252-0027

Einführung in die Programmierung

Übungen

Woche 10: Verlinkte Objekte, Klassen

Jonas Wetzel
Departement Informatik
ETH Zürich

Plan heute

- **Nachbesprechung**
  - Timed Bonus
- **Executable Graph Aufgabe**
- **Theorie**
  - Klassen, Vererbung
- **Prüfungsaufgaben (I cooked)**
- **Vorbesprechung**
- **Kahoot**

Organisatorisches

- **Website: n.ethz.ch/~jwetzel**
- **Whatsapp Gruppe**
- **Gute Summary von Zoe**
  - Siehe pdf
- **Website von Gohar (super TA)**
  - https://eprog23.wixstudio.io/hs23

Nachbesprechung

Aufgabe 1: Loop-Invariante

Gegeben ist eine Postcondition für das folgende Programm

```
public int compute(String s, char c) {
  // Precondition s != null
  int x;
  int n;

  x = 0;
  n = 0;

  // Loop Invariante:
  while (x < s.length()) {
    if (s.charAt(x) == c) {
      n = n + 1;
    }
    x = x + 1;
  }

  // Postcondition: count(s, c) == n
  return n;
}
```

Die Methode `count(String s, char c)` gibt zurück wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt". **Tipp:** Benutzen Sie die `substring` Methode.

Aufgabe 2: Linked List

Bisher haben Sie Arrays verwendet, wenn Sie mit einer grösseren Anzahl von Werten gearbeitet haben. Ein Nachteil von Arrays ist, dass die Grösse beim Erstellen des Arrays festgelegt werden muss und danach nicht mehr verändert werden kann. In dieser Aufgabe implementieren Sie selbst eine Datenstruktur, bei welcher die Grösse im Vornherein nicht bestimmt ist und welche jederzeit wachsen und schrumpfen kann: eine *linked list* oder *verkettete Liste*.

Eine verkettete Liste besteht aus mehreren Objekten, welche Referenzen zueinander haben. Für diese Aufgabe besteht jede Liste aus einem "Listen-Objekt" der Klasse `LinkedIntList`, welches die gesamte Liste repräsentiert, und aus mehreren "Knoten-Objekten" der Klasse `IntNode`, eines für jeden Wert in der Liste. Die Liste heisst "verkettet", weil jedes Knoten-Objekt ein Feld mit einer Referenz zum nächsten Knoten in der Liste enthält. Das `LinkedIntList`-Objekt schliesslich enthält eine Referenz zum ersten und zum letzten Knoten und hat ausserdem ein Feld für die Länge der Liste.

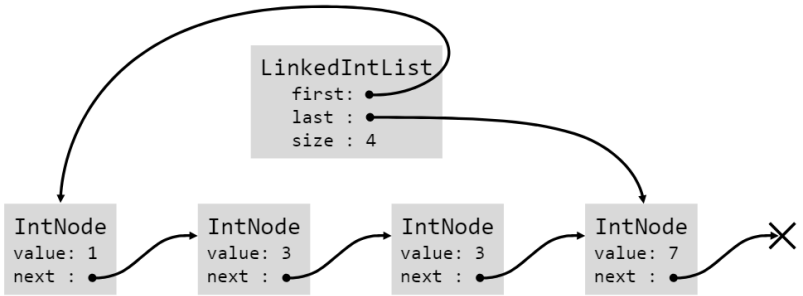


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: Linked List

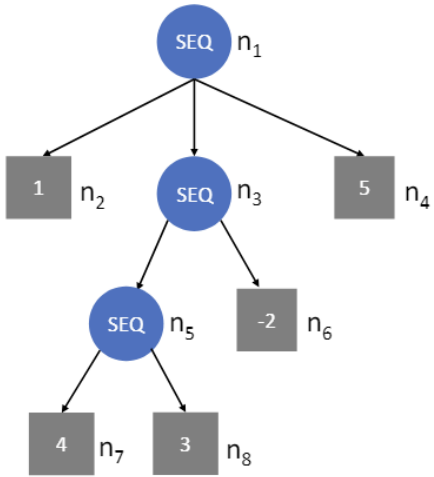
Name	Parameter	Rückg.-Typ	Beschreibung
`addLast`	`int value`	`void`	fügt einen Wert am Ende der Liste ein
`addFirst`	`int value`	`void`	fügt einen Wert am Anfang der Liste ein
`removeFirst`		`int`	entfernt den ersten Wert und gibt ihn zurück
`removeLast`		`int`	entfernt den letzten Wert und gibt ihn zurück
`clear`		`void`	entfernt alle Wert in der Liste
`isEmpty`		`boolean`	gibt zurück, ob die Liste leer ist
`get`	`int index`	`int`	gibt den Wert an der Stelle `index` zurück
`set`	`int index, int value`	`void`	ersetzt den Wert an der Stelle `index` mit `value`
`getSize`		`int`	gibt zurück, wie viele Werte die Liste enthält

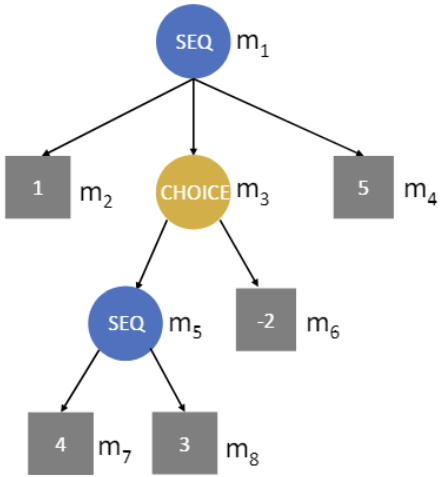
Einige dieser Methoden dürfen unter gewissen Bedingungen nicht aufgerufen werden. Zum Beispiel darf `removeFirst()` nicht aufgerufen werden, wenn die Liste leer ist, oder `get()` darf nicht aufgerufen werden, wenn der gegebene Index grösser oder gleich der aktuellen Länge der Liste ist. In solchen Situationen soll sich Ihr Programm mit einer Fehlermeldung beenden. Verwenden Sie folgendes Code-Stück dafür:

```
if(condition) {
    Errors.error(message);
}
```

Ersetzen Sie *`condition`* mit der Bedingung, unter welcher das Programm beendet werden soll, und *`message`* mit einer hilfreichen Fehlermeldung. Die `Errors`-Klasse befindet sich bereits in Ihrem Projekt, aber Sie brauchen sie im Moment nicht zu verstehen.

Aufgabe 3: Executable Graph





Sehen wir gleich in Detail

Aufgabe 4: Energiespiel

In dieser Aufgabe üben Sie den Umgang mit Enums. Dafür haben Sie einen Ordner `EnergieSpiel` mit drei Klassen `GameApp`, `Game` und `Player`, sowie ein Enum `Character`. Diese sind bereits so implementiert, dass alles funktioniert. Die Klasse `Player` hat jedoch ein Feld `character` von Typ `String`. Java lässt also zu, dass in diesem Feld ein beliebiger String abgespeichert werden kann. Das Spiel hat aber eigentlich nur genau drei Möglichkeiten: `HONEST`, `TRICKSTER` oder `SORCEROR`. Das Enum `Character` mit diesen drei Optionen existiert bereits. Ändern Sie den Typ des Feldes zu `Character` und passen Sie den Code in allen drei Klassen so an, dass die Charakter-Logik überall den Typ `Character` statt `String` verwendet.

Aufgabe 5: Timed Bonus

Die Bonusaufgabe für diese Übung wird erst am Dienstag Abend der Folgewoche (also am 19. 11.) um 17:00 Uhr publiziert und Sie haben dann 2 Stunden Zeit, diese Aufgabe zu lösen. Der Abgabetermin für die anderen Aufgaben ist wie gewohnt am Dienstag Abend um 23:59. Bitte planen Sie Ihre Zeit entsprechend.
Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. Importieren Sie das Eclipse-Projekt wie bisher.

Solving Timed Bonus

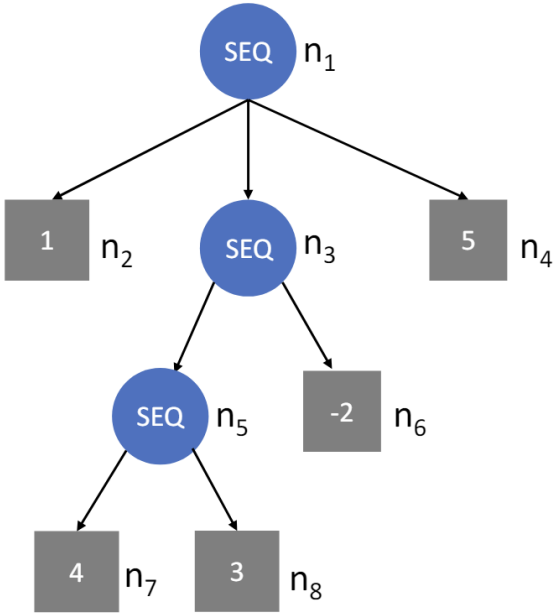
- **See pdf + code**

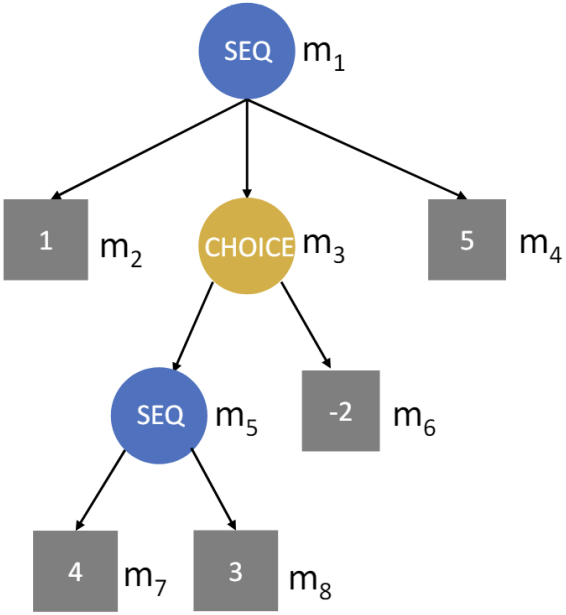
Advice

- **Sie geben immer wie „Teilaufgaben“, aber meistens macht die Aufgabe mehr Sinn, wenn man direkt alles betrachtet**
- **Weil wenn ihr es in Teilaufgaben löst müsst ihr meistens bei der zweiten alles wieder umändern 😥**

Probleme Lösen

Probleme Lösen: Executable Graph





Executable Graph

- **Siehe PDF mit Aufgabenstellung**

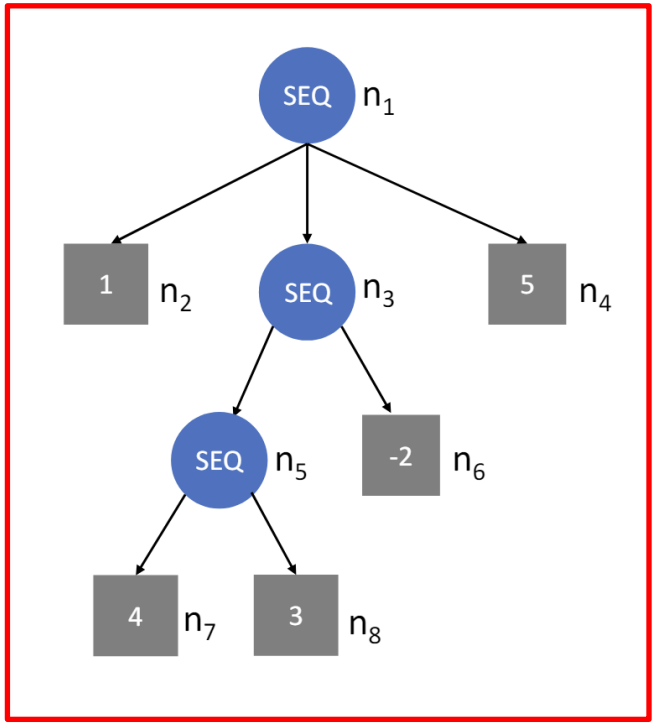
Probleme Lösen: Executable Graph

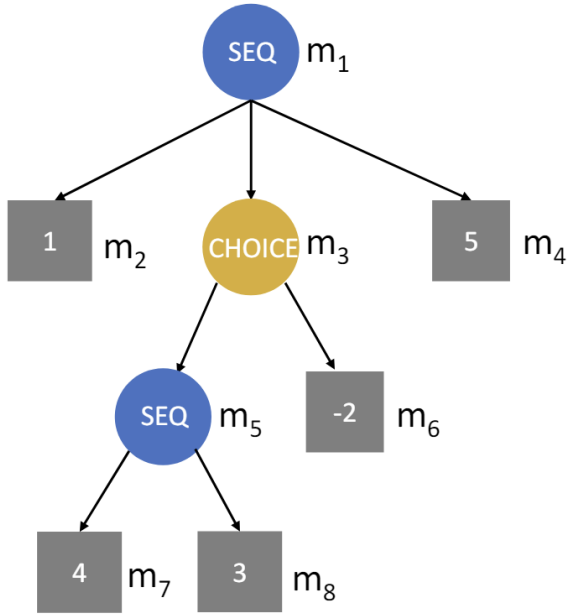
- **ADD Node:** Enthalten `value a` und wir setzen `(sum, counter)` auf `(sum + a, counter + 1)`. Kinderknoten werden bei der Ausführung ignoriert.

- **SEQ Node:** Kinderknoten werden nacheinander ausgeführt. Reihenfolge ist egal. Das `value` Attribut wird ignoriert.

- **CHOICE Node:** Ein beliebiger Knoten wird ausgeführt. Das `value` Attribut wird ignoriert.

