252-0027

Einführung in die Programmierung

Übungen

Woche 10: Verlinkte Objekte, Klassen

Timo Baumberger
Departement Informatik
ETH Zürich

Organisatorisches

- Mein Name: Timo Baumberger
- Website: timobaumberger.com
- Bei Fragen: tbaumberger@student.ethz.ch
  - Mails bitte mit «[EProg24]» im Betreff
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
  - Abgabe immer via Git
  - Lösungen in separatem Projekt auf Git

Never Ask A Woman

Her Age

A Man,

His Salary

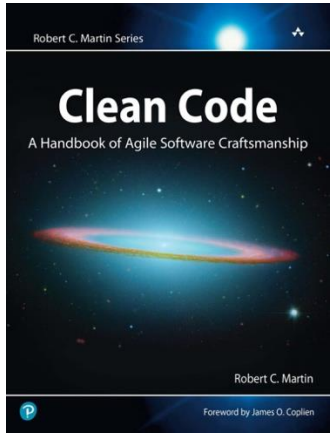
An ETH student

How the semester
is going during november

Programm

- Bonusaufgabe von u08
- Executable Graph Aufgabe
- Vererbung
- Neural Network
- ArrayList vs LinkedList
- Vorbesprechung
- Nachbesprechung
- Kahoot (wirklich 😂)

Wieso OOP???



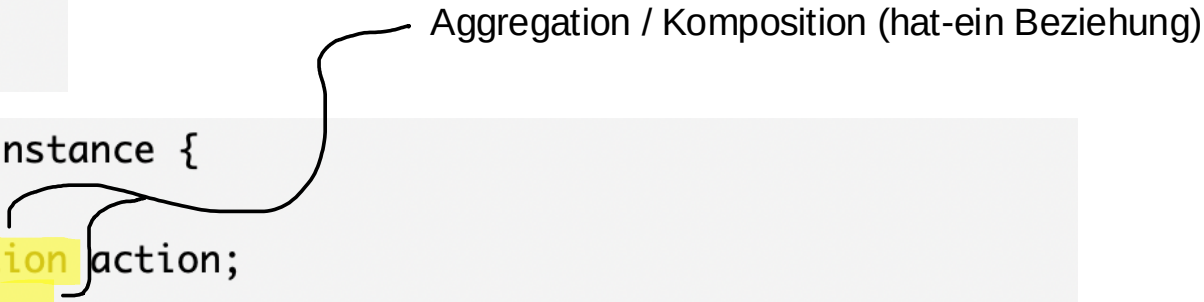
- Wird oft in der Software-Instudrie verwendet
- Ist mit den meisten Programmiersprachen möglich
- Wichtig um den Code von anderen Personen zu verstehen
- Gibt dem Code eine Struktur (Modularität)

> **The object-oriented model makes it easy to build up programs by accretion. What this often means, in practice, is that it provides a structured way to write spaghetti code.**
>
> Paul Graham

Bonusaufgabe u08

```java
public enum Action {
    ATTACK,
    SUMMON
}
```

```java
public class ActionInstance {

    public final Action action;
    public final Human source;
    public int delay;

    public ActionInstance(Action action, Human source, int delay) {
        this.action = action;
        this.source = source;
        this.delay = delay;
    }
}
```



```java
public enum Type {

    JESTER(0, 0),
    WARRIOR(0, 1),
    CLERIC(0, 2);

    private int attackDelay;
    private int summonDelay;

    private Type(int attackDelay, int summonDelay) {
        this.attackDelay = attackDelay;
        this.summonDelay = summonDelay;
    }

    public int getDelay(Action action) {
        if (action == Action.ATTACK) {
            return attackDelay;
        } else {
            return summonDelay;
        }
    }
}
```

