiefe hat der Baum?  
3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.



# Grundlagen Fragen

- a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.  
2, alle Knoten haben *höchstens* zwei Kinder.
- b) Welche Tiefe hat der Baum?  
3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.
- c) Wie viele Knoten hat der Baum und wie viele davon sind Blätter?



a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.

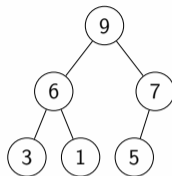
2, alle Knoten haben *höchstens* zwei Kinder.

b) Welche Tiefe hat der Baum?

3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.

c) Wie viele Knoten hat der Baum und wie viele davon sind Blätter?

6 Knoten ( $\{3, 6, 9, 1, 7, 5\}$ ), 3 davon sind Blätter (Knoten ohne Kinder,  $\{3, 1, 5\}$ )



a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.

2, alle Knoten haben *höchstens* zwei Kinder.

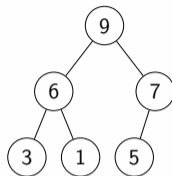
b) Welche Tiefe hat der Baum?

3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.

c) Wie viele Knoten hat der Baum und wie viele davon sind Blätter?

6 Knoten  $(\{3, 6, 9, 1, 7, 5\})$ , 3 davon sind Blätter (Knoten ohne Kinder,  $\{3, 1, 5\}$ )

d) Formt der Baum einen Max-Heap? Begründen Sie Ihre Antwort kurz.



a) Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.

2, alle Knoten haben *höchstens* zwei Kinder.

b) Welche Tiefe hat der Baum?

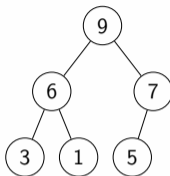
3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.

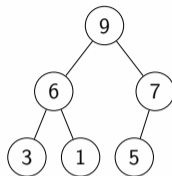
c) Wie viele Knoten hat der Baum und wie viele davon sind Blätter?

6 Knoten  $\{3, 6, 9, 1, 7, 5\}$ , 3 davon sind Blätter (Knoten ohne Kinder,  $\{3, 1, 5\}$ )

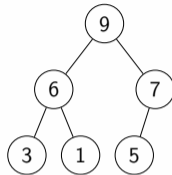
d) Formt der Baum einen Max-Heap? Begründen Sie Ihre Antwort kurz.

Ja, die Elternknoten sind immer kleiner als alle ihre Kinder.



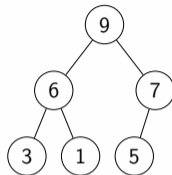


e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*



e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

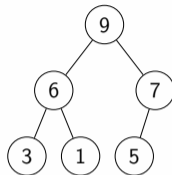
**Prä-Order:** 9, 6, 3, 1, 7, 5



e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

**In-Order:** 3, 6, 1, 9, 5, 7

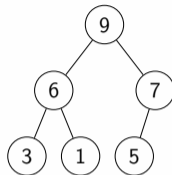


e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

**In-Order:** 3, 6, 1, 9, 5, 7

**Post-Order:** 3, 1, 6, 5, 7, 9



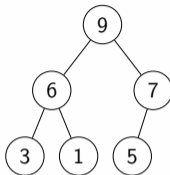
e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

**In-Order:** 3, 6, 1, 9, 5, 7

**Post-Order:** 3, 1, 6, 5, 7, 9

Unter Angabe der Variante (Prä-, In- oder Post-Order) reicht hier auch nur eine Version.



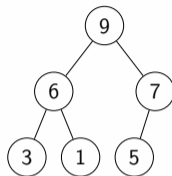
e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

**In-Order:** 3, 6, 1, 9, 5, 7

**Post-Order:** 3, 1, 6, 5, 7, 9

Unter Angabe der Variante (Prä-, In- oder Post-Order) reicht hier auch nur eine Version.



f) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?*

- e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

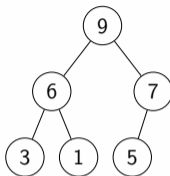
**In-Order:** 3, 6, 1, 9, 5, 7

**Post-Order:** 3, 1, 6, 5, 7, 9

Unter Angabe der Variante (Prä-, In- oder Post-Order) reicht hier auch nur eine Version.

- f) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?*

9, 6, 7, 3, 1, 5.