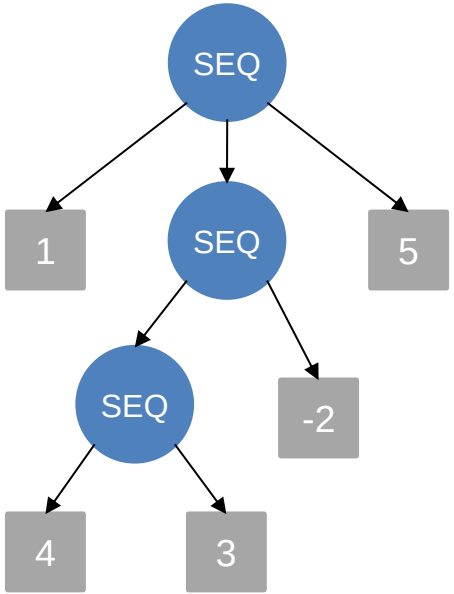
Probleme Lösen: Executable Graph





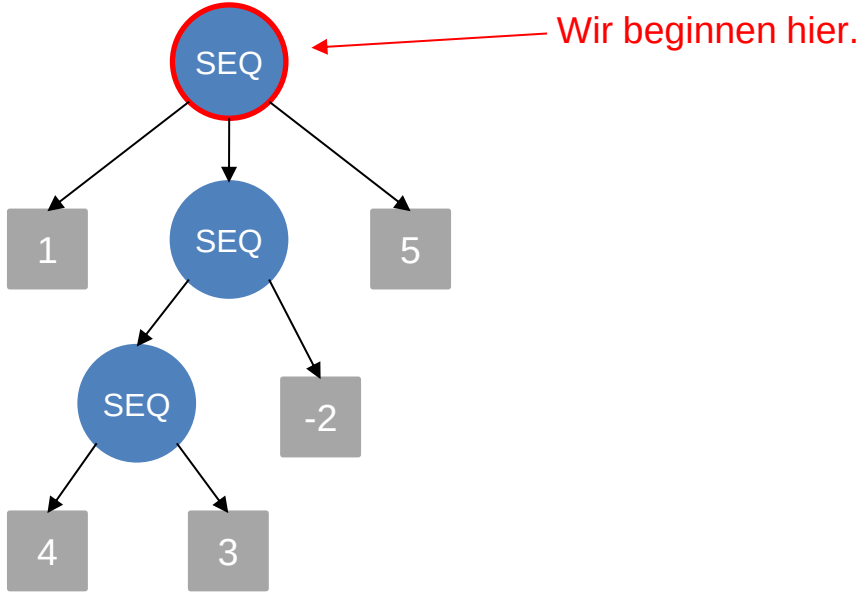
Probleme Lösen: Executable Graph

Startzustand: `(1,2)`

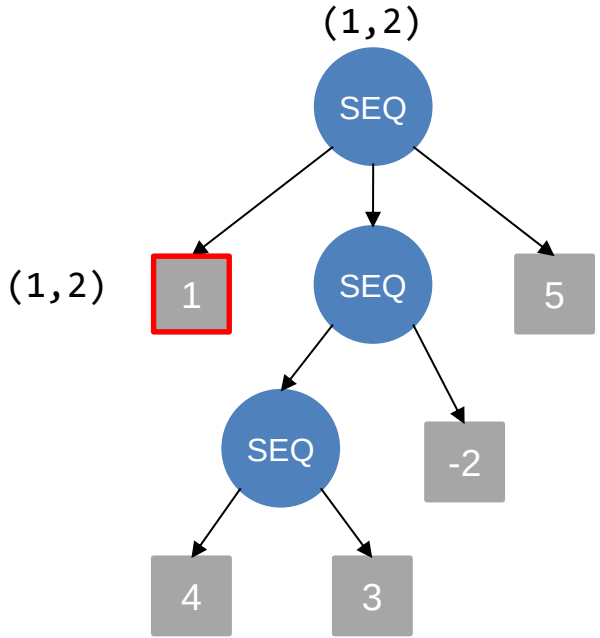


Probleme Lösen: Executable Graph

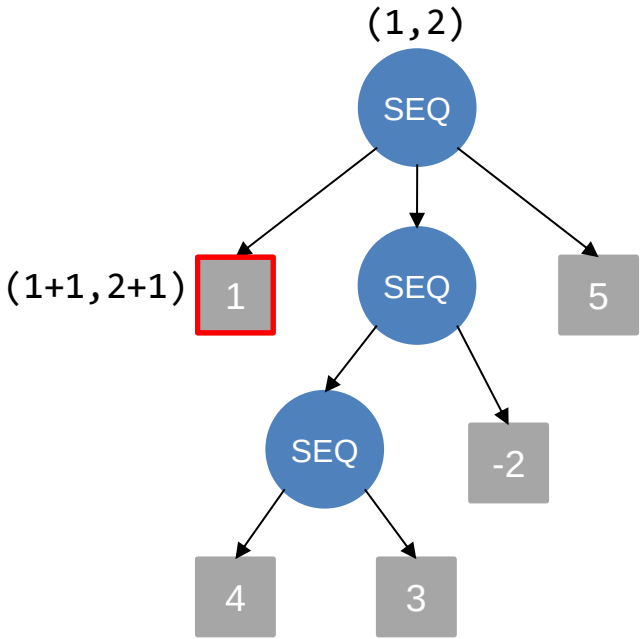
Startzustand: `(1,2)`



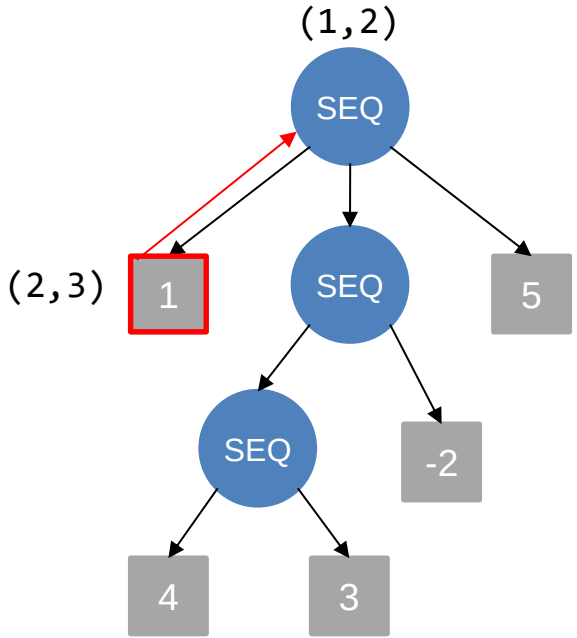
Probleme Lösen: Executable Graph



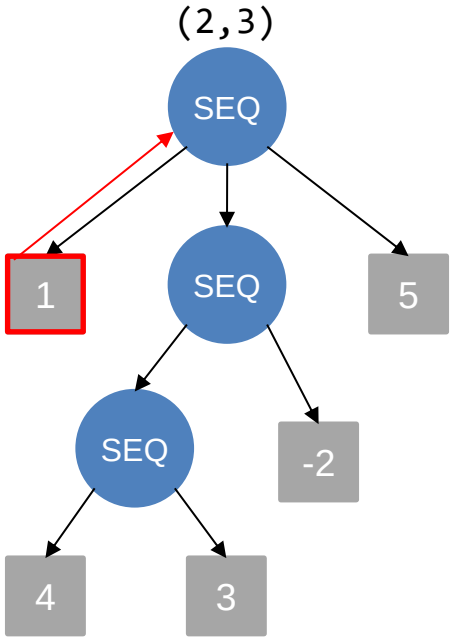
Probleme Lösen: Executable Graph



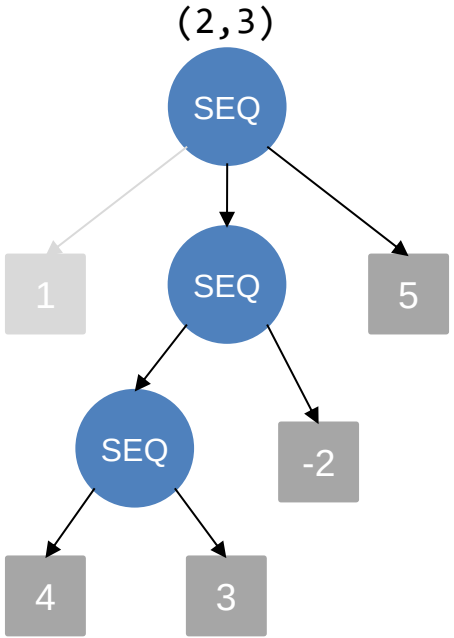
Probleme Lösen: Executable Graph



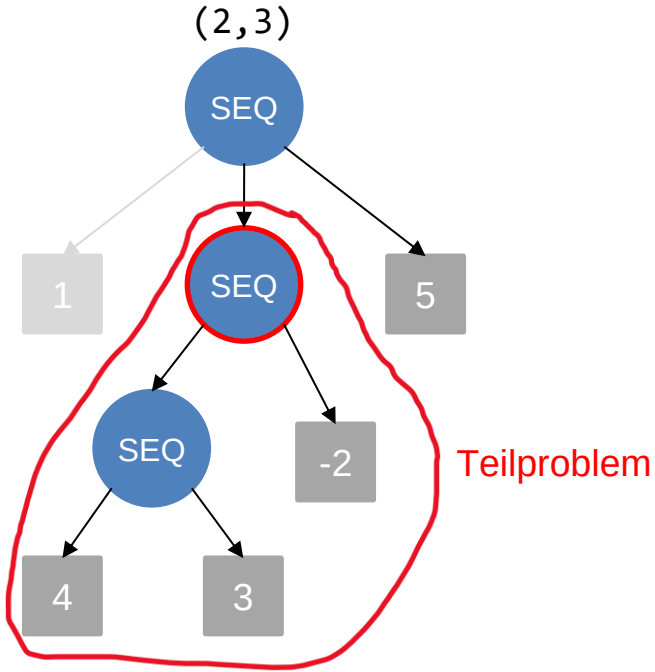
Probleme Lösen: Executable Graph



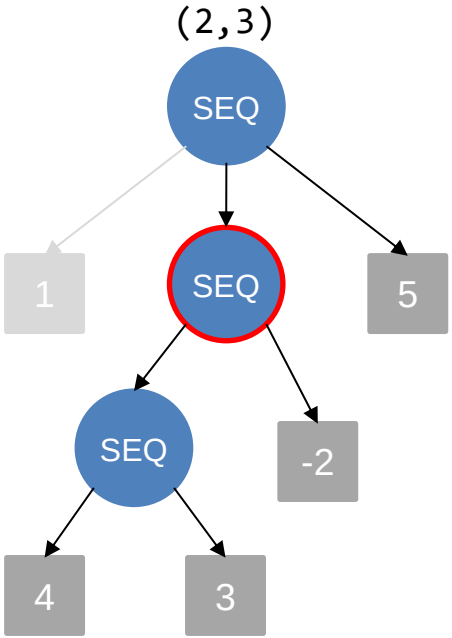
Probleme Lösen: Executable Graph

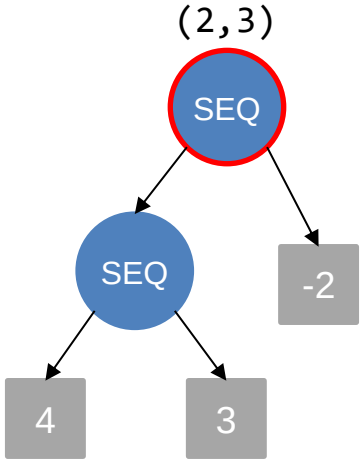


Probleme Lösen: Executable Graph

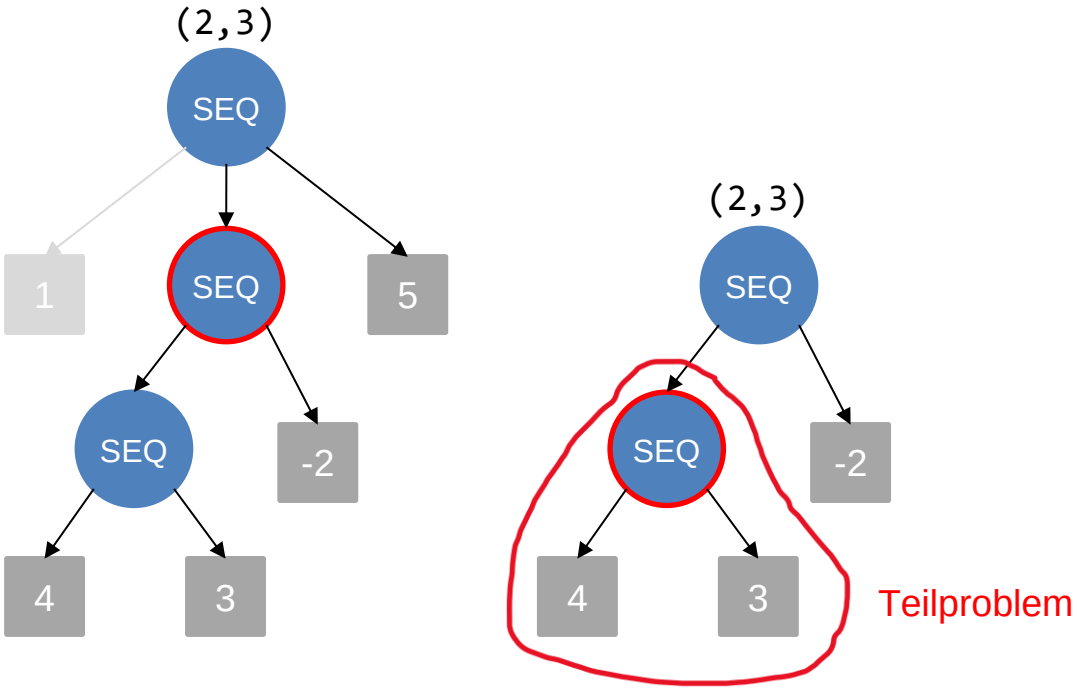


Probleme Lösen: Executable Graph

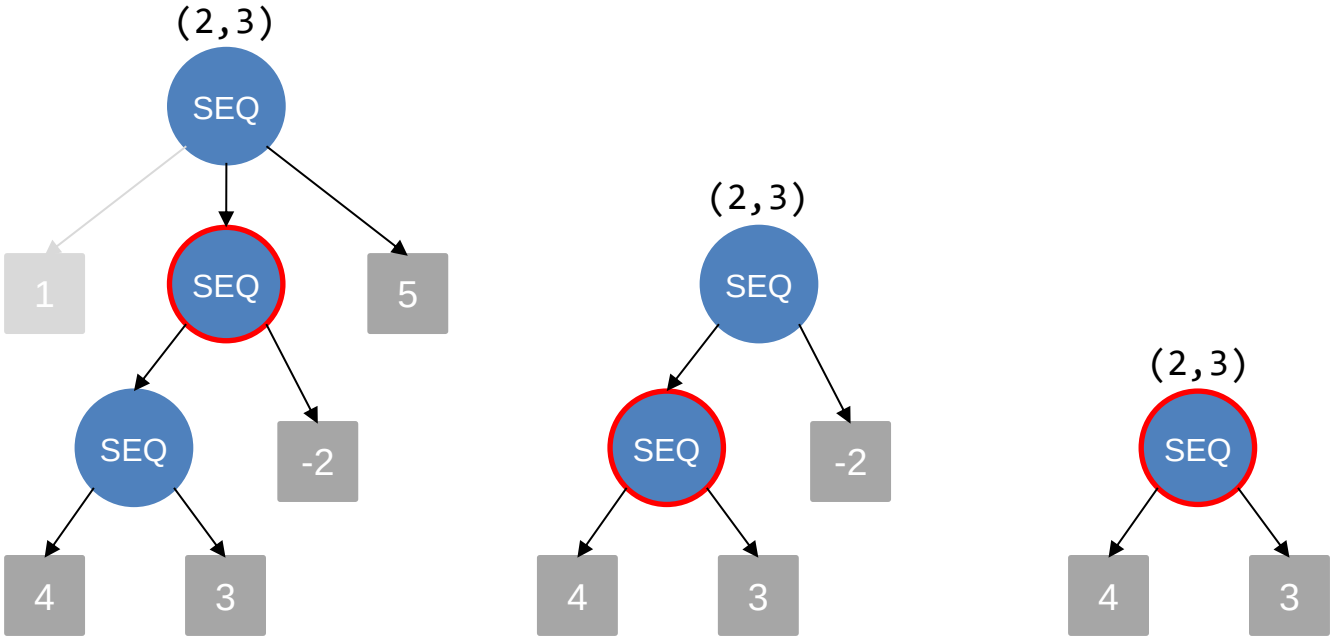




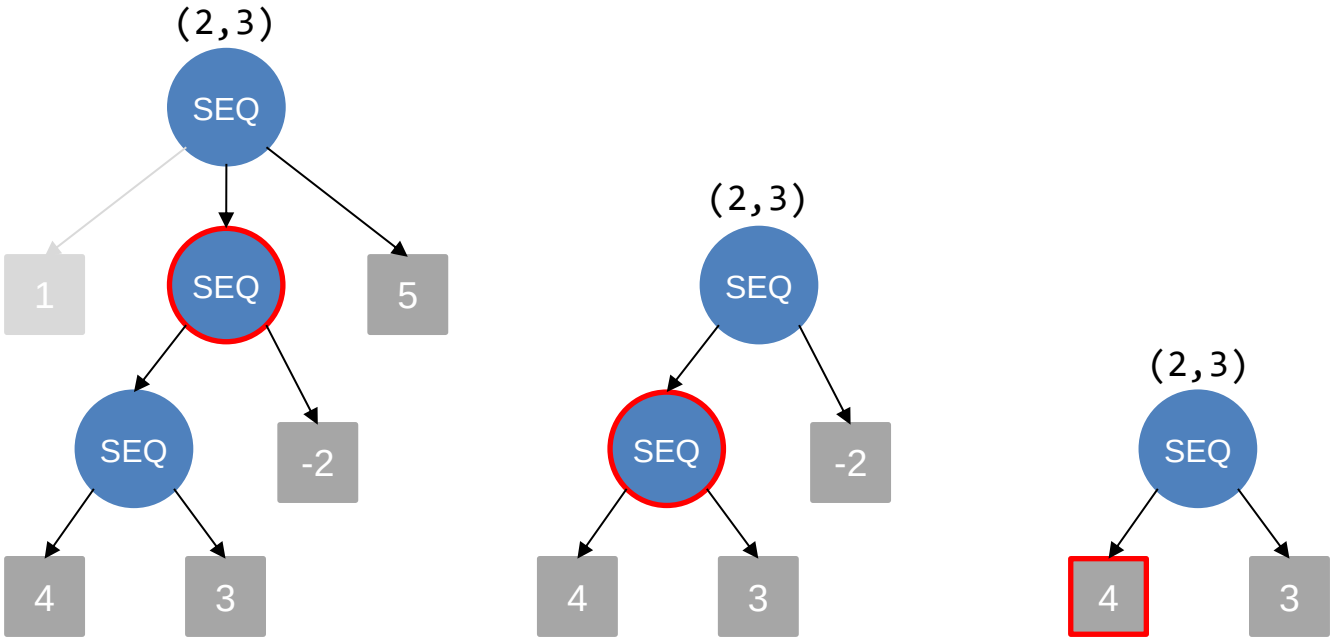
Probleme Lösen: Executable Graph



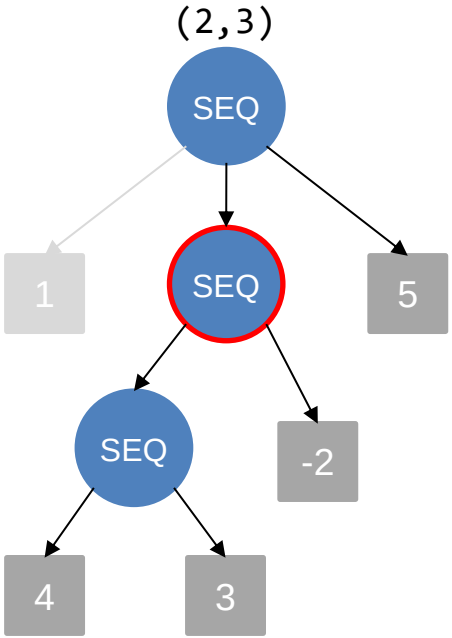
Probleme Lösen: Executable Graph

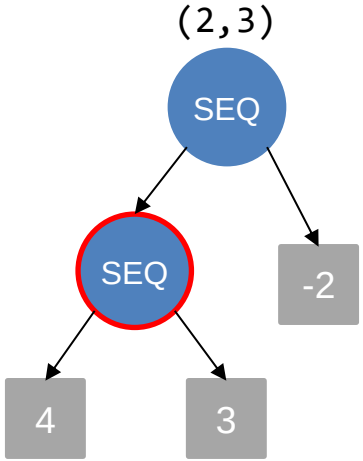


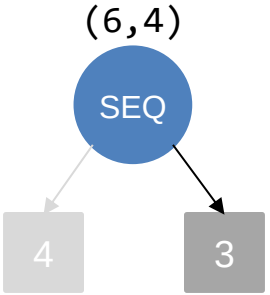
Probleme Lösen: Executable Graph