Die Klasse Type bietet eine Funktion an, die den korrekten Delay zurückgibt

```java
public class Human {

    public final Game gameInstance;
    public ActionInstance actionInstance;

    public final Type type;

    public int health, position;

    public Human(int health, int position, Type type, Game gameInstance) {
        this.health = health;
        this.position = position;
        this.type = type;
        this.gameInstance = gameInstance;
    }

    public int getHealth() {
        return health;
    }

    public int getPosition() {
        return position;
    }

    public boolean isAlive() {
        return health > 0;
    }
```

Nicht notwendig, da wir bereits die Klasse ActionInstance haben. Hätte aber alternativ verwendet werden können, um die Felder aus ActionInstance zu speichern

Boolean Flag hätte hier gereicht

```
29     public boolean scheduleAction(Action action) {
30         if (!isAlive()) {
31             return false;
32         }
33         if (actionInstance != null) {
34             return false;
35         }
36         int delay = type.getDelay(action);
37         ActionInstance instance = new ActionInstance(action, this, delay);
38         actionInstance = instance;
39         gameInstance.actionInstances.add(instance);
40         return true;
41     }
42 }
```

Könnte man zusammenfassen

Siehe vorherige Slide

```java
public class Game {

    public final List<Human> humans;
    public final List<ActionInstance> actionInstances;

    public Game() {
        this.humans = new ArrayList<>(100);
        this.actionInstances = new ArrayList<>(100);
    }
```

Man hätte auch nur Human speichern können, da im Human auch die ActionInstance vorhanden ist

```java
    Human createJester(int health, int position) {
        Human human = new Human(health, position, Type.JESTER, this);
        humans.add(human);
        return human;
    }

    Human createWarrior(int health, int position) {
        Human human = new Human(health, position, Type.WARRIOR, this);
        humans.add(human);
        return human;
    }

    Human createCleric(int health, int position) {
        Human human = new Human(health, position, Type.CLERIC, this);
        humans.add(human);
        return human;
    }
}
```

Wieso hat actionInstances eine Kapazität von 100?
- Jeder Human (maximal 100) kann maximal eine ActionInstance haben

Alternative Lösung:
- Verwende 3 Arrays
- Verwalte 3 Positionen
- Speichere array[1] in array[0] ***UND*** array[2] in array[1] ***UND*** array[2] = new ActionInstance[100]

```
11    void advanceTurn() {
12        ListIterator<ActionInstance> it = actionInstances.listIterator();
13        while (it.hasNext()) {
14            ActionInstance actionInstance = it.next();
15            Human human = actionInstance.source;
16
17            if (!human.isAlive()) {
18                human.actionInstance = null;
19                it.remove();
20                continue;
21            }
22            if (actionInstance.delay != 0) {
23                actionInstance.delay--;
24                continue;
25            }
26            // execute action
27            human.actionInstance = null;
28            it.remove();
29            if (human.type == Type.WARRIOR) {
30                if (actionInstance.action == Action.SUMMON) {
31                    human.health -= 5;
32                } else if (actionInstance.action == Action.ATTACK) {
33                    for (Human neighbour : humans) {
34                        int diff = Math.abs(human.position - neighbour.position);
35                        if (diff == 1) {
36                            neighbour.health -= 10;
37                        }
38                    }
39                }
40            } else if (human.type == Type.CLERIC) {
```

Wenn der assozierte Human (source) nicht mehr lebt können wir abbrechen

Falls delay != 0 müssen wir noch delay-viele Runden warten

Enums enthalten Konstanten. Wir verwenden somit immer die gleiche Konstante (ansonsten wäre die Instanz nicht konstant 😉)