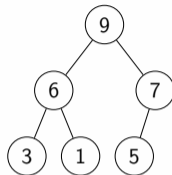
- e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

**Prä-Order:** 9, 6, 3, 1, 7, 5

**In-Order:** 3, 6, 1, 9, 5, 7

**Post-Order:** 3, 1, 6, 5, 7, 9

Unter Angabe der Variante (Prä-, In- oder Post-Order) reicht hier auch nur eine Version.



- f) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?*

9, 6, 7, 3, 1, 5.

Neben Pre-, In- und Post-Order gibt es für spezifische Bäume (wie Binärbäume) auch noch weitere Versionen, die uns aber erstmal egal sind.



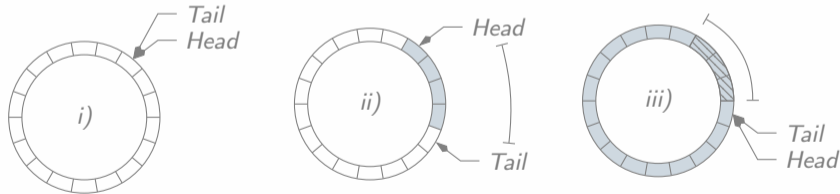
## Aufgabe 2: Self-Overriding Ring Buffer

## Aufgabe 2: Self-Overriding Ring Buffer

*In dieser Aufgabe sollen Sie einen Ring Buffer implementieren. Hierbei handelt es sich um eine dynamische Datenstruktur, ähnlich zu einer `DoublyLinkedList` aus der Vorlesung, jedoch verweist hier jedes Element auf seinen Vorgänger und Nachfolger, wodurch sie einen Ring formen. Zusätzlich hat ein Ring Buffer eine fest vorgegebene Kapazität. Hierfür werden intern zwei Referenzen `head` und `tail` verwendet, die so Beginn und Ende des belegten Bereichs markieren. Alle Positionen bis zum `tail` sind beginnend beim `head` Zeiger belegt. Wir betrachten ein modifiziertes Ring Buffer, bei welchem alte Elemente überschrieben werden, falls der Buffer voll ist. Der Ring Buffer soll nach dem First-In-First-Out Prinzip arbeiten: neue Elemente werden an `tail` angefügt und es werden vom `head` aus Elemente abgearbeitet. Ein Beispiel finden Sie im Folgenden.*

## Aufgabe 2: Self-Overriding Ring Buffer

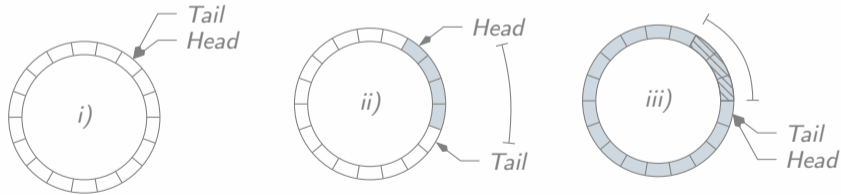
In dieser Aufgabe sollen Sie einen Ring Buffer implementieren. Hierbei handelt es sich um eine dynamische Datenstruktur, ähnlich zu einer `DoublyLinkedList` aus der Vorlesung, jedoch verweist hier jedes Element auf seinen Vorgänger und Nachfolger, wodurch sie einen Ring formen. Zusätzlich hat ein Ring Buffer eine fest vorgegebene Kapazität. Hierfür werden intern zwei Referenzen `head` und `tail` verwendet, die so Beginn und Ende des belegten Bereichs markieren. Alle Positionen bis zum `tail` sind beginnend beim `head` Zeiger belegt. Wir betrachten ein modifiziertes Ring Buffer, bei welchem alte Elemente überschrieben werden, falls der Buffer voll ist. Der Ring Buffer soll nach dem First-In-First-Out Prinzip arbeiten: neue Elemente werden an `tail` angefügt und es werden vom `head` aus Elemente abgearbeitet. Ein Beispiel finden Sie im Folgenden.



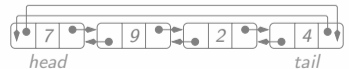
i) Der RingBuffer ist leer und `head`, sowie `tail` zeigen auf dasselbe Element. ii) Teilweise belegt, nachdem vier Elemente eingefügt wurden und `head`, sowie `tail` nun auf verschiedenen Elemente zeigen. iii) Der Buffer ist voll (`tail` hat `head` „überundet“) und wird nun überschreiben.

