11

Einfache Kreise und Inversion

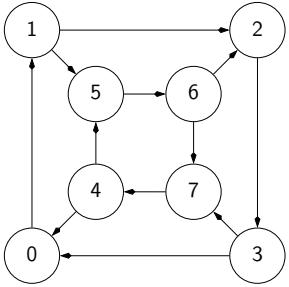
Tutorium 11

Florian Sihler ○ KW 5

Präsenzaufgabe

1 **Nächste Ausfahrt links**

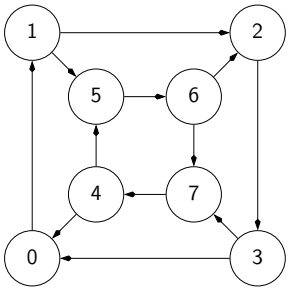
Geben Sie die Adjazenzliste für den Graphen links an und zeichnen Sie den Graphen für die Adjazenzmatrix rechts.



$$\begin{pmatrix} 0 & 3 & 0 & 0 & 6 \\ 3 & 0 & 5 & 0 & 7 \\ 0 & 5 & 0 & 1 & 9 \\ 4 & 0 & 1 & 0 & 2 \\ 6 & 7 & 9 & 2 & 0 \end{pmatrix}$$

Präsenzaufgabe - Lösung

- Zuerst eine Adjazenzliste:



$0 \to 1$

$1 \to 2 \to 5$

$2 \to 3$

$3 \to 7 \to 0$

$4 \to 5 \to 0$

$5 \to 6$

$6 \to 2 \to 7$

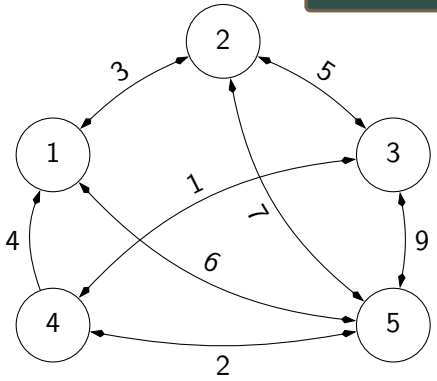
$7 \to 4$

Präsenzaufgabe - Lösung

- Nun mit der Adjazenzmatrix (Kanten wie ⟷ können auch ungerichtet dargestellt werden):

Natürlich können die Kanten in die andere Richtung auch ein anderes Gewicht aufweisen.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 3 & 0 & 0 & 6 \\ 3 & 0 & 5 & 0 & 7 \\ 0 & 5 & 0 & 1 & 9 \\ 4 & 0 & 1 & 0 & 2 \\ 6 & 7 & 9 & 2 & 0 \end{pmatrix} \end{array}$$



Übungsblatt 11 - Aufgabe 1

In dieser Aufgabe sollen Sie ein Programm schreiben, welches arithmetische Ausdrücke mit der umgekehrten polnischen Notation abarbeiten kann. Dabei handelt es sich um eine Schreibweise, bei der die Operatoren hinter den Operanden stehen, und nicht dazwischen wie bei der üblichen Infixnotation. Der arithmetische Ausdruck $(2 + 5) \times (4 - 1)$ lässt sich mit der UPN folgendermaßen darstellen: `2 5 + 4 1 - ×` Mit Hilfe dieser Notation lassen sich arithmetische Ausdrücke stapelweise abarbeiten. Um dies in der Praxis umzusetzen, sollen Sie folgendermaßen vorgehen:

1. Zuerst sollen Sie einen Stack implementieren, welcher nach dem `LIFO`-Prinzip arbeitet („last in first out“). In dieser Aufgabe soll dieser Stack Integer Werte verwalten. Hierfür benötigen Sie neben einem Konstruktor die folgenden Methoden:
   - Die Methode `public void push(int value)` soll den übergebenen Wert oben auf dem Stack platzieren.
   - Die Methode `public int pop()` soll den obersten Wert auf dem Stack entfernen und zurückgeben. Beachten Sie, dass der Stack unter Umständen leer sein kann. Werfen Sie in diesem Fall eine Exception.

   Verwenden Sie für Ihre Implementierung keine Arrays oder vorgefertigten dynamischen Datenstrukturen, sondern implementieren Sie den Stack selbst als (einfach) verkettete Liste.
2. Erweitern Sie ihre Stack-Implementierung nun um die folgenden Methoden:
   - Die Methode `public void add()` soll die obersten zwei Elemente auf dem Stack addieren und das Ergebnis auf den Stack pushen.
   - Die Methode `public void subtract()` soll das oberste Element vom darauffolgenden Element subtrahieren und das Ergebnis auf den Stack pushen.
   - Die Methode `public void multiply()` soll die obersten zwei Elemente auf dem Stack multiplizieren und das Ergebnis auf den Stack pushen.