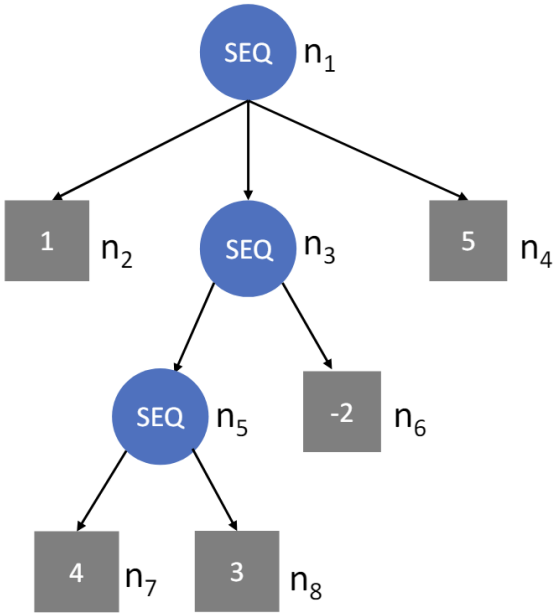
```
40            } else if (human.type == Type.CLERIC) {
41                if (actionInstance.action == Action.SUMMON) {
42                    int i = human.position;
43                    for (Human neighbour : humans) {
44                        if (!neighbour.isAlive()) {
45                            continue;
46                        }
47                        int diff = Math.abs(i - neighbour.position);
48                        if (diff >= 3 && diff <= 5) {
49                            neighbour.position = i;
50                        }
51                    }
52                } else if (actionInstance.action == Action.ATTACK) {
53                    for (Human neighbour : humans) {
54                        int diff = Math.abs(human.position - neighbour.position);
55                        if (diff == 1) {
56                            neighbour.health -= 3;
57                        }
58                    }
59                }
60            }
61        }
62    }
```

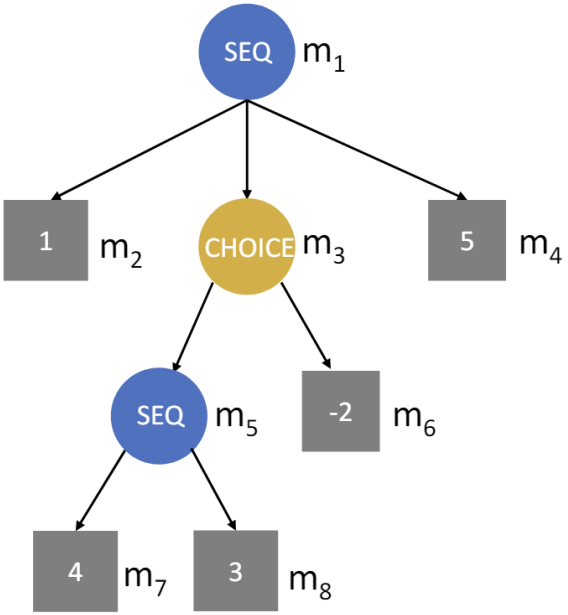
An was hat es gelegen?

- **Übung macht den Meister!**
- **Zu weit gedacht…**
- **ArrayList (Einfüge-Ordnung == Ausgabe-Ordnung)**
- **Iterator (kann während Iteration Elemente löschen)**

Probleme Lösen

Probleme Lösen: Executable Graph





Probleme Lösen: Executable Graph

In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Der Programmzustand ist dabei immer durch ein Tupel $(sum, counter)$ gegeben, wobei *sum* und *counter* ganze Zahlen sind. Programmzustände werden durch `ProgramState`-Objekte modelliert, wobei `ProgramState.getSum()` (bzw. `ProgramState.getCounter()`) dem ersten Element (bzw. dem zweiten Element) des Tupels entspricht.

Eine Ausführung des Programms manipuliert den Programmzustand. Das Resultat eines Programms ist gegeben durch den erreichten Programmzustand, nachdem alle Operationen im Programm ausgeführt wurden. Programme können nichtdeterministisch sein: Das heisst, für ein einzelnes Programm kann es für den gleichen Startzustand mehrere Programmausführungen geben, welche zu unterschiedlichen Resultaten führen.