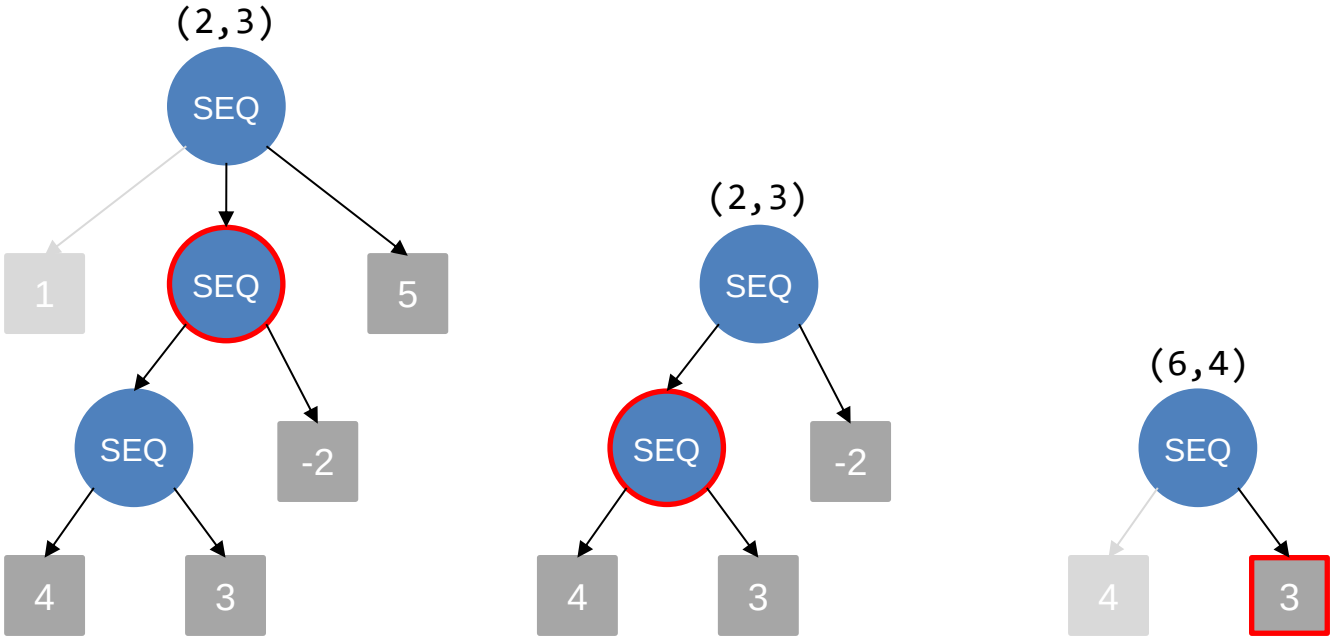
Probleme Lösen: Executable Graph



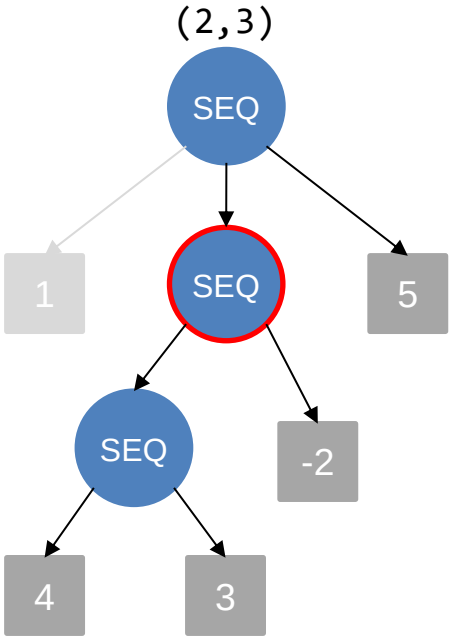


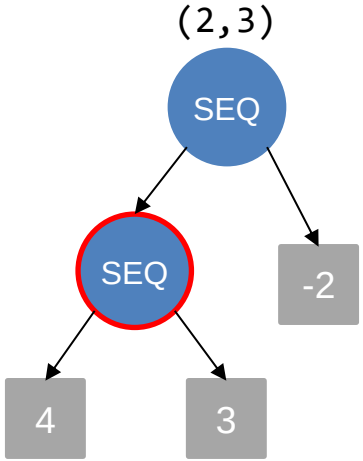


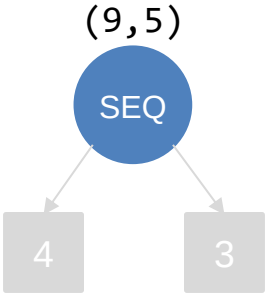
Probleme Lösen: Executable Graph



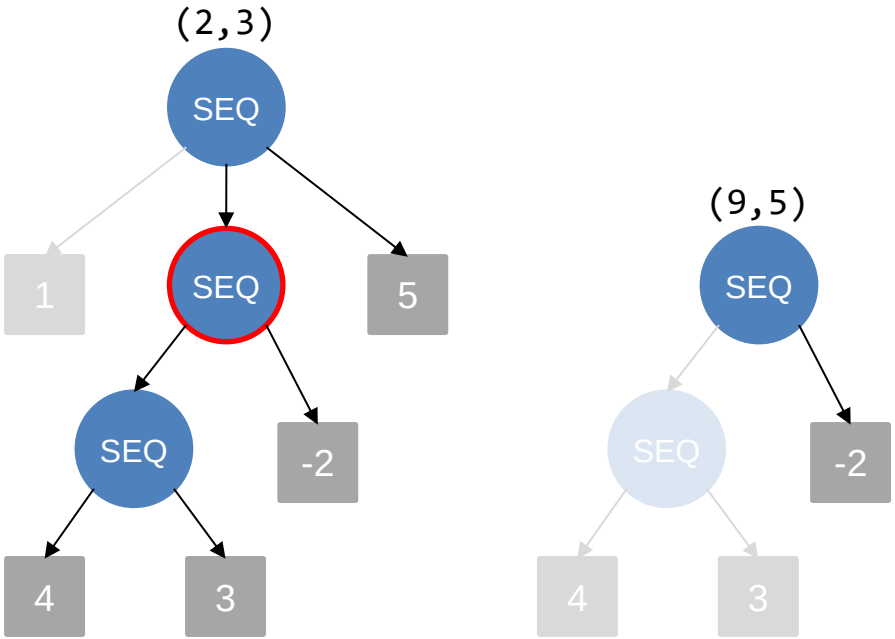
Probleme Lösen: Executable Graph



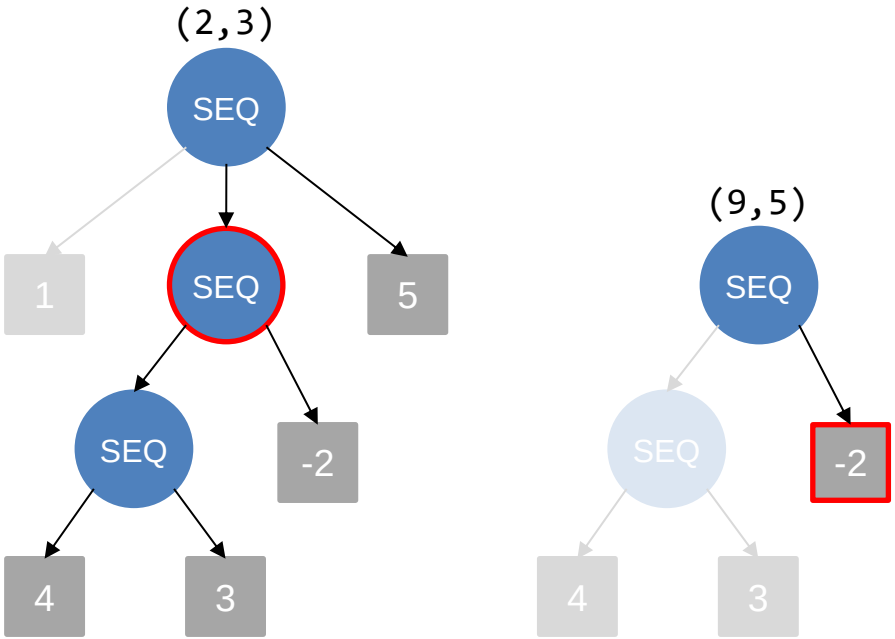




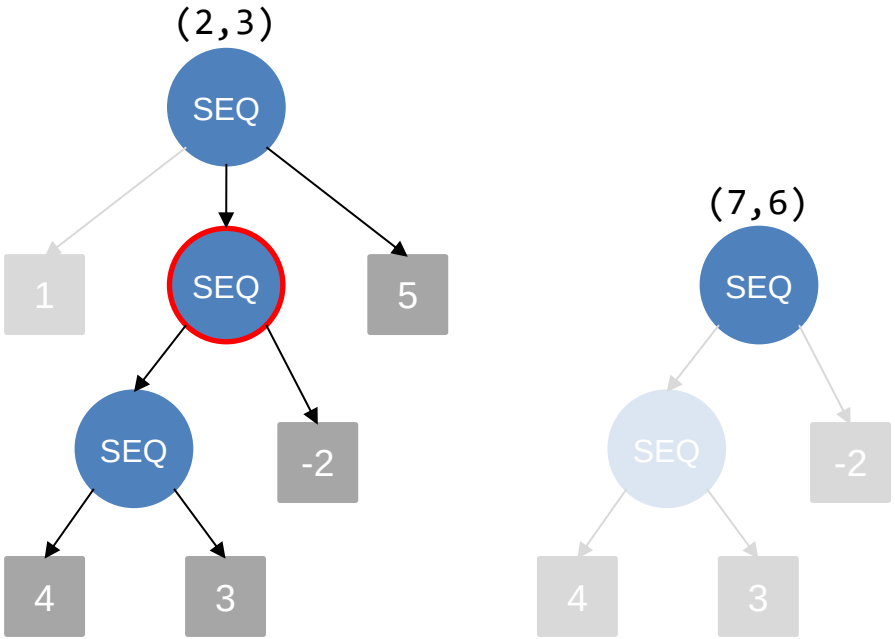
Probleme Lösen: Executable Graph



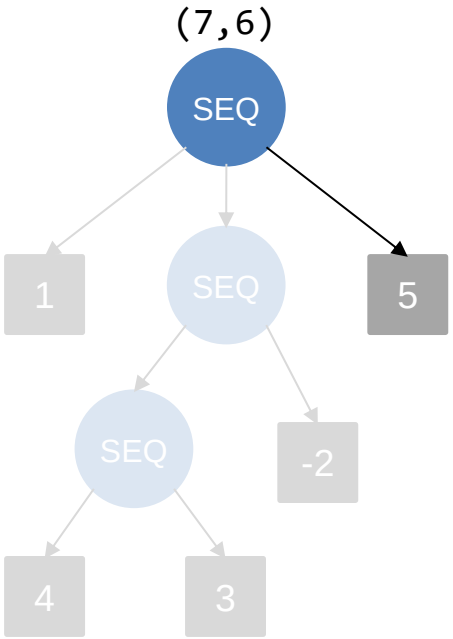
Probleme Lösen: Executable Graph



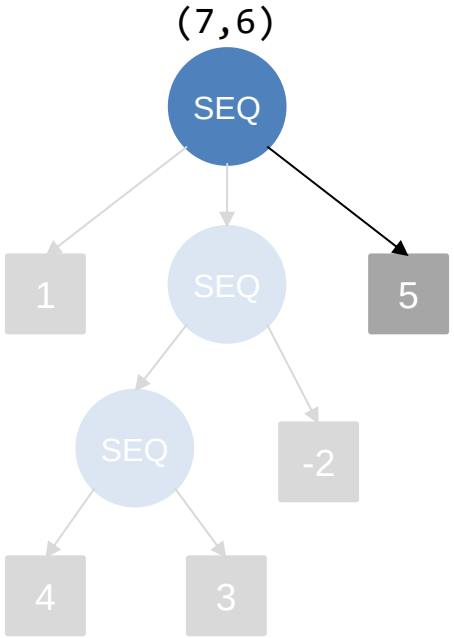
Probleme Lösen: Executable Graph



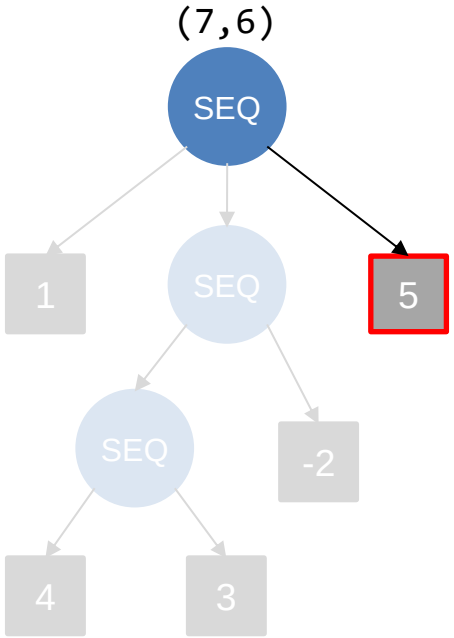
Probleme Lösen: Executable Graph



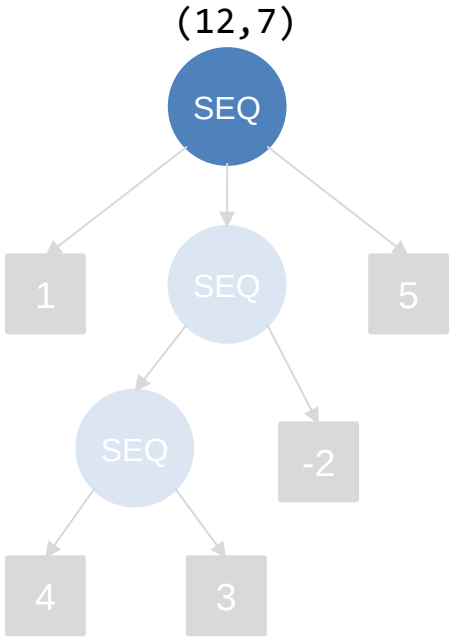
Probleme Lösen: Executable Graph



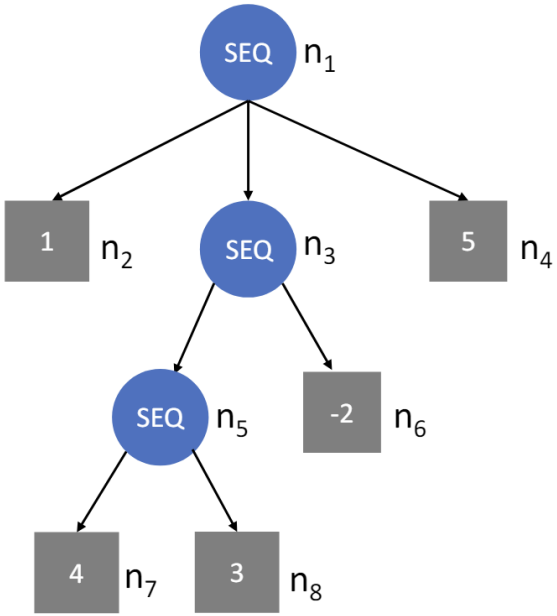
Probleme Lösen: Executable Graph

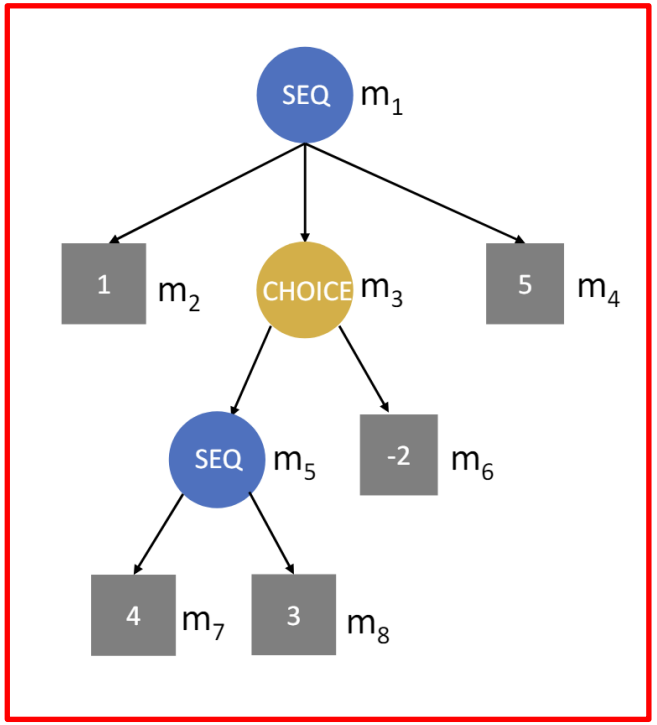


Probleme Lösen: Executable Graph



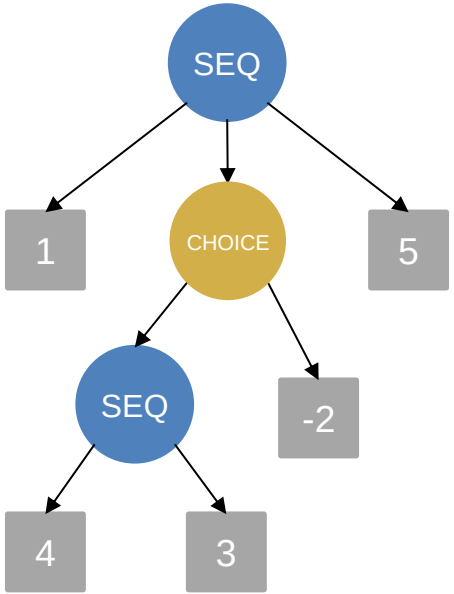
Probleme Lösen: Executable Graph





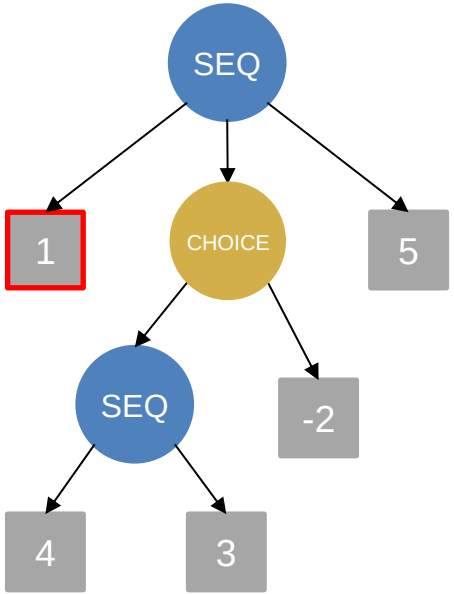
Probleme Lösen: Executable Graph

Startzustand: `(0,0)`



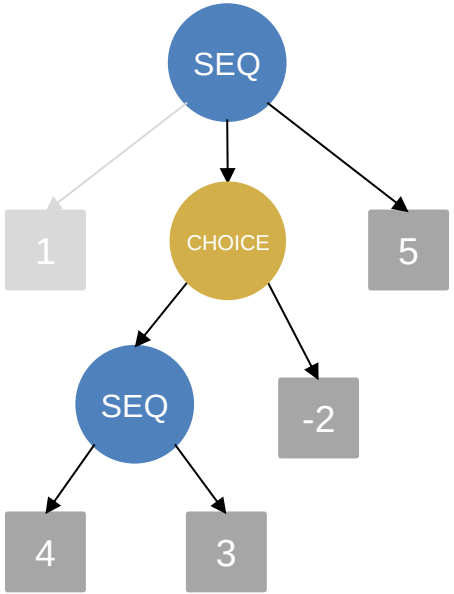
Probleme Lösen: Executable Graph