   Beachten Sie bei diesen Methoden den Fall, dass eventuell weniger als zwei Elemente auf dem Stack vorhanden sein könnten.

Übungsblatt 11 - Aufgabe 1a) Stack

- Unser Stack baut sich erstmal ganz analog zur Queue von letzter Woche (RPNStack.java).

```
public class RPNStack {
    class Element {
        public final int value;
        public Element next = null;
        public Element(int value) { this.value = value; }
    }
    private Element first;
    // private Element last;
    private int size;
}
```

- Allerdings nehmen wir bei einem Stack die Elemente nur von einer Seite.
- Wir sparen uns also `last` und nennen die Klasse nun `RPNStack`.

Übungsblatt 11 - Aufgabe 1a) Stack push

- Anstelle von `enqueue` und `dequeue` nun `push` und `pop`:

```
public class RPNStack {
    class Element { final int value; Element next = null; }
    private Element first;
    private int size;
    public void push(int value) {
        Element top = new Element(value);
        top.next = this.first;
        this.first = top;
        this.size++;
    }
}
```

Übungsblatt 11 - Aufgabe 1a) Stack pop

```
public class RPNStack {
    class Element { final int value; Element next = null; }
    private Element first;
    private int size;
    public void push(int value) { ... }

    public int pop() {
        if(this.size == 0) throw new NoSuchElementException();
        int value = this.first.value;
        this.first = this.first.next;
        this.size--;
        return value;
    }
}
```

Übungsblatt 11 - Aufgabe 1b)

- Wir können für jede Operation eine Methode wie folgt implementieren:

```
public void subtract() {
    if(this.size < 2) throw new NoSuchElementException(); // Nötig? → Nö
    int a = this.pop();
    int b = this.pop();
    this.push(b - a); // this.push(-this.pop() + this.pop())
}
```

- Aber wo wäre da der Spaß? Wir wollen eine Enumeration! (Als Übung. Natürlich.)

```
public class RPNStack {
    class Element { final int value; Element next = null; }
    ...
   private enum BinaryOperation {
      ADD, SUB, MUL
   }
}
```

Aufgabe für die #coolenKids™ Wie kann man das mit Lambdas/Methoden Referenzen noch mehr #exciting machen?

Übungsblatt 11 - Aufgabe 1b) BinaryOperation

```
public class RPNStack {
    ...
    public void push(int value) { ... }
    public int pop() throws NoSuchElementException { ... }
    private enum BinaryOperation { ADD, SUB, MUL }
    private void performBinaryOperation(BinaryOperation operation) {
        if(this.size < 2) throw new NoSuchElementException(); // Nötig? → Nopsie
        int a = this.pop();
        int b = this.pop();
        switch(operation) {
            case ADD: this.push(a + b); break;
            case SUB: this.push(b - a); break;
            case MUL: this.push(a * b); break;
        }
    }
    public void add() { this.performBinaryOperation(BinaryOperation.ADD); }
    public void subtract() { this.performBinaryOperation(BinaryOperation.SUB); }
    public void multiply() { this.performBinaryOperation(BinaryOperation.MUL); }
}
```

Übungsblatt 11 - Aufgabe 2

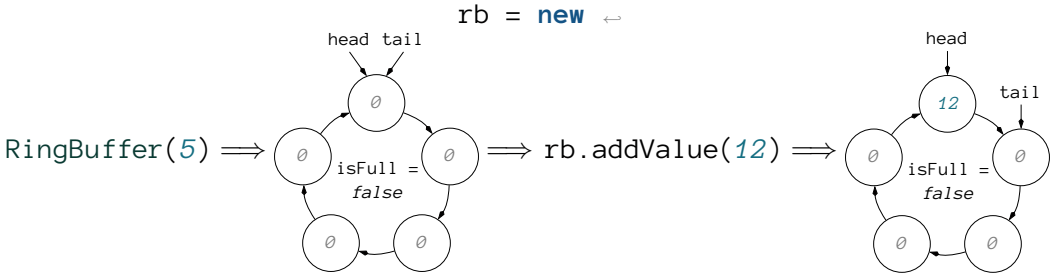
In dieser Aufgabe sollen Sie einen „Ringbuffer“ implementieren. Hierbei handelt es sich um eine dynamische Datenstruktur ähnlich zu einer Queue, jedoch verweist hier das letzte Element der Queue wieder auf das Erste. Zusätzlich hat ein Ringbuffer eine fest vorgegebene Kapazität. Solch eine Datenstruktur findet zum Beispiel Anwendung in Szenarien wo es einen Produzenten gibt der Werte liefert, und einen Konsumenten, der diese Werte verarbeitet. Hierfür werden intern die zwei Zeiger `head` und `tail` verwendet. Alle Positionen bis zum `tail` Zeiger sind beginnend beim `head` Zeiger belegt. Falls die zwei Zeiger auf das selbe Element verweisen, so ist der Ringbuffer entweder komplett gefüllt oder aber komplett leer ist. Um diese zwei Fälle zu unterscheiden, benötigt man eine zusätzliche Zustandsvariable.
Um den Ringbuffer zu implementieren, definieren Sie sich zunächst einen Konstruktor, welcher die Kapazität des Ringbuffers als Parameter übernimmt und die Datenstruktur anlegt. Implementieren Sie basierend darauf nun die folgenden zwei Methoden:

- Die Methode `public void addValue(int value)` soll den übergebenen Wert an der Position im Ringbuffer speichern, auf die der Zeiger `tail` zeigt. Sollte der Buffer komplett gefüllt sein, soll eine `BufferOverflowException` geworfen werden.
- Die Methode `public int getValue()` soll den Wert an der Position zurückliefern, auf die der Zeiger `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` geworfen werden.

Verwenden Sie für ihre Implementierung keine Arrays oder vorgefertigten dynamischen Datenstrukturen, sondern implementieren Sie den Ringbuffer selbst als (einfach) verkettete Liste.

Übungsblatt 11 - Aufgabe 2

- Von uns wird ein einfacher Ringbuffer verlangt.
  - Idee: Eine veränderbare `Element`-Klasse.
  - Wir generieren einen Kreis mit gegebener Kapazität.
  - Wir rotieren mit `head`.



Übungsblatt 11 - Aufgabe 2 Grundkonstrukt

- Wir beginnen wieder mit der Queue-Idee (RingBuffer.java).

```
public class RingBuffer {
    class Element {
        public /* final */ int value; // Jetzt veränderbar
        public Element next = null;
        public Element(int value) { this.value = value; }
    }
    private Element head;
    private Element tail;
    private boolean isFull;
}
```

- Hier heißen Start und Ende eben `head` und `tail`.
- Weiter soll `Element` veränderbar sein.
- Anstelle der Länge speichern wir zunächst einfach mal, ob der Buffer voll ist.

Übungsblatt 11 - Aufgabe 2 — Grundkonstruktor

- Der Konstruktor übernimmt hier die Arbeit, eine „leere“ Ringliste zu konstruieren.
- Der naive Ansatz:

```
public class RingBuffer {
    class Element { int value; Element next = null; }
    private Element head, tail;
    private boolean isFull;
    public RingBuffer(int capacity) {
        this.isFull = capacity <= 0;
        if(this.isFull) return;

        Element current = new Element(0); // Magic-Number-Standard
        this.head = this.tail = current;

        // TODO: füllen
    }
}
```

Übungsblatt 11 - Aufgabe 2 — Grundkonstruktor, II

```
public class RingBuffer {
    class Element { int value; Element next = null; }
    private Element head, tail;
    private boolean isFull;
    public RingBuffer(int capacity) {
        // ...
        Element current;

        for(int i = 1; i < capacity; i++) {
            Element next = new Element(0);
            current.next = next;

            if(i == capacity - 1) // Verbindung des letztes Elementes
                next.next = this.head;

            current = next;
        }
    }
}
```

Übungsblatt 11 - Aufgabe 2

Pump It

```
public class RingBuffer {
    class Element { int value; Element next = null; }
    private Element head, tail;
    private boolean isFull;
    public RingBuffer(int capacity) { ... }

    public void addValue(int value) {
        if ((this.head == this.tail) && this.isFull) // sind wir wirklich voll?
            throw new BufferOverflowException(); // java.nio!

        this.tail.value = value;
        this.tail = this.tail.next;

        // was wäre mit equals?
        if (this.tail == this.head)
            this.isFull = true;
    }
}
```

Objects.equals(tail, head) wäre keine gute Idee. Es funktioniert aus zwei Gründen: 1) *Objects*::equals handhabt ein tail = **null**, welches aber nur bei capacity = *0* auftritt (es ist ja sonst ein „Ring“) und 2) wir haben es nicht für *Element* implementiert. Die Standardimplementation ist einfach mit ==. Dennoch: Wir wollen hier die Identität und *nicht* die Gleichheit prüfen. Das ist ein Fall, in dem == viel besser ist.