Knoten in Graphen werden durch `Node`-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n, wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden drei Arten von Knoten, wobei die Methode `Node.getType()` die Knotenart als `String` zurückgibt. Um ein Programm, welches durch den Knoten `n` repräsentiert wird, auszuführen, muss man den "Knoten `n` ausführen". Wir beschreiben nun die drei Knotenarten und jeweils die Ausführung der Knoten:

Probleme Lösen: Executable Graph

1. Additionsknoten (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen Knoten werden bei der Ausführung ignoriert.

2. Sequenzknoten (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten `n` werden die Kinderknoten von `n` nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()` ist irrelevant.

3. Auswahlknoten (`Node.getType()` ist "CHOICE"): Bei der Ausführung eines Auswahlknoten `n` wird ein beliebiger Kinderknoten von `n` ausgewählt und ausgeführt. `Node.getValue()` ist irrelevant. Diese Knoten führen zu Nichtdeterminismus.

Sie dürfen davon ausgehen, dass Sequenz- und Auswahlknoten immer mindestens einen Kinderknoten haben, und dass es zwischen zwei Knoten immer höchstens einen Pfad gibt. Die folgende Abbildung zeigt zwei Beispielgraphen, wobei Knoten mit der Beschriftung "SEQ" (bzw. "CHOICE") Sequenzknoten (bzw. Auswahlknoten) entsprechen und die Zahlen in Additionsknoten den Additionswerten entsprechen.

Probleme Lösen: Executable Graph

Implementieren Sie `GraphExecution.allResults(Node n, ProgramState initState)`, welche für den Startzustand `initState` alle möglichen Resultate für das Programm repräsentiert durch `n` zurückgibt. Die Resultate sollten als eine Liste von `ProgramState`-Objekten zurückgegeben werden (repräsentiert durch die Klasse `LinkedProgramStateList`). Die Reihenfolge der zurückgegeben Liste spielt keine Rolle. Wenn das gleiche Resultat durch genau k verschiedene Ausführungen generiert werden kann, dann muss das Resultat k Mal in der zurückgegeben Liste vorkommen. Zwei Ausführungen sind unterschiedlich, wenn es mindestens einen Knoten gibt, der in einer aber nicht in der anderen Ausführung ausgeführt wird.

Wir stellen zwei Testdateien zur Verfügung. "GraphExecutionTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingGraphExecutionTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie ihre Lösung zuerst ausgiebig mit "GraphExecutionTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingGraphExecutionTest.java" verwenden, um zu sehen, wie ihre Lösung an einer Prüfung abgeschnitten hätte.