Startzustand: `(0,0)`



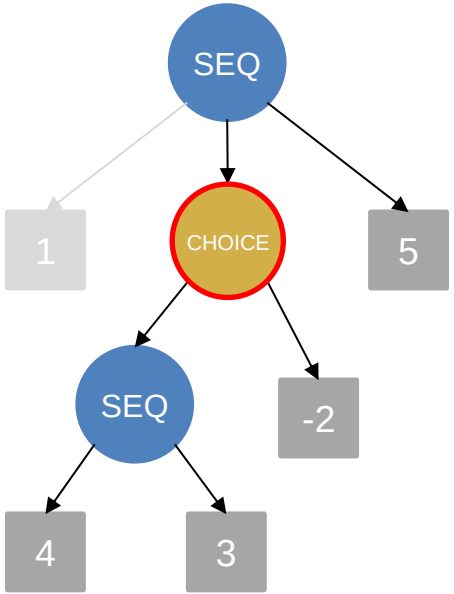
Probleme Lösen: Executable Graph

Startzustand: `(1,1)`



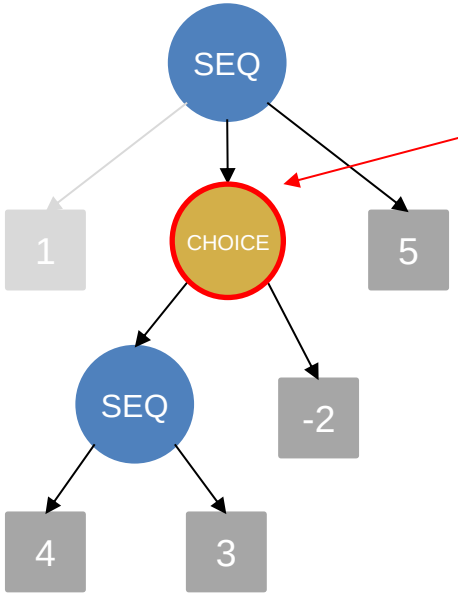
Probleme Lösen: Executable Graph

Startzustand: `(1,1)`



Probleme Lösen: Executable Graph

Startzustand: `(1,1)`

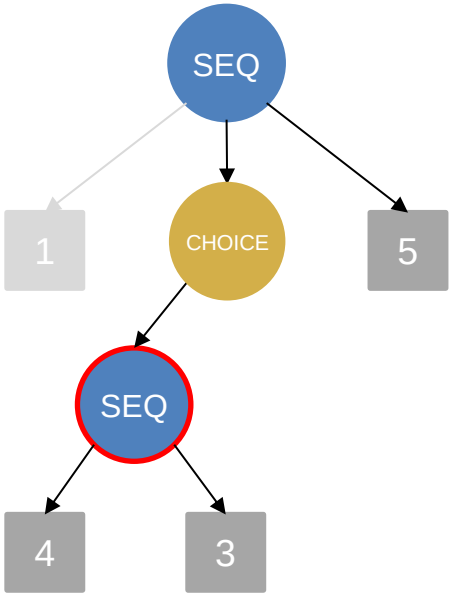


Wir haben jetzt die Wahl zwischen zwei Kinderknoten.

Probleme Lösen: Executable Graph

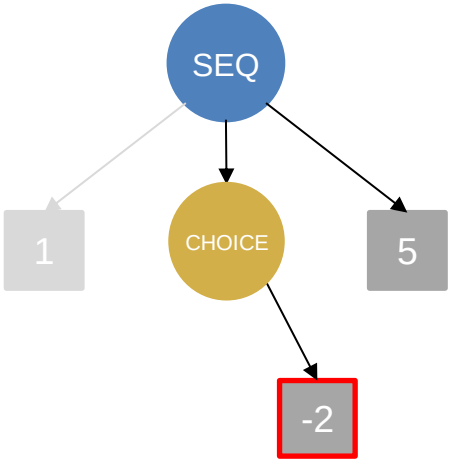
Resultat 1

(1,1)



Resultat 2

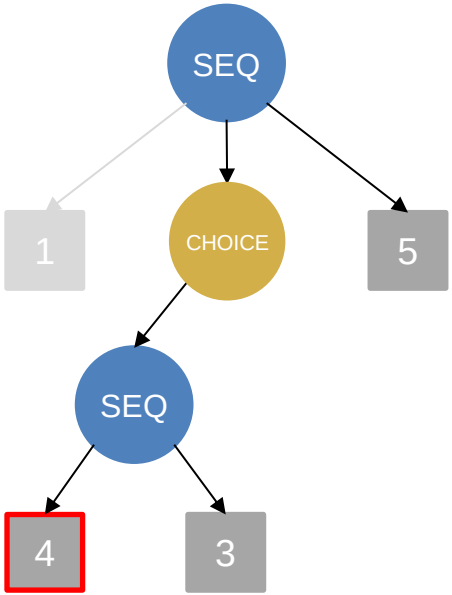
(1,1)



Probleme Lösen: Executable Graph

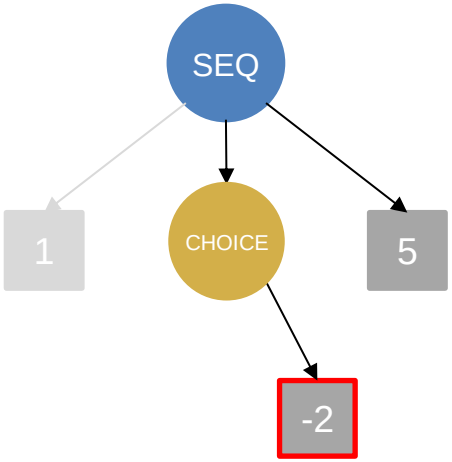
Resultat 1

(1,1)



Resultat 2

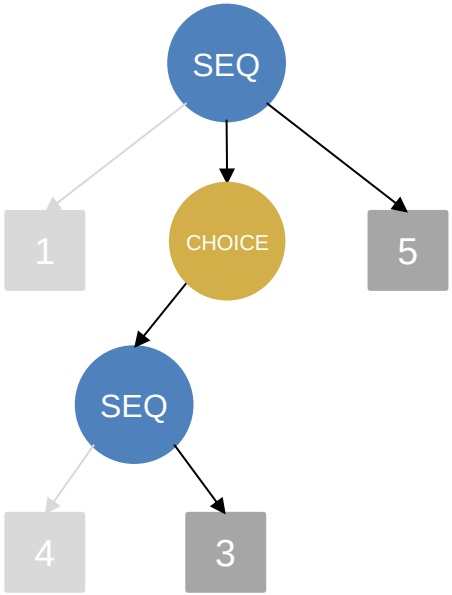
(1,1)



Probleme Lösen: Executable Graph

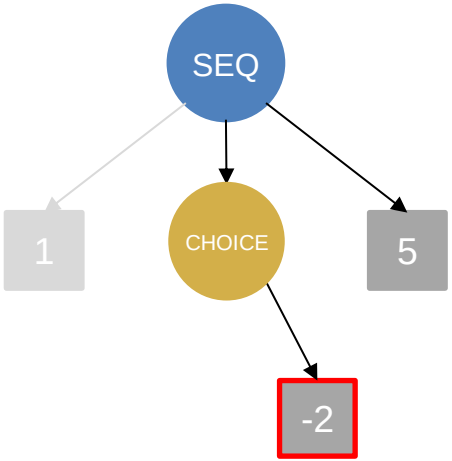
Resultat 1

(5,2)



Resultat 2

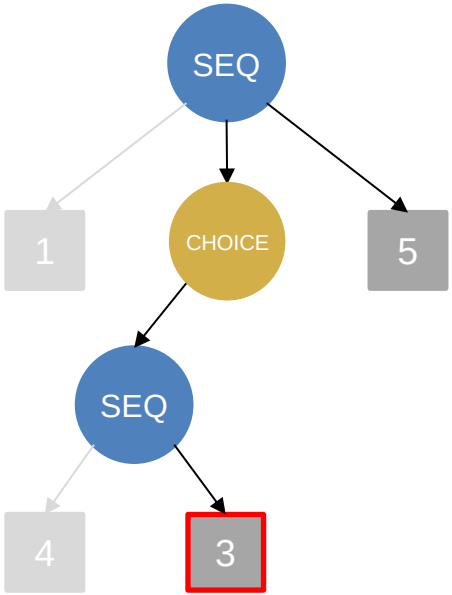
(1,1)



Probleme Lösen: Executable Graph

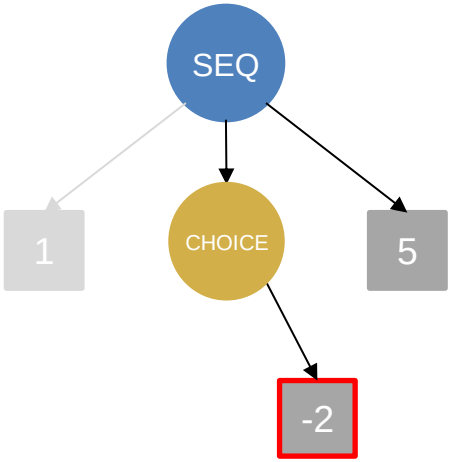
Resultat 1

(5,2)



Resultat 2

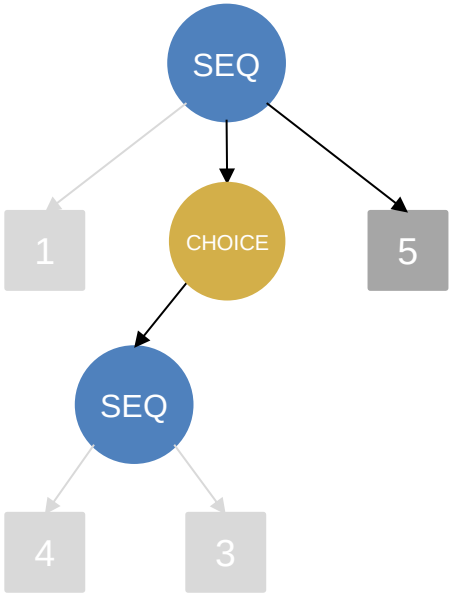
(1,1)



Probleme Lösen: Executable Graph

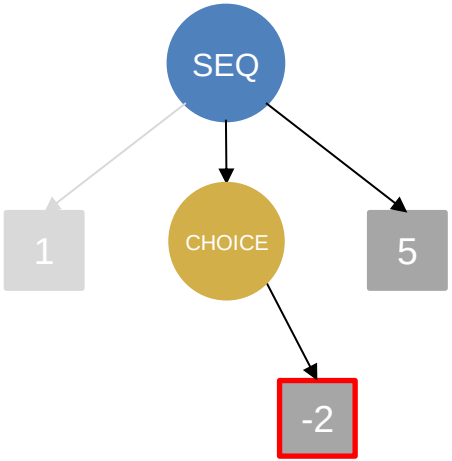
Resultat 1

(8,3)



Resultat 2

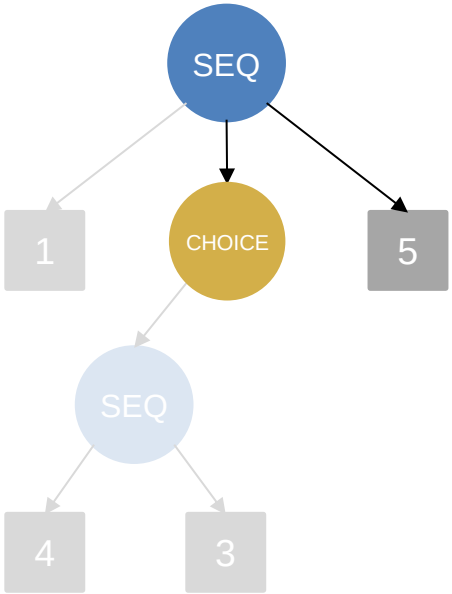
(1,1)