Übungsblatt 11 - Aufgabe 2

Take It

```
public class RingBuffer {
    class Element { int value; Element next = null; }
    private Element head, tail;
    private boolean isFull;
    public RingBuffer(int capacity) { ... }
    public void addValue(int value) { ... }
    public int getValue() {
        if ((this.head == this.tail) && !this.isFull) // sind wir leer?
            throw new NoSuchElementException();

        this.isFull = false;
        int value = this.head.value;
        this.head = this.head.next; // Nicht direkt gefordert, aber logisch :D

        return value;
    }
}
```

Übungsblatt 11 - Aufgabe 2 Thats It

- Ein paar Ideen, die das Ganze interessanter machen.
- Anstelle einfach `this.tail.value = value` zu setzen, kann man probieren immer neue `Element`-Objekte zu erzeugen und einzufügen und wieder zu entfernen.
- Erlauben, dass die Kapazität auch nach der Erstellung verändert werden kann (vergrößern und verkleinern).
- Wenn der Ringbuffer voll ist, soll das älteste `Element` durch das neue ersetzt werden.

Übungsblatt 11 - Zusatzaufgabe 5/6 — Shake It

In dieser Aufgabe sollen Sie noch einmal einen Sortieralgorithmus selbst implementieren. Dieses Mal handelt es sich um den (Cocktail) Shaker Sort, welcher eine Variation des Bubblesort Algorithmus darstellt. Hierbei wird das Array in jedem Sortierschritt alternierend von vorne und hinten durchlaufen, wobei die bereits sortieren Elemente vorne und hinten nicht mehr betrachtet werden. Auch diesen Algorithmus sollen Sie wieder rekursiv implementieren. Wenn Sie nicht wissen wie Sie beginnen sollen, versuchen Sie den Algorithmus zuerst iterativ zu implementieren und wandeln Sie anschließend die Schleifen in rekursive Methodenaufrufe um. Auch bei dieser Aufgaben soll ihre fertige Implementierung des Algorithmus keine Schleifen und keine vorgefertigten Sortierfunktion (wie z.B. `Arrays.sort()`) enthalten.

Übungsblatt 11 - Zusatzaufgabe 5/6 Bubblesort

```java
static void bubbleSort(int[] arr) {
    for (int n = arr.length; n > 1; n--) {
        for (int i = 0; i < n - 1; i++) {
            if (arr[i] > arr[i + 1])
                swap(arr, i, i + 1);
        }
    }
}
```

```java
static void bubbleSort(int[] arr, int n) {
    if (n > 1) {
        move(arr, 0, n);
        bubbleSort(arr, n - 1);
    }
}

static void move(int[] arr, int i, int n) {
    if (i < n - 1) {
        if (arr[i] > arr[i + 1])
            swap(arr, i, i + 1);
        move(arr, i + 1, n);
    }
}
```

Show me the moves (ShakerSort.java)

Übungsblatt 11 - Zusatzaufgabe 5/6 — Ein Blick auf move

```java
static void moveUp(int[] array, int index, int end) {
    if (index < end) { // Exemplarisch mit dem unteren
        if (array[index] < array[index - 1])
            swap(array, index, index - 1);
        return moveUp(array, index + 1, end);
    }
}

static void moveDown(int[] array, int index, int end) {
    if (index > end) {
        if (array[index] < array[index - 1])
            swap(array, index, index - 1);
        return moveDown(array, index - 1, end);
    }
}
```

```java
static void move(int[] arr, int i, int n) {
  if (i < n - 1) {
    if (arr[i] > arr[i + 1])
      swap(arr, i, i + 1);
    move(arr, i + 1, n);
  }
}
```

Übungsblatt 11 - Zusatzaufgabe 5/6 Ping Pong

```java
static void shakerSort(int[] array, int n) {
    int offset = array.length - n;
    if (n > array.length / 2) { // perförmänce büüst
        moveUp(array, offset + 1, n);
        moveDown(array, n - 1, offset);

        shakerSort(array, n - 1);
    }
}
```

```java
static void bubbleSort(int[] arr, int n) {
  if (n > 1) {
    move(arr, 0, n);
    bubbleSort(arr, n - 1);
  }
}
```

Übungsblatt 11 - Zusatzaufgabe 5/6 Effizienter

```
static boolean moveUp(int[] array, int index, int end, boolean swapped) {
    if (index < end) {
        if (array[index] < array[index - 1]) {
            swap(array, index, index - 1);
            swapped = true;
        }
        return moveUp(array, index + 1, end, swapped);
    }
    return swapped;
}

static boolean moveDown(int[] array, int index, int end, boolean swapped) {
    if (index > end) {
        if (array[index] < array[index -1]) {
            swap(array, index, index - 1);
            swapped = true;
        }
        return moveDown(array, index - 1, end, swapped);
    }
    return swapped;
}
```

Übungsblatt 11 - Zusatzaufgabe 5/6 Effizienter, II

```
public static void shakerSort(int[] array, int n) {
    int offset = array.length - n;
    if (n > array.length / 2) {
        if(!moveUp(array, offset + 1, n, false)) return;
        if(!moveDown(array, n - 1, offset, false)) return;

        shakerSort(array, n - 1);
    }
}
```


Li(s)t!