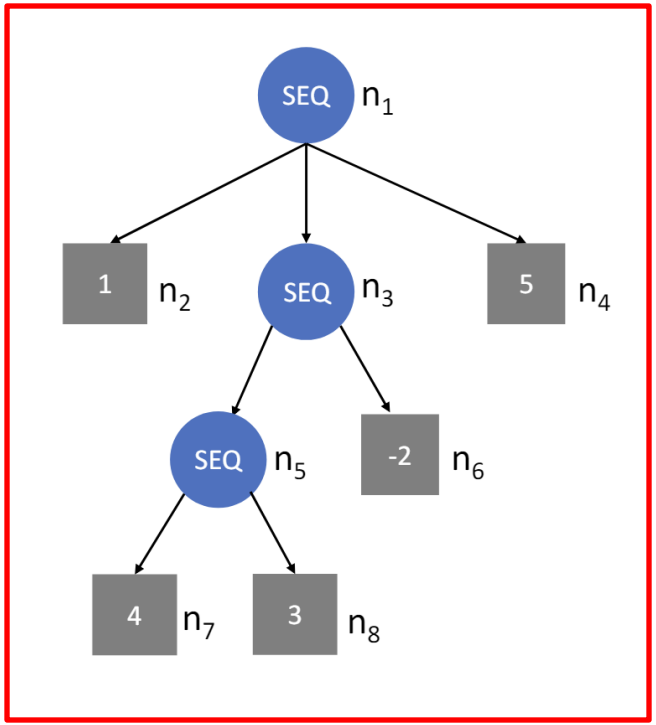
Probleme Lösen: Executable Graph

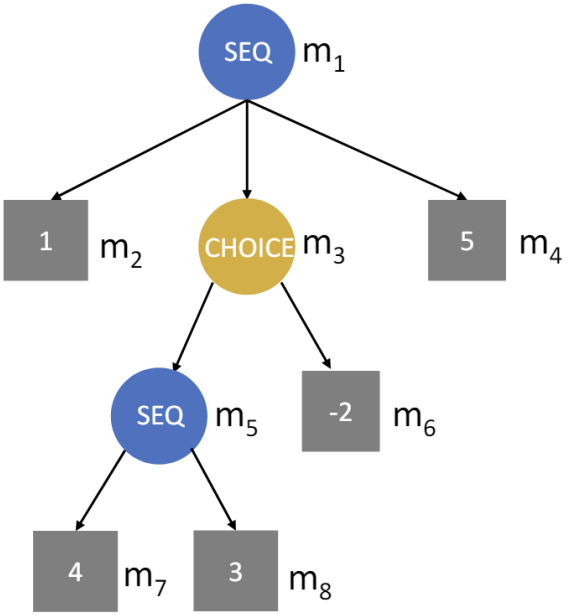
- **ADD Node:** Enthalten `value a` und wir setzen `(sum, counter)` auf `(sum + a, counter + 1)`. Kinderknoten werden bei der Ausführung ignoriert.

- **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.