Probleme Lösen: Executable Graph

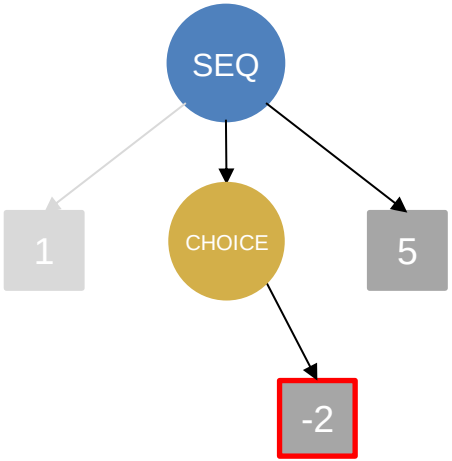
Resultat 1

(8,3)



Resultat 2

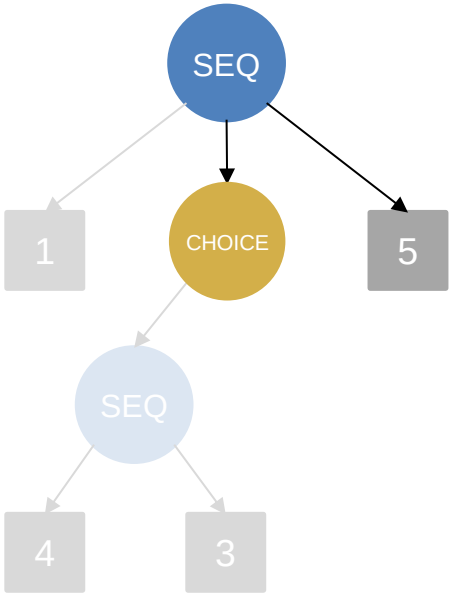
(1,1)



Probleme Lösen: Executable Graph

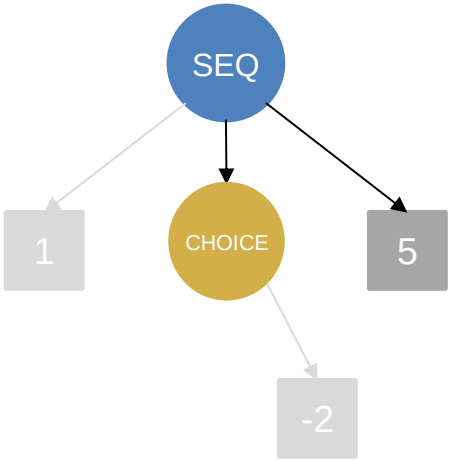
Resultat 1

(8,3)



Resultat 2

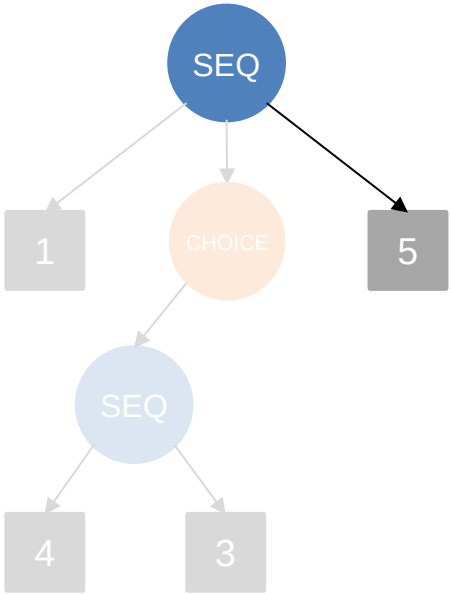
(-1,2)



Probleme Lösen: Executable Graph

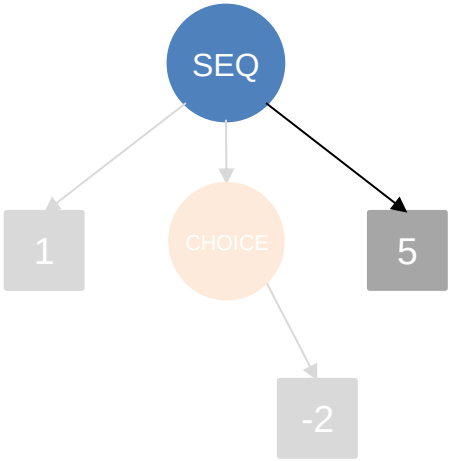
Resultat 1

(8,3)

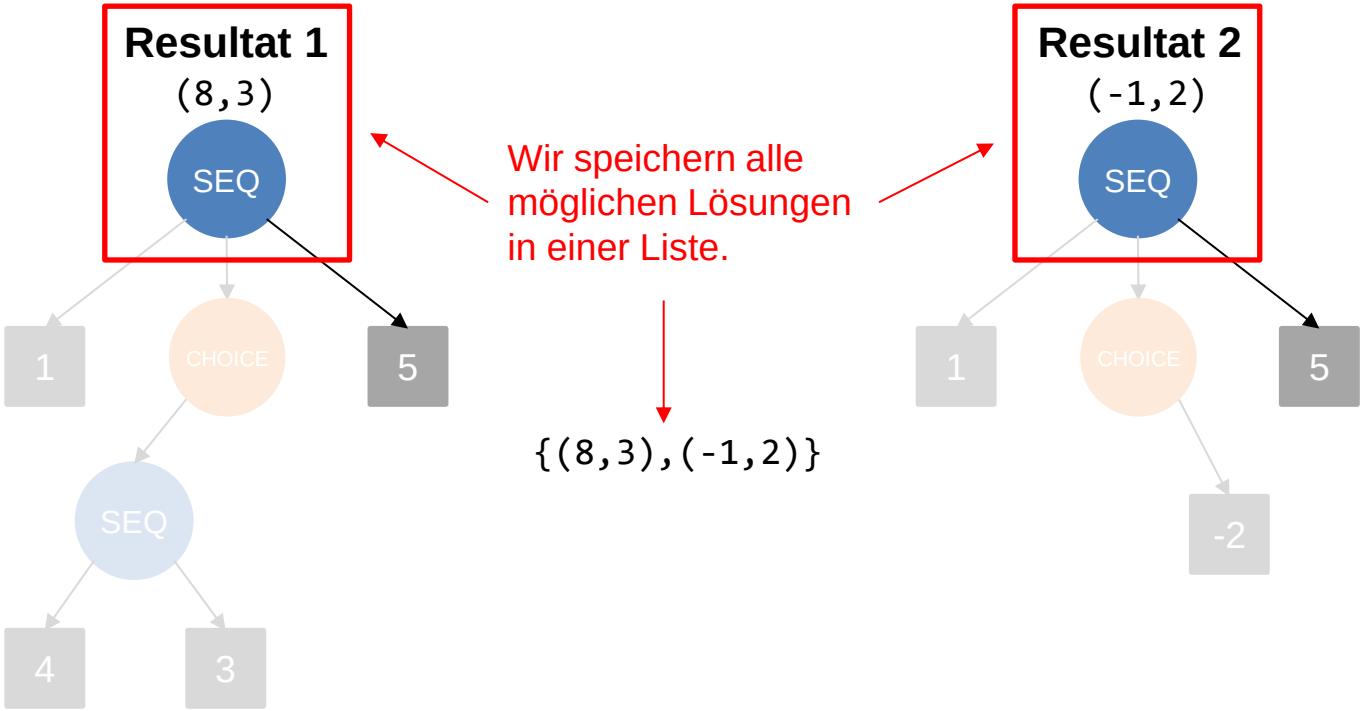


Resultat 2

(-1,2)

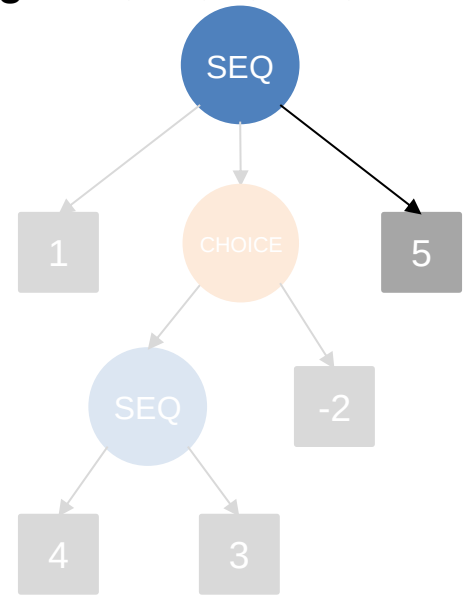


Probleme Lösen: Executable Graph



Probleme Lösen: Executable Graph

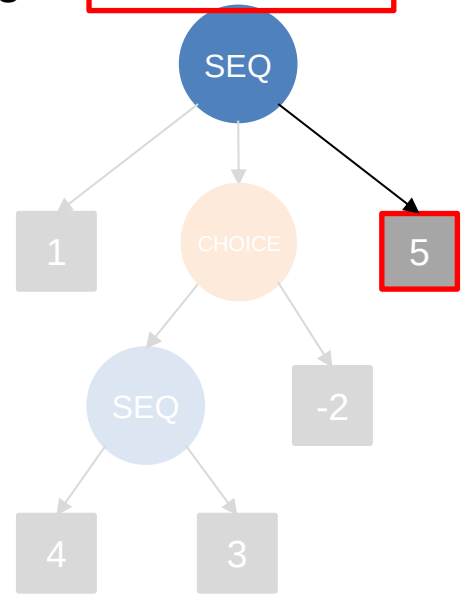
Lösungen: `{(8,3),(-1,2)}`



Probleme Lösen: Executable Graph

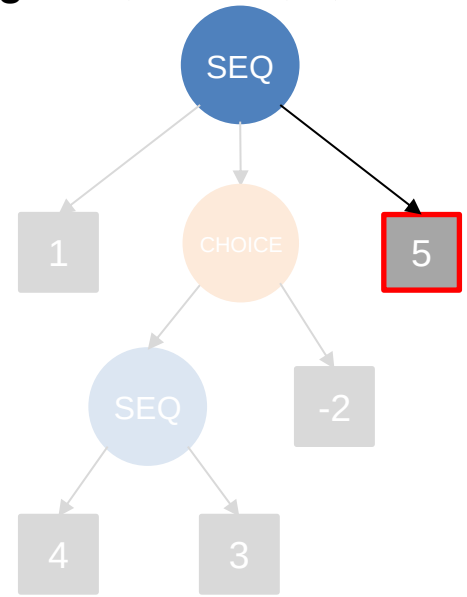
Lösungen: `{(8,3),(-1,2)}`

Wir müssen jedes Zwischenresultat in der Liste updaten.



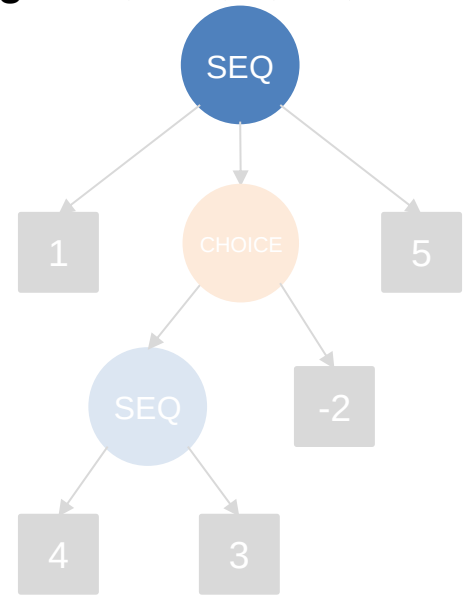
Probleme Lösen: Executable Graph

Lösungen: {(8+5,3+1),(-1+5,2+1)}



Probleme Lösen: Executable Graph

Lösungen: `{(13,4),(4,3)}`



Wie lösen wir das Problem?

- Wir nutzen eine **Helfermethode** `allResultsGo` welche statt nur einem Programmstate eine Liste von Programmstates als Parameter hat.

- **ADD:** Für alle Zwischenresultate addiere `value` dazu, erhöhe den Counter um `1` und füge das Resultat zu `next` hinzu.

- **SEQ:** Rufe für alle Kinderknoten die Methode `allResultsGo` auf. Wir speichern die zurückgegebene Liste und nutzen diese als Parameter für den nächsten Kinderknoten.

- **CHOICE:** Rufe für alle Kinderknoten die Methode `allResultsGo` auf. Wir berechnen die Resultate für jeden Kinderknoten separat und fügen die Listen zusammen.

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
    if (n.getType().equals("ADD")) {
        LinkedProgramStateList next = new LinkedProgramStateList();
        for (int i = 0; i < states.size; i += 1) {
            ProgramState state = states.get(i);
            next.addLast(new ProgramState(state.getSum() + n.getValue(), state.getCounter() + 1));
        }
        return next;
    }
(…)
}
```

Neue Liste wird erstellt, da wir nicht `states` ändern wollen.

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
    if (n.getType().equals("ADD")) {
        LinkedProgramStateList next = new LinkedProgramStateList();
        for (int i = 0; i < states.size; i += 1) {
            ProgramState state = states.get(i);
            next.addLast(new ProgramState(state.getSum() + n.getValue(), state.getCounter() + 1));
        }
        return next;
    }
(…)
}
```

Für jedes Zwischenresult in der Liste states wird value zur Summe hinzugefügt und der Counter erhöht.

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
    if (n.getType().equals("ADD")) {
        LinkedProgramStateList next = new LinkedProgramStateList();
        for (int i = 0; i < states.size; i += 1) {
            ProgramState state = states.get(i);
            next.addLast(new ProgramState(state.getSum() + n.getValue(), state.getCounter() + 1));
        }
        return next;
    }
(…)
}
```

Für jedes Zwischenresult in der Liste states wird value zur Summe hinzugefügt und der Counter erhöht.

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
(…)
    } else if (n.getType().equals("SEQ")) {
        LinkedProgramStateList next = states;
        for (Node ch : n.getSubnodes()) { //Recursively update the results
            next = allResultsGo(ch, next);
        }
        return next;
    }
(…)
}
```

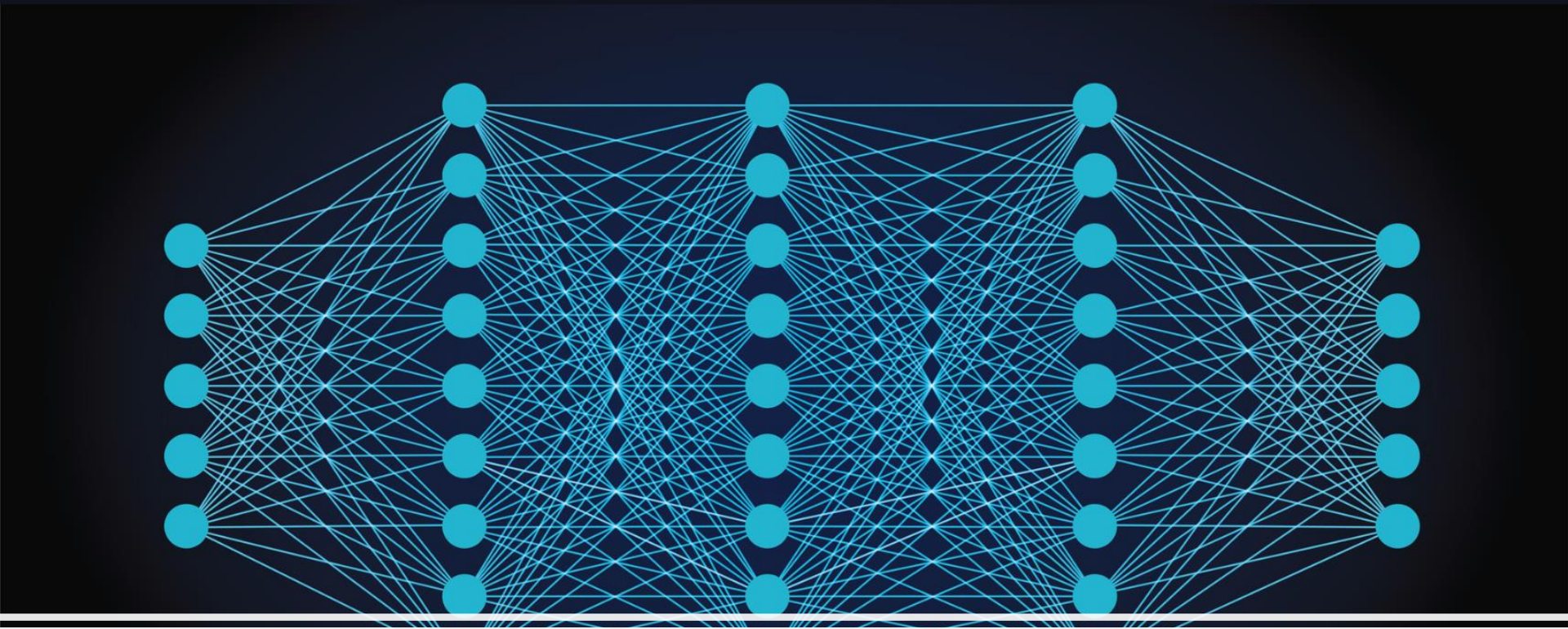
Wir verändern die Liste auf welche states verweist **nicht**, weil wir bei ADD notes eine neue Liste erstellen. Hier wird aber **keine** Kopie erstellt!