## Aufgabe 2: Self-Overriding Ring Buffer

In dieser Aufgabe sollen Sie einen Ring Buffer implementieren. Hierbei handelt es sich um eine dynamische Datenstruktur, ähnlich zu einer `DoublyLinkedList` aus der Vorlesung, jedoch verweist hier jedes Element auf seinen Vorgänger und Nachfolger, wodurch sie einen Ring formen. Zusätzlich hat ein Ring Buffer eine fest vorgegebene Kapazität. Hierfür werden intern zwei Referenzen `head` und `tail` verwendet, die so Beginn und Ende des belegten Bereichs markieren. Alle Positionen bis zum `tail` sind beginnend beim `head` Zeiger belegt. Wir betrachten ein modifiziertes Ring Buffer, bei welchem alte Elemente überschrieben werden, falls der Buffer voll ist. Der Ring Buffer soll nach dem First-In-First-Out Prinzip arbeiten: neue Elemente werden an `tail` angefügt und es werden vom `head` aus Elemente abgearbeitet. Ein Beispiel finden Sie im Folgenden.



i) Der RingBuffer ist leer und `head`, sowie `tail` zeigen auf dasselbe Element. ii) Teilweise belegt, nachdem vier Elemente eingefügt wurden und `head`, sowie `tail` nun auf verschiedene Elemente zeigen. iii) Der Buffer ist voll (`tail` hat `head` „übrundet“) und wird nun überschrieben. Die Elemente sind wie eine `DoublyLinkedList` verbunden und haben zusätzlich eine Verknüpfung er beiden äußeren Elemente (unten rechts).



## Aufgabe 2: Self-Overriding Ring Buffer

## Aufgabe 2: Self-Overriding Ring Buffer

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

## Aufgabe 2: Self-Overriding Ring Buffer

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.
- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden.

## Aufgabe 2: Self-Overriding Ring Buffer

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.
- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden.
- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

## Aufgabe 2: Self-Overriding Ring Buffer

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.
- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden.
- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.
- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

## Aufgabe 2: Self-Overriding Ring Buffer

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.
- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden.
- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.
- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.
- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value;  
    private Element next;  
    public Element() { this.next = null; }  
    public void setNextElement(Element nextElement) { this.next = nextElement; }  
    public Element getNextElement() { return this.next; }  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value; private Element next;  
  
    public Element() { this.next = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value; private Element next;  
    private Element previous;  
    public Element() { this.next = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value; private Element next;  
    private Element previous;  
    public Element() { this.next = null; this.previous = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value; private Element next;  
    private Element previous;  
    public Element() { this.next = null; this.previous = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
    public void setPreviousElement(Element p) {  
        this.previous = p;  
    }  
  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

```
class Element {  
    private int value; private Element next;  
    private Element previous;  
    public Element() { this.next = null; this.previous = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
    public void setPreviousElement(Element p) {  
        this.previous = p;  
    }  
    public Element getPreviousElement() {  
        return this.previous;  
    }  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Extend the Element

- a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

 `Element.java`

```
class Element {  
    private int value; private Element next;  
    private Element previous;  
    public Element() { this.next = null; this.previous = null; }  
    public void setNextElement(Element n) { this.next = n; }  
    public Element getNextElement() { return this.next; }  
    public void setPreviousElement(Element p) {  
        this.previous = p;  
    }  
    public Element getPreviousElement() {  
        return this.previous;  
    }  
    public void setValue(int value) { this.value = value; }  
    public int getValue() { return this.value; }  
}
```

# Start des RingBuffers

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

```
public class RingBuffer {
```

```
}
```

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

```
public class RingBuffer {  
    private Element head;  
    private Element tail;  
  
}
```

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

```
public class RingBuffer {  
    private Element head;  
    private Element tail;  
    private boolean isEmpty;
```

```
}
```

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

```
public class RingBuffer {  
    private Element head;  
    private Element tail;  
    private boolean isEmpty;  
  
    public RingBuffer(int capacity) {  
  
    }  
}
```

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

```
public class RingBuffer {  
    private Element head;  
    private Element tail;  
    private boolean isEmpty;  
  
    public RingBuffer(int capacity) {  
        . . .  
    }  
}
```

# Start des RingBuffers

- b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

 `RingBuffer.java`

```
public class RingBuffer {  
    private Element head;  
    private Element tail;  
    private boolean isEmpty;  
  
    public RingBuffer(int capacity) {  
        . . .  
    }  
}
```







# Der Konstruktor

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
  
    }  
}
```