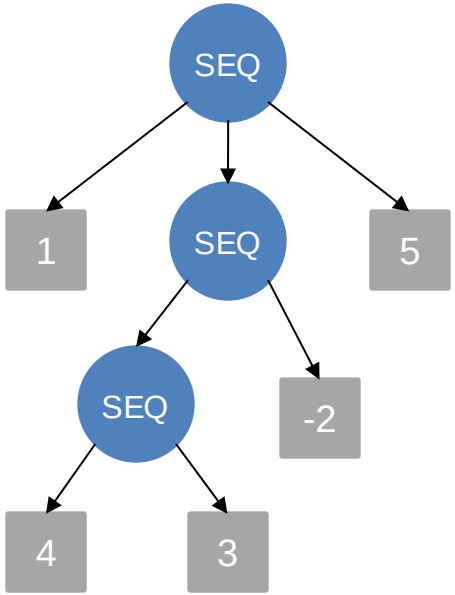
Probleme Lösen: Executable Graph





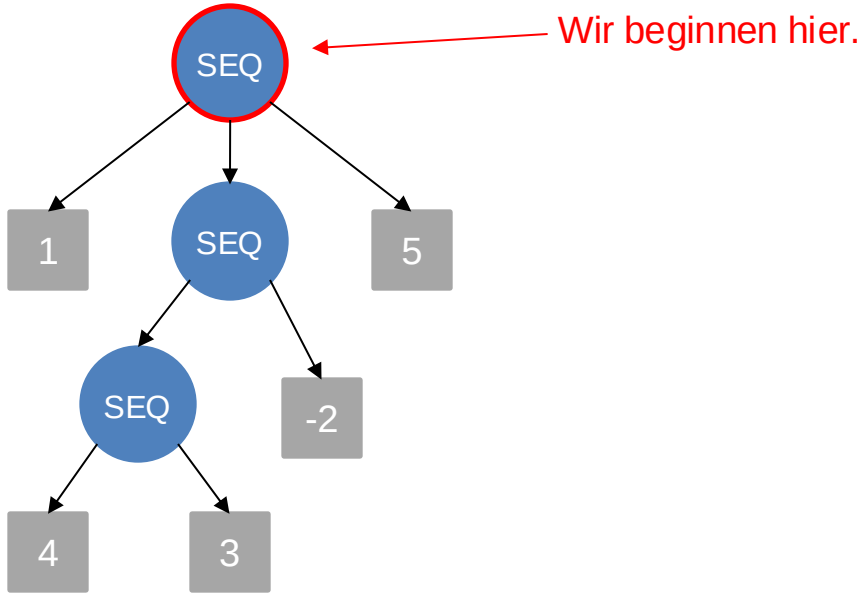
Probleme Lösen: Executable Graph

Startzustand: `(1,2)`

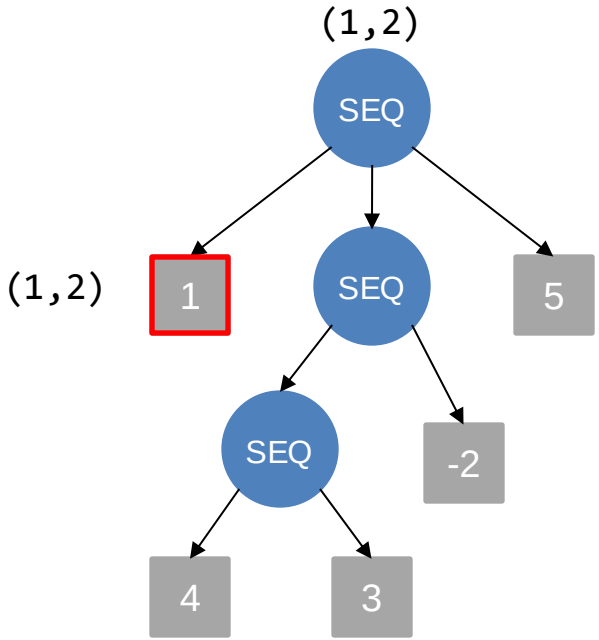


Probleme Lösen: Executable Graph

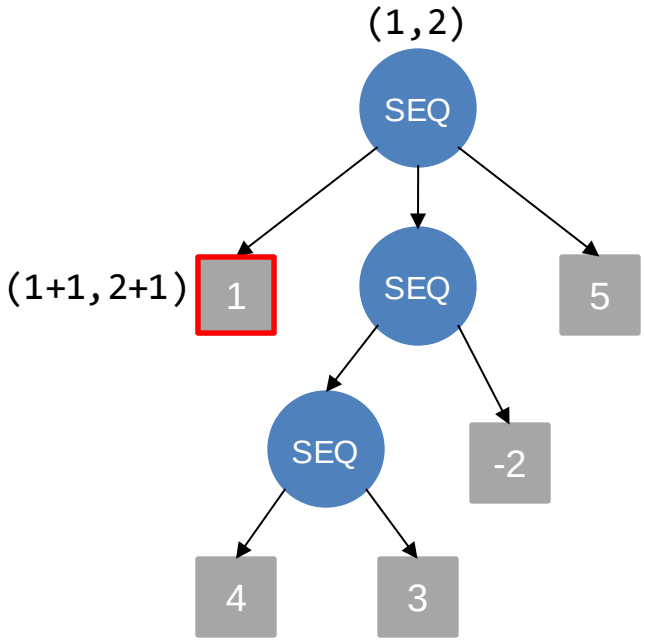