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
(…)
    } else if (n.getType().equals("SEQ")) {
        LinkedProgramStateList next = states;
        for (Node ch : n.getSubnodes()) {
            next = allResultsGo(ch, next);
        }
        return next;
    }
(…)
}
```

Wir rufen für jeden Kinderknoten die Methode rekursiv auf.

```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
(…)
    } else if (n.getType().equals("CHOICE")) {
        LinkedProgramStateList next = new LinkedProgramStateList();
        for (Node ch : n.getSubnodes()) {
            LinkedProgramStateList results = allResultsGo(ch, states);
            for (int i = 0; i < results.size; i += 1) {
                next.addLast(results.get(i));
            }
        }
        return next;
    }
    return null;
}
```

Für jeden Kinderknoten wird eine Liste zurückgegeben und wir fügen diese dann zusammen.

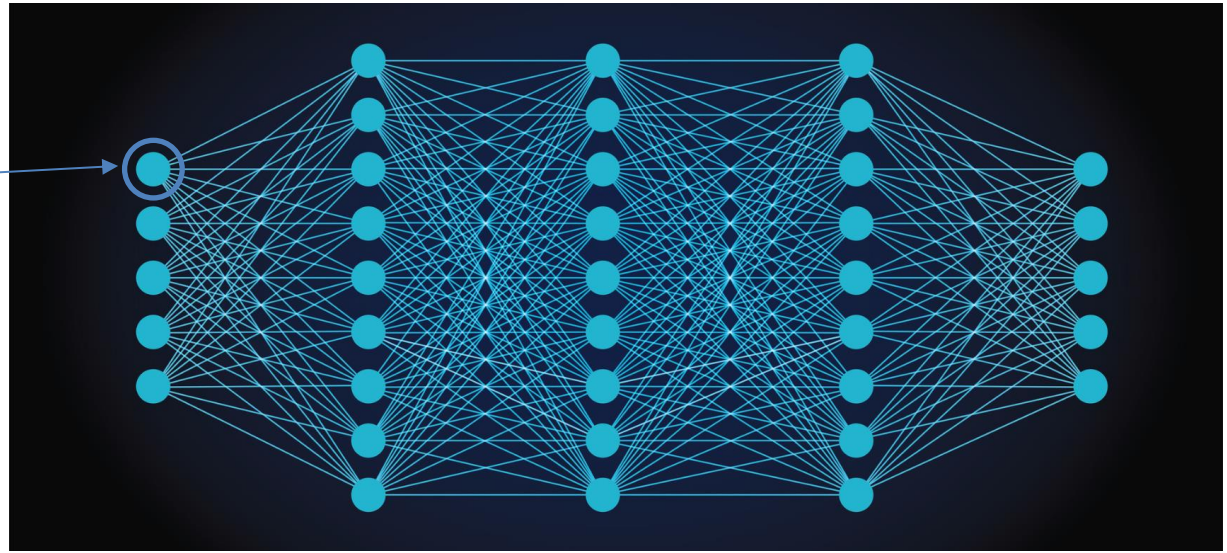


Klassen: Neural Network

Klassen: Neural Network

Ziel: AI in Java. Dafür brauchen wir Neuronale Netwerke (NNs)

- Wir schreiben also eine Klasse `NeuralNetwork`, welche ein neuronales Netzwerk modelliert.

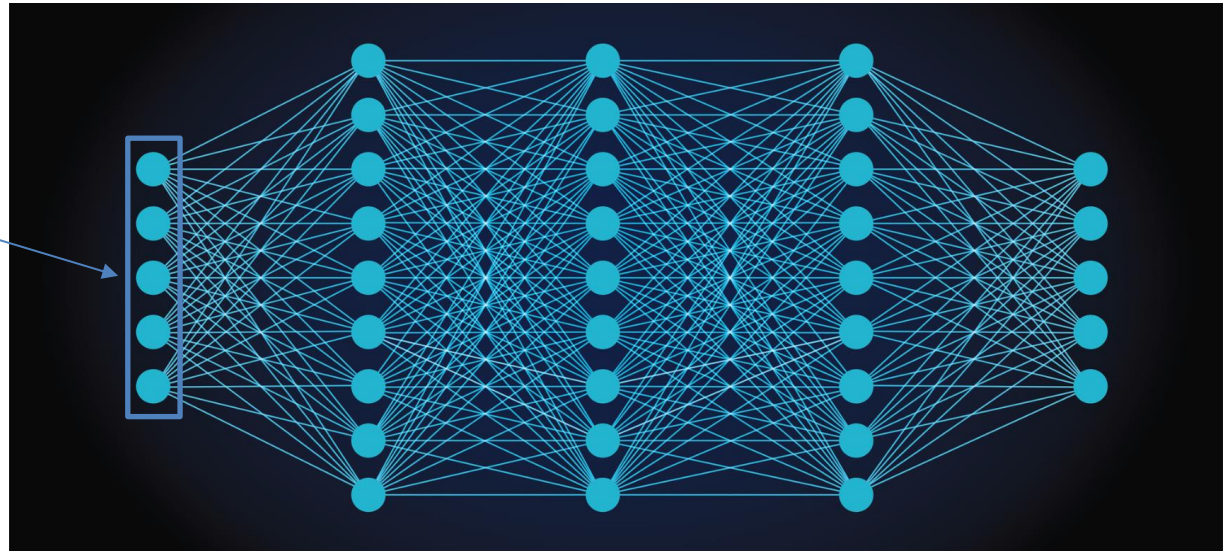


Klassen: Neural Network

Ziel: AI in Java. Dafür brauchen wir Neuronale Netwerke (NNs)

- Wir schreiben also eine Klasse `NeuralNetwork`, welche ein neuronales Netzwerk modelliert.

Ein “Layer” von einem NN.

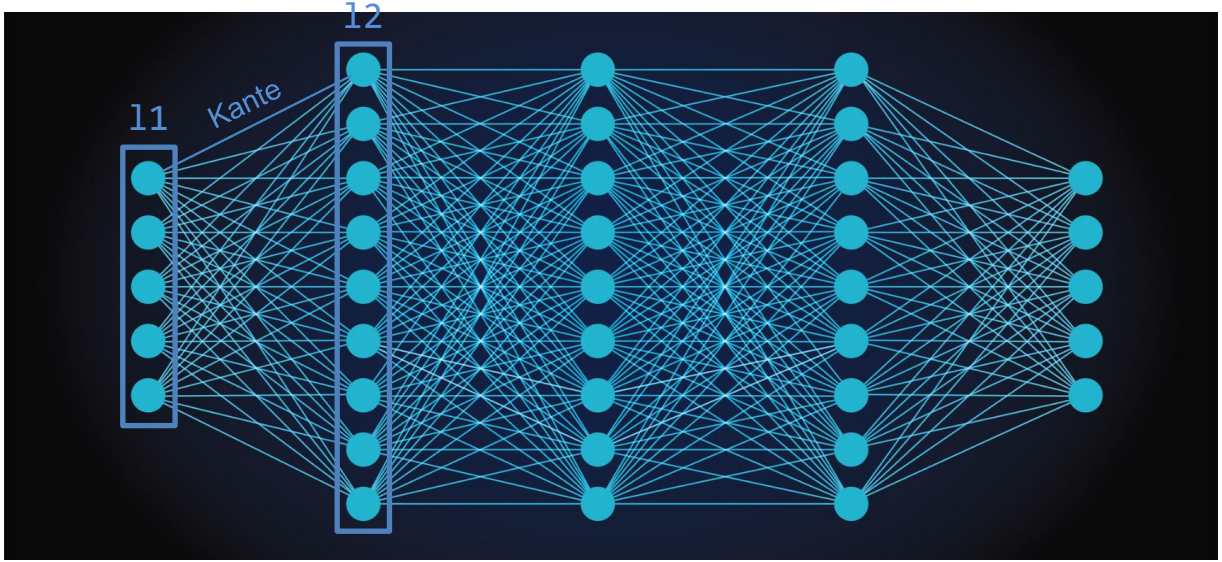


Klassen: Neural Network

Ziel: AI in Java. Dafür brauchen wir Neuronale Netwerke (NNs)

- Wir schreiben also eine Klasse `NeuralNetwork`, welche ein neuronales Netzwerk modelliert.

Zwei Layers l1 und l2 welche durch Kanten verbunden sind, welche jeweils ein Gewicht besitzen.

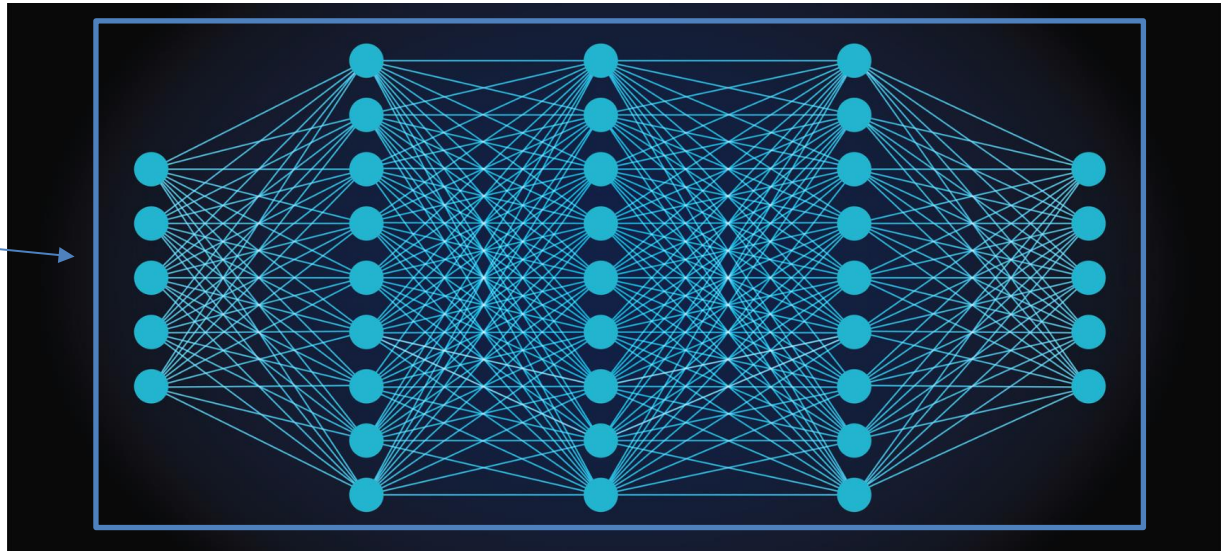


Klassen: Neural Network

Ziel: AI in Java. Dafür brauchen wir Neuronale Netwerke (NNs)

- Wir schreiben also eine Klasse `NeuralNetwork`, welche ein neuronales Netzwerk modelliert.

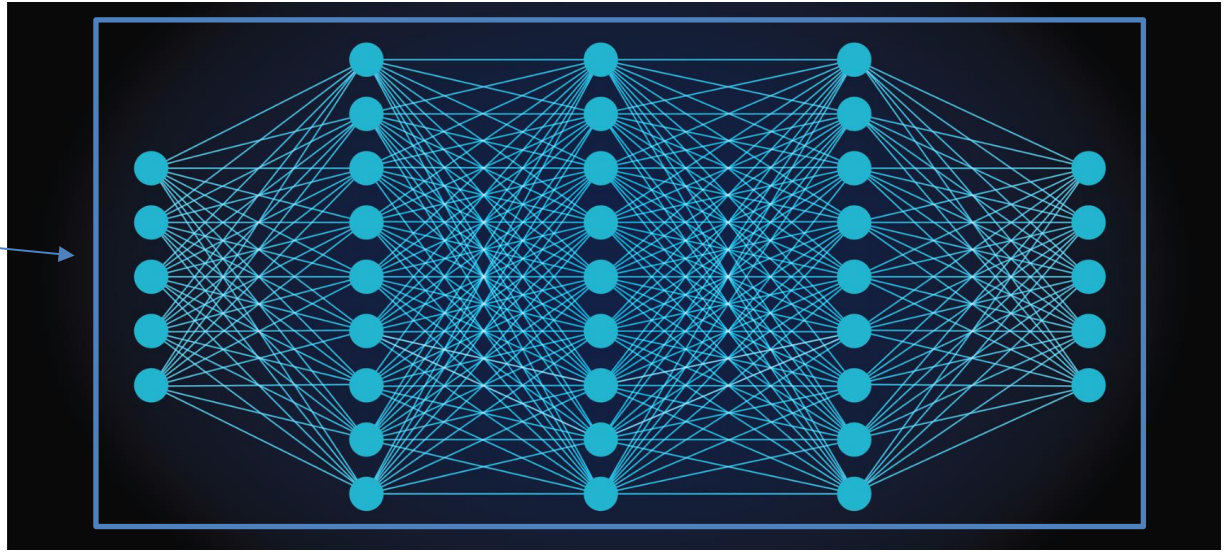
Das NN



Klassen: Neural Network

Das Neural Netzwerk ist aber nicht nur eine Ansammlung an Layers, Weights und Neuronen (was einen **Typ** definiert). Es besitzt zusätzlich ein **Verhalten**, welches durch Methoden der Klasse definiert wird.

Das NN



```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Klasse (gespeichert in Neuron.java) – `public` access modifier

```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Attribut – `private` access modifier

Wir wollen, dass jeder die Klasse nutzen kann, aber der Zugriff auf die Attribute nur durch Methoden der Klasse erlaubt ist.

```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Konstruktoren

Getter und Setter Methoden

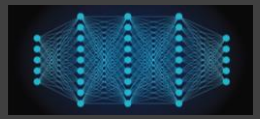
```java
public class Layer {
    private Neuron[] neurons;
    private double[] weights;

    public Layer(Neuron[] neurons, double[] weights) {
        this.neurons = neurons;
        this.weights = weights;
    }

    public Layer() {
        this(null, null);
    }

    public double[] getWeights() {
        return weights;
    }

    public void setWeights(double[] weights) {
        this.weights = weights;
    }

    public void changeWeight(int index, double value) {
        this.weights[index] = value;
    }
}
```

```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Ein Benutzer kann nach dem Erstellen eines Layers nur noch die Weights ändern, nicht mehr aber die interne Struktur des Layers.

Klassenmethode – auch Member-Methode

```java
public class NeuralNetwork {
    private Layer[] layers;

    public NeuralNetwork(Layer[] layers) {
        this.layers = layers;
    }

    public NeuralNetwork() {
        this(null);
    }

    public Layer getOutputs() {
        return this.layers[layers.length - 1];
    }

    public void train() {
        // TODO
    }
}
```



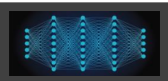
```java
public class Layer {
    private Neuron[] neurons;
    private double[] weights;

    public Layer(Neuron[] neurons, double[] weights) {
        this.neurons = neurons;
        this.weights = weights;
    }

    public Layer() {
        this(null, null);
    }

    public double[] getWeights() {
        return weights;
    }

    public void setWeights(double[] weights) {
        this.weights = weights;
    }

    public void changeWeight(int index, double value) {
        this.weights[index] = value;
    }
}
```

Layer.java

```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value)
{
        this.value = value;
    }
}
```

Neuron.java

Hier geben wir dem User nur Zugriff auf den Output des NNs der Rest geschieht intern.