# Der Konstruktor

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
  
        Element current = new Element();  
  
    }  
}
```

# Der Konstruktor

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
  
        Element current = new Element();  
        this.head = this.tail = current;  
        this.isEmpty = true;  
  
    }  
}
```

# Der Konstruktor

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
  
        Element current = new Element();  
        this.head = this.tail = current;  
        this.isEmpty = true;  
        current = fillBuffer(capacity, current);  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
  
        Element current = new Element();  
        this.head = this.tail = current;  
        this.isEmpty = true;  
        current = fillBuffer(capacity, current);  
        this.head.setPreviousElement(current);  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) {  
        if (capacity == 0) { this.isEmpty = false; return; }  
        else if (capacity < 0) { throw new NegativeArraySizeException(); }  
  
        Element current = new Element();  
        this.head = this.tail = current;  
        this.isEmpty = true;  
        current = fillBuffer(capacity, current);  
        this.head.setPreviousElement(current);  
        current.setNextElement(head);  
    }  
}
```



}

}

```
public class RingBuffer {
    private Element head, tail; private boolean isEmpty;
    RingBuffer(int capacity) { ... }

    private Element fillBuffer(int capacity, Element current) {

    }
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
  
        }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
            Element next = new Element();  
  
        }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
            Element next = new Element();  
            current.setNextElement(next);  
            next.setPreviousElement(current);  
        }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
            Element next = new Element();  
            current.setNextElement(next);  
            next.setPreviousElement(current);  
            if (i == capacity - 1)  
                next.setNextElement(head);  
        }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
            Element next = new Element();  
            current.setNextElement(next);  
            next.setPreviousElement(current);  
            if (i == capacity - 1)  
                next.setNextElement(head);  
            current = next;  
        }  
    }  
}
```

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    private Element fillBuffer(int capacity, Element current) {  
        for (int i = 1; i < capacity; i++) {  
            Element next = new Element();  
            current.setNextElement(next);  
            next.setPreviousElement(current);  
            if (i == capacity - 1)  
                next.setNextElement(head);  
            current = next;  
        }  
        return current;  
    }  
}
```



- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
}
```

- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    public int peek() {  
  
    }  
}
```

- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    public int peek() {  
        if (this.isEmpty)  
  
        else  
  
    }  
}
```

- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    public int peek() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        else  
  
    }  
}
```

- c) Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
    public int peek() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        else  
            return head.getValue();  
    }  
}
```

# Put Put Put

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
}
```

# Put Put Put

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public void put(int value) {  
  
    }  
}
```

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public void put(int value) {  
        if (this.tail == this.head && !this.isEmpty)  
            this.head = this.head.getNextElement();  
  
    }  
}
```

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public void put(int value) {  
        if (this.tail == this.head && !this.isEmpty)  
            this.head = this.head.getNextElement();  
        this.tail.setValue(value);  
  
    }  
}
```

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public void put(int value) {  
        if (this.tail == this.head && !this.isEmpty)  
            this.head = this.head.getNextElement();  
        this.tail.setValue(value);  
        this.tail = this.tail.getNextElement();  
    }  
}
```

- d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public void put(int value) {  
        if (this.tail == this.head && !this.isEmpty)  
            this.head = this.head.getNextElement();  
        this.tail.setValue(value);  
        this.tail = this.tail.getNextElement();  
        this.isEmpty = false;  
    }  
}
```

# Operation Removal

# Operation Removal

- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

# Operation Removal

- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
  
}
```

```
}
```

# Operation Removal

e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
  
    }  
}
```

# Operation Removal

e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
    }  
}
```

# Operation Removal

e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        int value = this.head.getValue();  
  
    }  
}
```

# Operation Removal

- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        int value = this.head.getValue();  
        this.head = this.head.getNextElement();  
    }  
}
```

# Operation Removal

- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        int value = this.head.getValue();  
        this.head = this.head.getNextElement();  
        this.isEmpty = this.head == tail;  
    }  
}
```

# Operation Removal

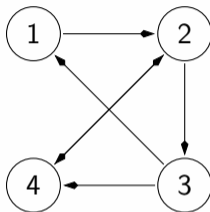
- e) Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.