Startzustand: `(1,2)`



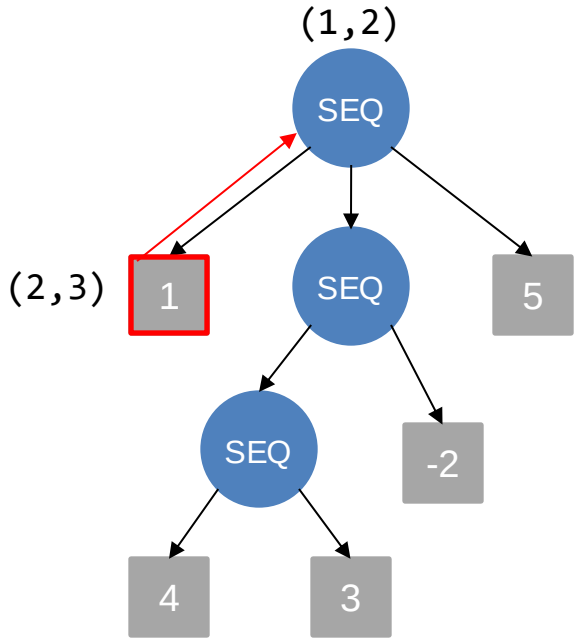
Probleme Lösen: Executable Graph



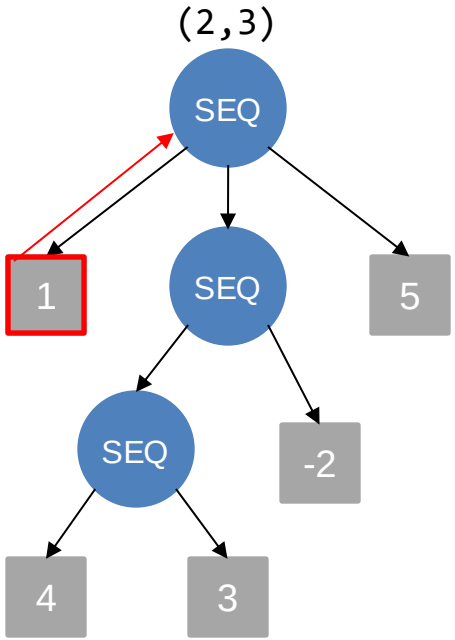
Probleme Lösen: Executable Graph



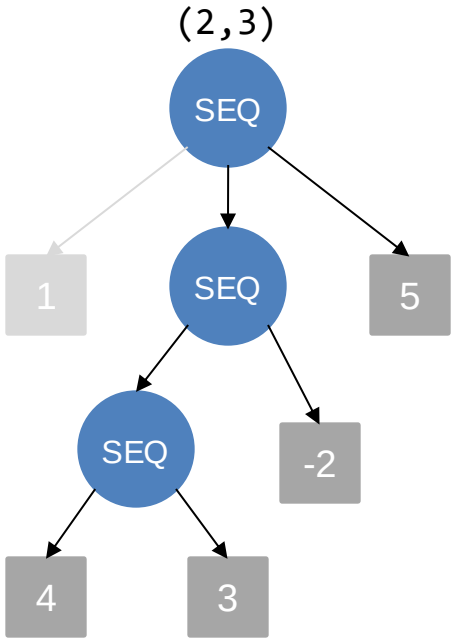
Probleme Lösen: Executable Graph