```java
public class NeuralNetwork {
    private Layer[] layers;

    public NeuralNetwork(Layer[] layers) {
        this.layers = layers;
    }

    public NeuralNetwork() {
        this(null);
    }

    public Layer getOutputs() {
        return this.layers[layers.length - 1];
    }

    public void train() {
        // TODO
    }
}
```

NeuralNetwork.java

```java
public class Layer {
    private Neuron[] neurons;
    private double[] weights;

    public Layer(Neuron[] neurons, double[] weights) {
        this.neurons = neurons;
        this.weights = weights;
    }

    public Layer() {
        this(null, null);
    }

    public double[] getWeights() {
        return weights;
    }

    public void setWeights(double[] weights) {
        this.weights = weights;
    }

    public void changeWeight(int index, double value) {
        this.weights[index] = value;
    }
}
```

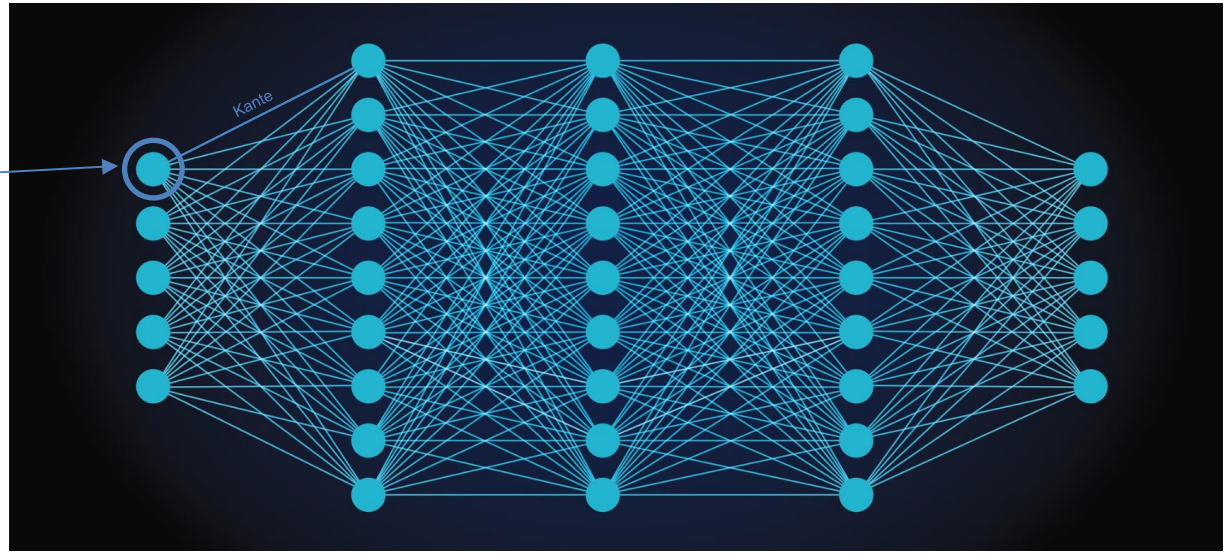
Layer.java

```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Neuron.java

Klassen: Neural Network

Problem: Was wenn wir nicht jedes Neuron in einem Layer mit allen Neuronen im nächsten Layer mit Kanten verbinden wollen?

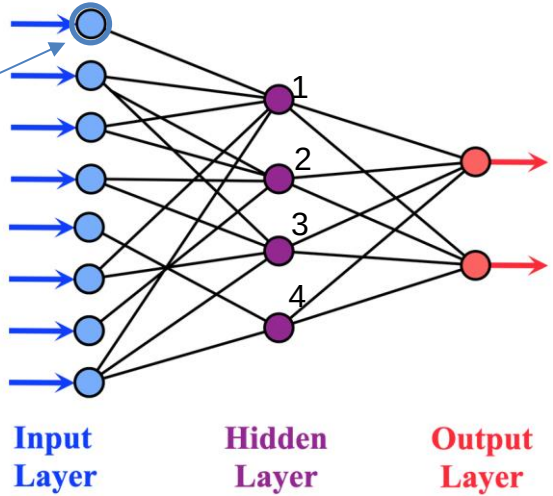


Klassen: Neural Network

Problem: Was wenn wir nicht jedes Neuron in einem Layer mit allen Neuronen im nächsten Layer mit Kanten verbinden wollen?

Ein `boolean` array pro Neuron würde reichen.

1	0	0	0
1	2	3	4

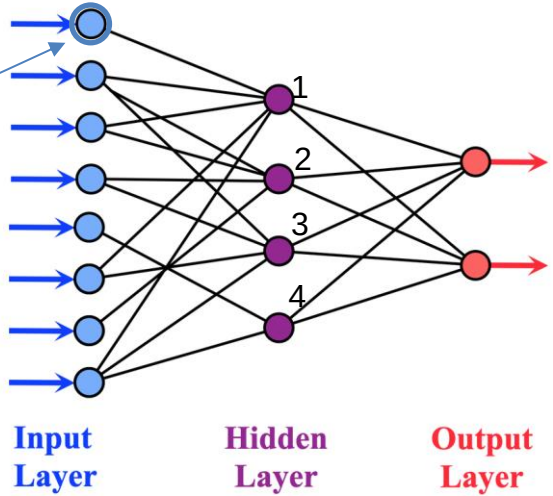


Klassen: Neural Network

Problem: Was wenn wir nicht jedes Neuron in einem Layer mit allen Neuronen im nächsten Layer mit Kanten verbinden wollen?

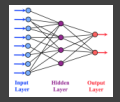
Ein `boolean` array pro Neuron würde reichen.

1	0	0	0
1	2	3	4



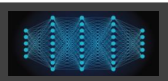
✨ **Vererbung!** ✨

```java
public class SparseNetwork extends NeuralNetwork {

    private boolean[][] connections;

    public SparseNetwork(Layer[] layers, boolean[][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



SparseNetwork.java

```java
public class NeuralNetwork {
    private Layer[] layers;

    public NeuralNetwork(Layer[] layers) {
        this.layers = layers;
    }

    public NeuralNetwork() {
        this(null);
    }

    public Layer getOutputs() {
        return this.layers[layers.length - 1];
    }

    public void train() {
        // TODO
    }
}
```



NeuralNetwork.java

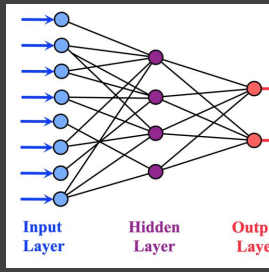
```java
public class Neuron {
    private int value;

    public Neuron(int value) {
        this.value = value;
    }

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

Neuron.java

```java
public class Layer {
    private Neuron[] neurons;
    private double[] weights;

    public Layer(Neuron[] neurons, double[] weights)
{
        this.neurons = neurons;
        this.weights = weights;
    }

    public Layer() {
        this(null, null);
    }

    public double[] getWeights() {
        return weights;
    }

    public void setWeights(double[] weights) {
        this.weights = weights;
    }

    public void changeWeight(int index, double value)
{
        this.weights[index] = value;
    }
}
```

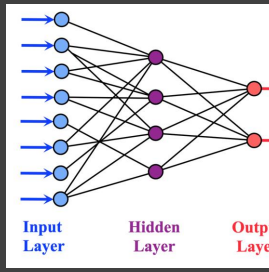
Layer.java

```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



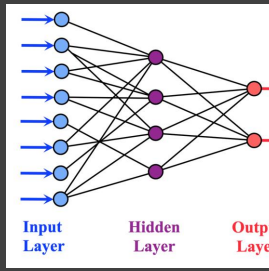
Connections Array pro Layer.
In jedem Layer ein Array pro Neuron.

```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



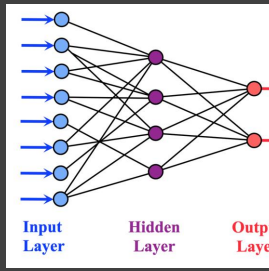
Die restlichen Attribute werden von NeuralNetwork geerbt.

```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



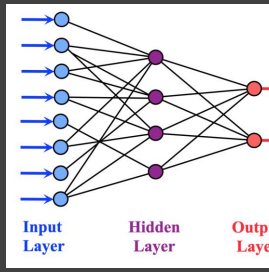
Konstruktoren werden nie geerbt!

```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



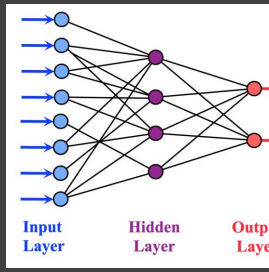
Wir rufen den Konstruktor der Superklasse und initialisieren das Connections-Array zusätzlich.

```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```



Die `train`-Methode aus der Superklasse kennt kein `connection` Array. Wir überschreiben diese Methode also.

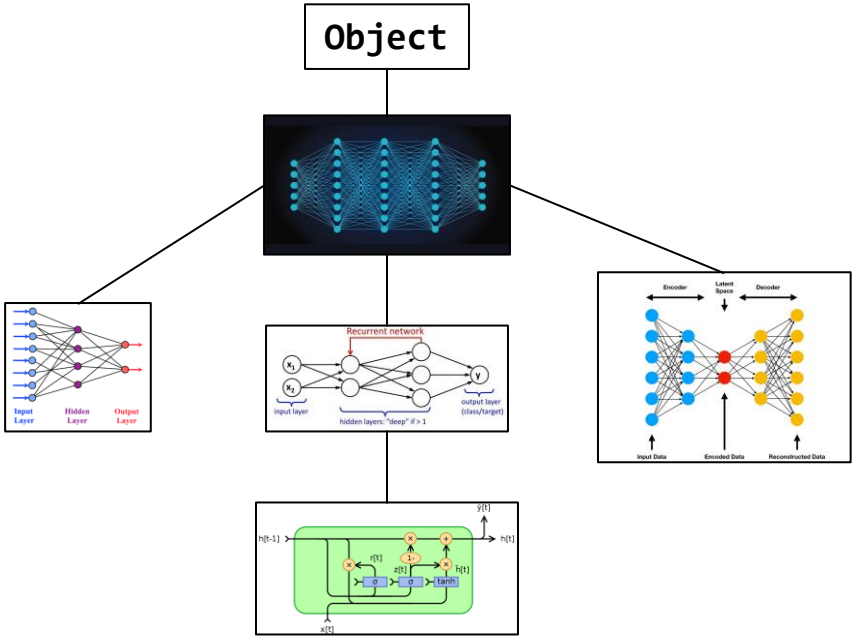
```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

    public SparseNetwork(Layer[] layers, boolean[][][] connections) {
        super(layers);
        this.connections = connections;
    }

    public SparseNetwork() {
        SparseNetwork(null, null);
    }

    @Override
    public void train() {
        // TODO
    }
}
```

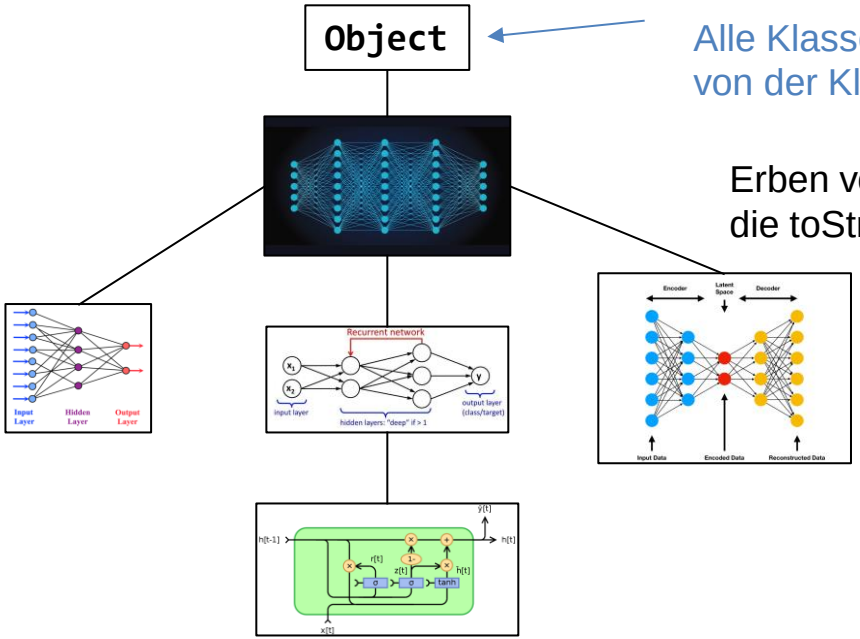


@Override stellt sicher, dass dies wirklich eine Überschreibung ist. Ansonsten gibt es einen Fehler bei der Ausführung.

Klassen: Neural Network

Wir könnten diverse neuronale Netzwerke so durch eine Klasse beschreiben…

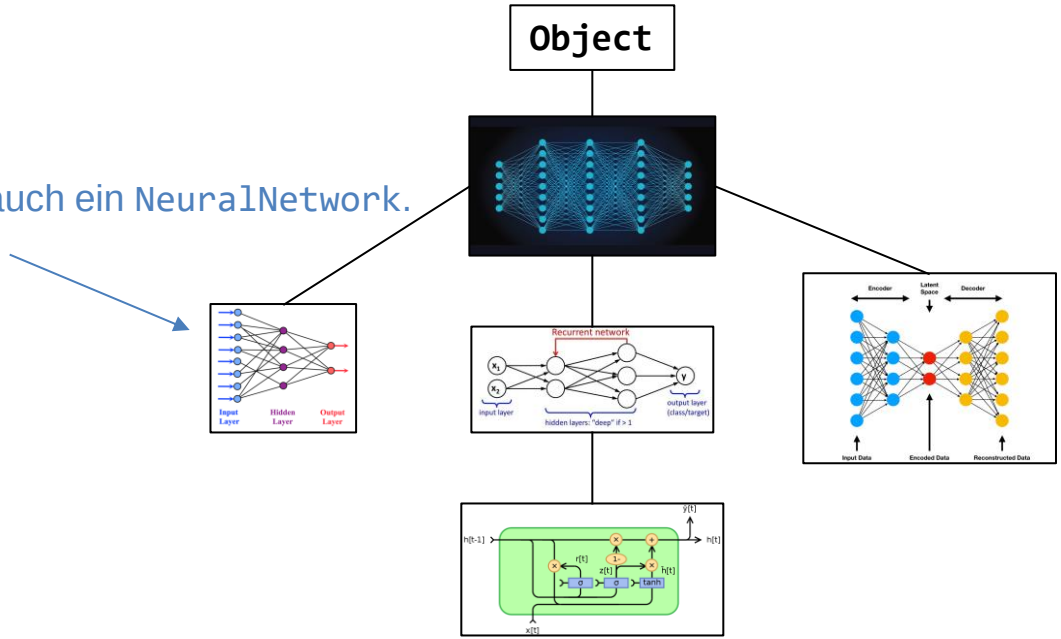




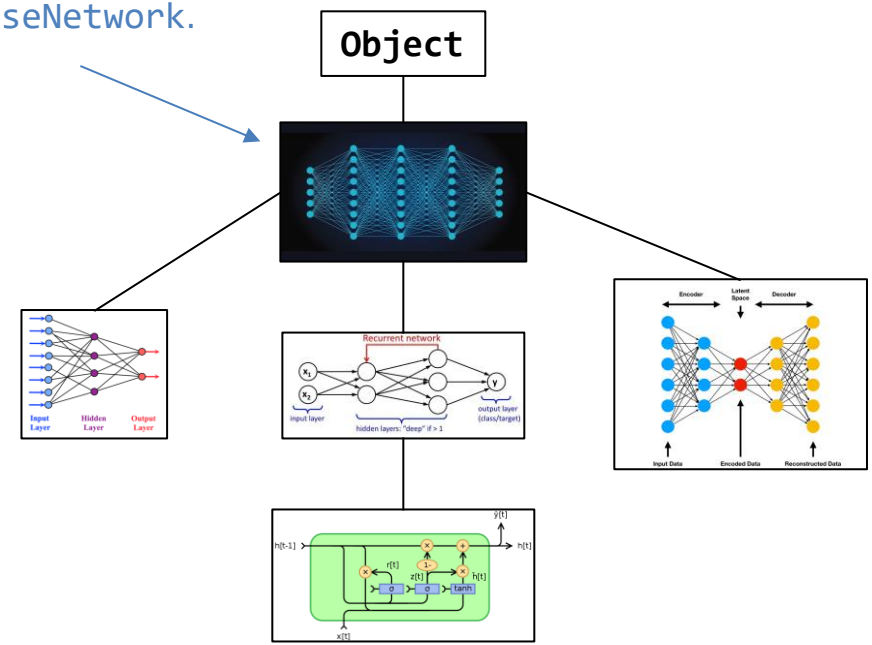
Alle Klassen sind Subklassen von der Klasse `Object`.

Erben von Object zum Beispiel die toString Methode

SparseNetwork ist auch ein NeuralNetwork.

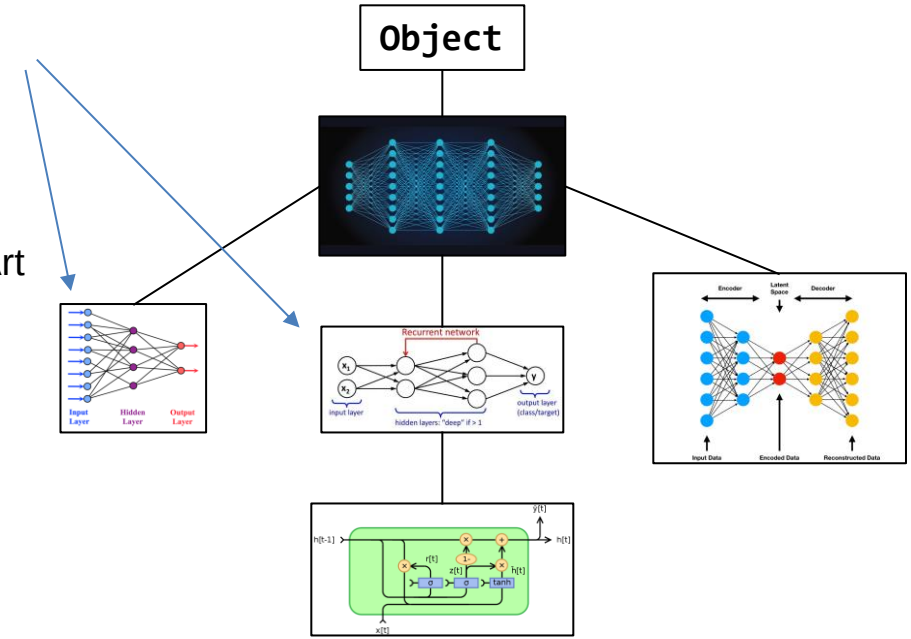


`NeuralNetwork` ist kein `SparseNetwork`.



Diese zwei Klassen erben beide von `NeuralNetwork`, aber sie sind nicht direkt verwandt.

Aber SparseNetwork, ConvolutionalNetwork, SickoNetwork sind alles eine Art NeuralNetwork und sie haben viel Gemeinsam



- **see InheritanceExample.java**

LinkedList vs ArrayList

ArrayList

Wir benutzen die `ArrayList` als ein Array ohne fixe Länge. Statt `int`, `boolean`, `double` benutzen wir `Integer`, `Boolean`, `Double` (die Wrapper-Typen).

Operation	Array	ArrayList
Initialisieren mit Typ `int`	`= new int[5];`	`= new ArrayList<Integer>();`
Initialisieren mit Typ `double`	`= new double[5];`	`= new ArrayList<Double>();`
Initialisieren mit Typ `boolean`	`= new boolean[5];`	`= new ArrayList<Boolean>();`

ArrayList

Wir benutzen die `ArrayList` als ein Array ohne fixe Länge. Statt `int`, `boolean`, `double` benutzen wir `Integer`, `Boolean`, `Double` (die Wrapper-Typen).

Operation	Array `arr`	ArrayList `arrList`
Lese Element an Index `i`	`arr[i]`	`arrList.get(i)`
Element an Index `i` auf `e` setzen	`arr[i] = e;`	`arrList.set(i, e);`
Erstes Element	`arr[0]`	`arrList.getFirst()`
Letztes Element	`arr[arr.length – 1]`	`arrList.getLast()`
Länge	`arr.length`	`arrList.size()`

ArrayList

Wir benutzen die `ArrayList` als ein Array ohne fixe Länge. Statt `int`, `boolean`, `double` benutzen wir `Integer`, `Boolean`, `Double` (die Wrapper-Typen).

Operation	ArrayList `arrList`
Füge Element `e` an Index `i` hinzu	`arrList.add(i, e)`
Füge Element `e` am Anfang der Liste hinzu	`arrList.addFirst(e);`
Füge Element `e` am Ende der Liste hinzu	`arrList.addLast(e)`
Prüfe ob Element `e` enthalten ist	`arrList.contains(e)`
In Array umwandeln	`arrList.toArray()`

LinkedList

Wir benutzen die `LinkedList` wie die Liste, welche in der Vorlesung konstruiert wurde. Sie erlaubt effizientes entfernen / hinzufügen von Elementen und eignet sich deshalb sehr gut als Queue / Stack.

Operation	Array	LinkedList
Initialisieren mit Typ `int`	`= new int[5];`	`= new LinkedList<Integer>();`
Initialisieren mit Typ `double`	`= new double[5];`	`= new LinkedList<Double>();`
Initialisieren mit Typ `boolean`	`= new boolean[5];`	`= new LinkedList<Boolean>();`

Ebenfalls funktionieren alle vorherigen Methoden von ArrayList auch für die LinkedList.

LinkedList

Wir benutzen die `LinkedList` wie die Liste, welche in der Vorlesung konstruiert wurde. Sie erlaubt effizientes entfernen / hinzufügen von Elementen und eignet sich deshalb sehr gut als Queue / Stack.

Operation	LinkedList `list`
Liste als Stack (Element entfernen)	`list.pop()`
Liste als Stack (Element e hinzufügen)	`list.push(e)`
Liste als Queue (Element entfernen)	`list.poll()`
Liste als Queue (Element e hinzufügen)	`list.add(e)`

LinkedList

Wir benutzen die `LinkedList` wie die Liste, welche in der Vorlesung konstruiert wurde. Sie erlaubt effizientes entfernen / hinzufügen von Elementen und eignet sich deshalb sehr gut als Queue / Stack.

Operation	`LinkedList list`
Liste als Stack (Element entfernen)	`list.pop()`
Liste als Stack (Element e hinzufügen)	`list.push(e)`
Liste als Queue (Element entfernen)	`list.poll()`
Liste als Queue (Element e hinzufügen)	`list.add(e)`

Schwierig zu merken

LinkedList

Wir benutzen die `LinkedList` wie die Liste, welche in der Vorlesung konstruiert wurde. Sie erlaubt effizientes entfernen / hinzufügen von Elementen und eignet sich deshalb sehr gut als Queue / Stack.

Operation	LinkedList `list`
Liste als Stack (Element entfernen)	`list.removeFirst()`
Liste als Stack (Element e hinzufügen)	`list.addFirst(e)`
Liste als Queue (Element entfernen)	`list.removeLast()`
Liste als Queue (Element e hinzufügen)	`list.addFirst(e)`

Prüfungsaufgaben

- **Das ist lowkey advanced also falls nicht alles 100% klar ist, ist das nicht so schlimm, wir sehen das eh nächste Woche. (und übernächste (und nächste))**
- **FS20**

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

Implements können wir erstmal ignorieren. Das ist im Prinzip das selbe wie extends nur dass wir nichts tatsächlich erben, sondern nur gezwungen werden etwas zu implementieren.

Siehe Beispiel code

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

2.