```
public class RingBuffer {  
    private Element head, tail; private boolean isEmpty;  
    RingBuffer(int capacity) { ... }  
    public int remove() {  
        if (this.isEmpty)  
            throw new NoSuchElementException();  
        int value = this.head.getValue();  
        this.head = this.head.getNextElement();  
        this.isEmpty = this.head == tail;  
        return value;  
    }  
}
```

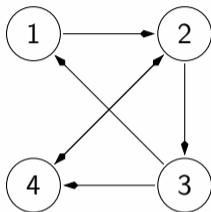
# Aussicht: Übungsblatt 11

# Aufgabe 1: Graphen

# Aufgabe 1: Graphen

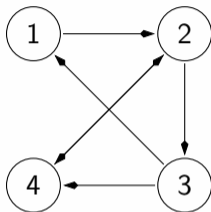


# Aufgabe 1: Graphen



$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

# Aufgabe 1: Graphen



$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

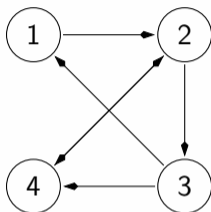
1.  $\rightarrow 2$

2.  $\rightarrow 3 \rightarrow 4$

3.  $\rightarrow 1 \rightarrow 4$

4.  $\rightarrow 2$

# Aufgabe 1: Graphen



$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

1.  $\rightarrow 2$

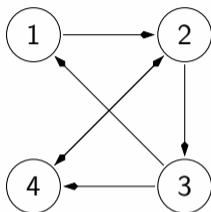
2.  $\rightarrow 3 \rightarrow 4$

3.  $\rightarrow 1 \rightarrow 4$

4.  $\rightarrow 2$

- › Je nach Kontext bietet sich die Adjazenzmatrix (links) oder die Adjazenzliste (rechts) mehr an

# Aufgabe 1: Graphen



$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

1.  $\rightarrow 2$

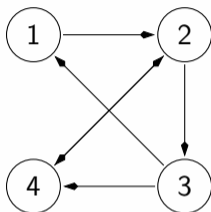
2.  $\rightarrow 3 \rightarrow 4$

3.  $\rightarrow 1 \rightarrow 4$

4.  $\rightarrow 2$

- › Je nach Kontext bietet sich die Adjazenzmatrix (links) oder die Adjazenzliste (rechts) mehr an
- › Für die Durchläufe markiert S den Startknoten

# Aufgabe 1: Graphen



$$\begin{array}{c} 1 \quad 2 \quad 3 \quad 4 \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

1.  $\rightarrow 2$

2.  $\rightarrow 3 \rightarrow 4$

3.  $\rightarrow 1 \rightarrow 4$

4.  $\rightarrow 2$

- › Je nach Kontext bietet sich die Adjazenzmatrix (links) oder die Adjazenzliste (rechts) mehr an
- › Für die Durchläufe markiert S den Startknoten
- › Auch hier hilft die Episode

# Aufgaben 2 und 3

## Aufgaben 2 und 3

- › Die Implementation eines Stacks sollte nach Schema F funktionieren

## Aufgaben 2 und 3

- › Die Implementation eines Stacks sollte nach Schema F funktionieren
- › Bei der Baum-Traversierung nicht abschrecken lassen

## Aufgaben 2 und 3

- › Die Implementation eines Stacks sollte nach Schema F funktionieren
- › Bei der Baum-Traversierung nicht abschrecken lassen
  - Bei Problemen erst einen kleinen Baum mit nur einer Operation probieren

## Aufgaben 2 und 3

- › Die Implementation eines Stacks sollte nach Schema F funktionieren
- › Bei der Baum-Traversierung nicht abschrecken lassen
  - Bei Problemen erst einen kleinen Baum mit nur einer Operation probieren
  - Gerne auch Unterstützung für weitere Funktionen!

# Abschließendes



- > Meldet euch für die Übungsleistung an

- Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)

- Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“

- > Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“
- > Meldet euch auch für die Prüfung an

- > Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“
- > Meldet euch auch für die Prüfung an
- > Feedback bezüglich des Tutoriums wird immer gern gesehen

- > Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“
- > Meldet euch auch für die Prüfung an
- > Feedback bezüglich des Tutoriums wird immer gern gesehen
  - Ist hier eine anonyme Umfrage erwünscht?

- > Meldet euch für die Übungsleistung an
  - Dies geht in [campusonline](#)
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“
- > Meldet euch auch für die Prüfung an
- > Feedback bezüglich des Tutoriums wird immer gern gesehen
  - Ist hier eine anonyme Umfrage erwünscht?
- > Die Betrachtung von Algorithmen und Datenstrukturen ist elementar!