```
Beta b = new Beta();
b.method1();
```

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

2.

```
Beta b = new Beta();
b.method1();
```

Beta m1 x = Beta

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Alpha a = new Beta();
a.method1();
```

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Alpha a = new Beta();
a.method1();
```

Beta m1 x = Beta

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Alpha a = new Gamma();
a.method1();
```

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Alpha a = new Gamma();
a.method1();
```

Gamma m1 x = Gamma
Gamma2 m0 x = Gamma

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Phi p = new Phi();
p.method1();
```

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

```
Phi p = new Phi();
p.method1();
```

Compiler-Fehler

Gegeben seien diese Klassen und Interfaces in separaten Dateien (der DefaultPackage):

```
interface I1 {
    public void method1();
}

interface I2 {
    public void method1();
}

class Alpha implements I1 {
    String x = "Alpha";

    public void method1() {
        System.out.println("Alpha m1 x = " + x);
    }

}

class Beta  extends Alpha implements I2 {
    String x = "Beta";

    public void method1() {
        System.out.println("Beta m1 x = " + x);
    }
}

class Phi implements I1 {
    String x = "Phi";

    public void method1() {
        super.method1();
        System.out.println("Phi m1 x = " + x);
    }
}
```

```
class Gamma  extends Alpha {
    String x = "Gamma";

    void method0() {
        System.out.println("Gamma2 m0 x = " + x);
    }
    public void method1() {
        System.out.println("Gamma m1 x = " + x);
        method0();
    }
}

class Iota extends Gamma  {

    public void method0() {
        System.out.println("Iota m0  x = " + x);
    }
}

class Eta extends Gamma  {

    void method0(String s) {
        System.out.println("Eta m0 s =" + s);
    }
}
```

Für mehr siehe FS20

Was sind Interfaces

- An interface in Java is a blueprint for a class that defines a set of abstract methods that the implementing class must provide. It allows you to define a contract that other classes must adhere to. (selbst verfasst)
- Wann nutzen wir das? Wenn wir Klassen dazu zwingen möchten irgendeine Methode zu implementieren. Falls eine Klasse ein Interface implementiert, dann wissen wir, dass alle Methoden von dem Interface in er Klasse sind und irgendwas machen.

Prüfungsaufgaben

- Beispiel: wir haben eine Klasse Person und haben eine LinkedList mit Personen. Nun wollen wir die sortieren nach Alter. Weil LinkedList eine Collection ist, können wir Collections.sort(people_list) machen
- Aber Java weiss nicht wie man die Liste sortieren soll. Wie vergleiche ich zwei Personen? (grau zone, bitte meldet mich nicht bei eth)
- Collections.sort erwartet deswegen, dass wir eine compareTo Methode implementieren, die uns das ermöglicht
- Genauer: Person muss das Comparable Interface implementieren, welches verlangt, dass die compareTo methode definiert ist

- **See code example interfaces**

- **Okay, zurück zu der Prüfungsaufgabe**

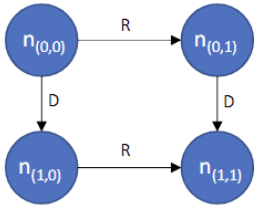
Vorbesprechung

Aufgabe 1: Square Grid

In dieser Aufgabe betrachten wir gerichtete Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g, h können gleich sein). Wir unterscheiden dabei zwischen der rechten und der unteren Kante (und damit dem rechten und dem unteren Knoten).

Die Klasse `Node` repräsentiert einen Knoten in einem solchen Graphen. Die Methode `Node.getRight()` (bzw. `Node.getDown()`) gibt den rechten Knoten (bzw. unteren Knoten) zurück (als `Node`-Objekt). Wenn der rechte Knoten von n_0 nicht existiert, dann gibt `Node.getRight()` `null` zurück (analog für den unteren Knoten). Die Methode `Node.setRight(Node r)` (bzw. `Node.setDown(Node d)`) setzt den rechten (bzw. unteren) Knoten.

Das Ziel der Aufgabe ist, einen von einem `Node`-Objekt definierten Graphen zu analysieren. Konkret geht es darum, die Grösse des grössten quadratischen Gitters in dem Graphen zu bestimmen, der mit dem übergebenen `Node`-Objekt beschrieben wird, welches den gleichen Ursprungsknoten wie der Graph hat.



Aufgabe 1: Square Grid

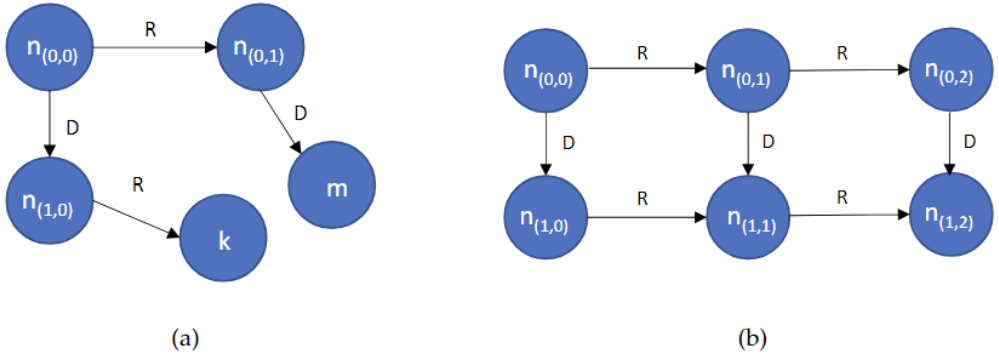


Abbildung 2: Graphen mit quadratischen Gittern als Teilgraphen

Referenzen vs Objekte

Macht die, sowas kommt oft in der Prüfung

Aufgabe 2: Umkehrung

In einem vorherigen Übungsblatt haben Sie eine Linked List für Integers implementiert. In dieser Aufgabe fügen Sie dieser `LinkedIntList` eine weitere Methode hinzu, welche die Liste umkehrt. Eine Liste gilt als umgekehrt, wenn für jedes Paar von Nodes `a` und `b`, für welche zuvor `a == b.next` gegolten hat, in der neuen (umgekehrten) Liste `b == a.next` gilt. Zusätzlich entspricht nach der Umkehrung der erste Node der neuen Liste dem letzten Node der ursprünglichen Liste (und umgekehrt).

Vervollständigen Sie die Methode `reverse()` in der Klasse `LinkedIntList`. Die Methode soll, wie oben definiert, die Liste umkehren. Achten Sie darauf, dass Sie wirklich die Reihenfolge der Nodes selbst umkehren. Es reicht nicht aus, die Reihenfolge der enthaltenen `int`-Werte umzukehren. Es müssen auch in der umgekehrten Liste dieselben Instanzen von `IntNodes` wie in der ursprünglichen Liste verwendet werden. Erstellen Sie also *keine* neuen `IntNodes` mit `new IntNode()`. In der Datei "UmkehrungTest.java" finden Sie einen einfachen Test.

Aufgabe 3: “KI” für das Ratespiel

In Übung 5 implementierten Sie ein Spiel, in welchem der Computer ein Wort auswählt und der Spieler dieses erraten muss. Dort war der Spieler der Benutzer des Programms. In dieser Aufgabe sollen Sie verschiedene “künstliche” Spieler entwickeln. Das heisst, anstelle des Menschen, der über die Konsole Tipps eingibt, werden die Tipps von (mehr oder weniger “intelligenten”) Programmen abgegeben. Ihr Ziel ist es, einen künstlichen Spieler zu entwickeln, der über mehrere Spiele hinweg die Wörter in so wenig Versuchen wie möglich errät.

Die Übungsvorlage enthält bereits den Code für das Ratespiel. Gegenüber Übung 5 ist dieser nun in verschiedene Klassen aufgeteilt. Die drei Hauptklassen sind `RateSpiel`, `Computer` und `Spieler`. Die Klasse `RateSpielApp` enthält eine `main`-Methode, welche das Spiel aufsetzt und durchführt. Durch die Aufteilung ist es möglich, mittels Vererbung Spieler mit unterschiedlichem Verhalten zu schreiben. Die Klasse `Spieler` enthält nämlich nur die Deklarationen der benötigten Methoden, aber keine (sinnvolle) Funktionalität. Subklassen von `Spieler` überschreiben diese Methoden und definieren damit das Verhalten eines Spielers.

Ein konkreter Spieler ist ebenfalls schon in der Vorlage vorhanden: der `KonsolenSpieler`. Dieser besitzt allerdings keine eigene “Intelligenz”, sondern holt sich die Tipps über die Konsole vom Benutzer. Ein `RateSpiel` mit einem `KonsolenSpieler` verhält sich also so wie das Spiel in Übung 5. Starten Sie die `RateSpielApp` und überzeugen Sie sich selbst[1].

Aufgabe 4: Klassenrätsel

In dieser Aufgabe sollen Sie zeigen, dass Sie mit Klassen und Vererbung umgehen können. Im Anhang A finden Sie ein Programm, welches Instanzen von Klassen erstellt und Methoden aufruft. Das Programm macht nichts Sinnvolles und dient nur dem Testen Ihrer Fähigkeiten. In Anhang B befinden sich die verwendeten Klassen, jedoch sind die Klassen noch nicht vollständig. Bei manchen der Klassen fehlt noch die `extends`-Klausel, welche angibt, dass eine Klasse von einer anderen Klasse erbt. Ihre Aufgabe ist es, die nötigen `extends`-Klauseln hinzuzufügen, so dass alles kompiliert und so dass die Ausgabe des Programms von Anhang A am Ende so aussieht wie im Anhang C gezeigt.

Der Code von Anhang A and Anhang B befindet sich in Ihrem `src`-Ordner. Zusätzlich enthält "KlassenTest.java" einen Unit-Test, welcher prüft, ob die Ausgabe des Programms dem Output aus Anhang C entspricht. Beachten Sie, dass Sie für diese Aufgabe **ausschliesslich** `extends`-Klauseln hinzufügen (diese kann es nur an den grauen Boxen aus Anhang B geben), kein anderer Code darf verändert werden.

Tipp: Lösen Sie die Aufgabe zuerst auf Papier, ohne die Hilfe von Eclipse. Sobald Sie herausgefunden haben, welche Klassen von welchen Klassen erben, testen Sie Ihre Lösung in Eclipse. Dies hilft Ihnen, Ihr Wissen über Vererbung zu testen. In der Vergangenheit wurden ähnliche Aufgaben im schriftlichen Teil der Prüfung gestellt.

Wichtig für die Prüfung! Wenn ihr nichts macht, dann macht das. Aber actually alle Aufgaben sind ganz gut bis auf vlt dieses KI und Umkehrung

Bonus

- **Arbeitet mit collections**
- **Nutzt collections.sort()**
- **Definiert euch Helferklassen, wrapper etc.**

Kahoot

- **https://create.kahoot.it/details/8cbdb70a-231e-4b7e-af95-5ffbbd5f47c1**