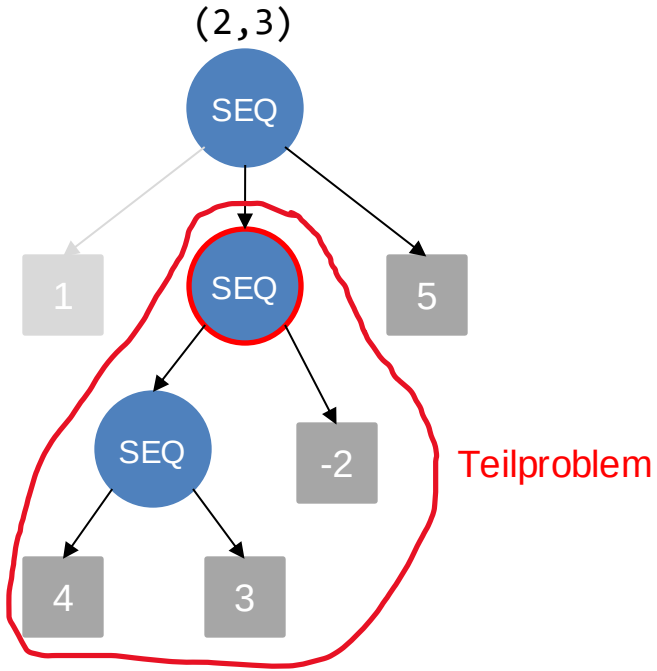
Probleme Lösen: Executable Graph



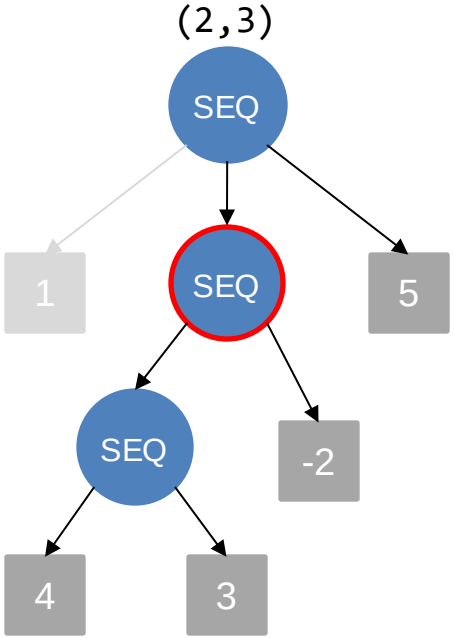
Probleme Lösen: Executable Graph

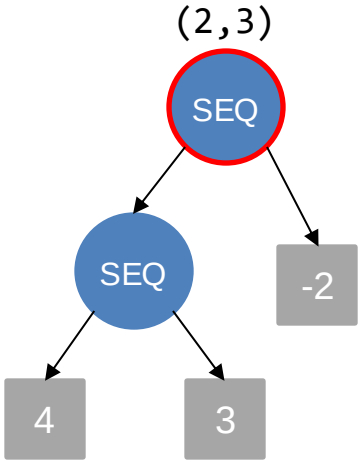


Probleme Lösen: Executable Graph

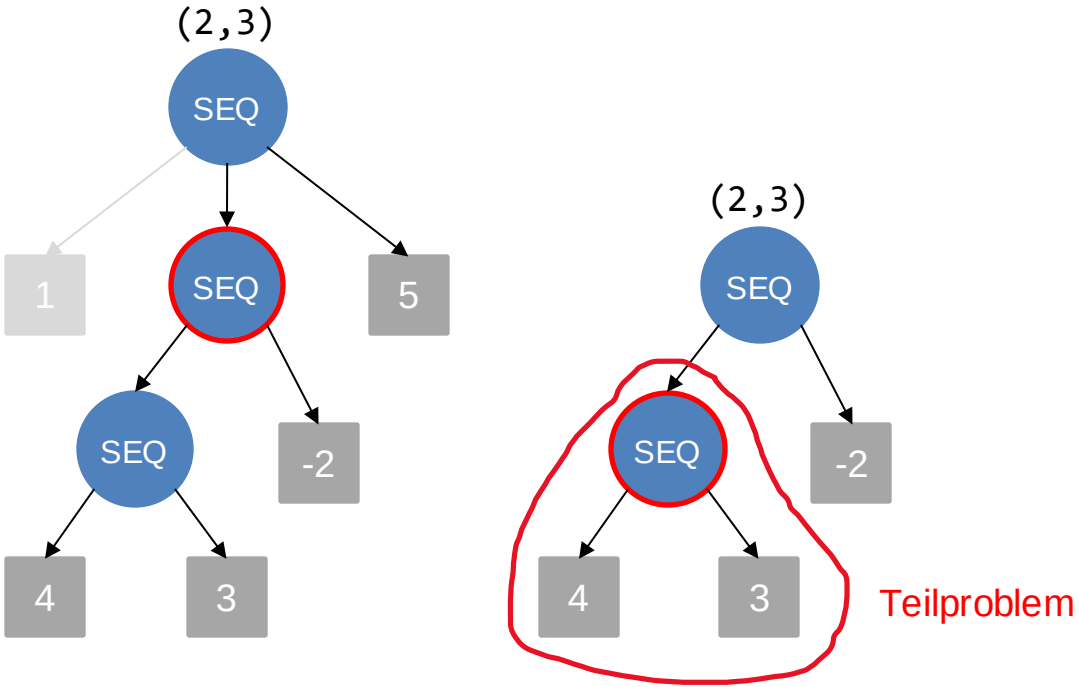


Probleme Lösen: Executable Graph

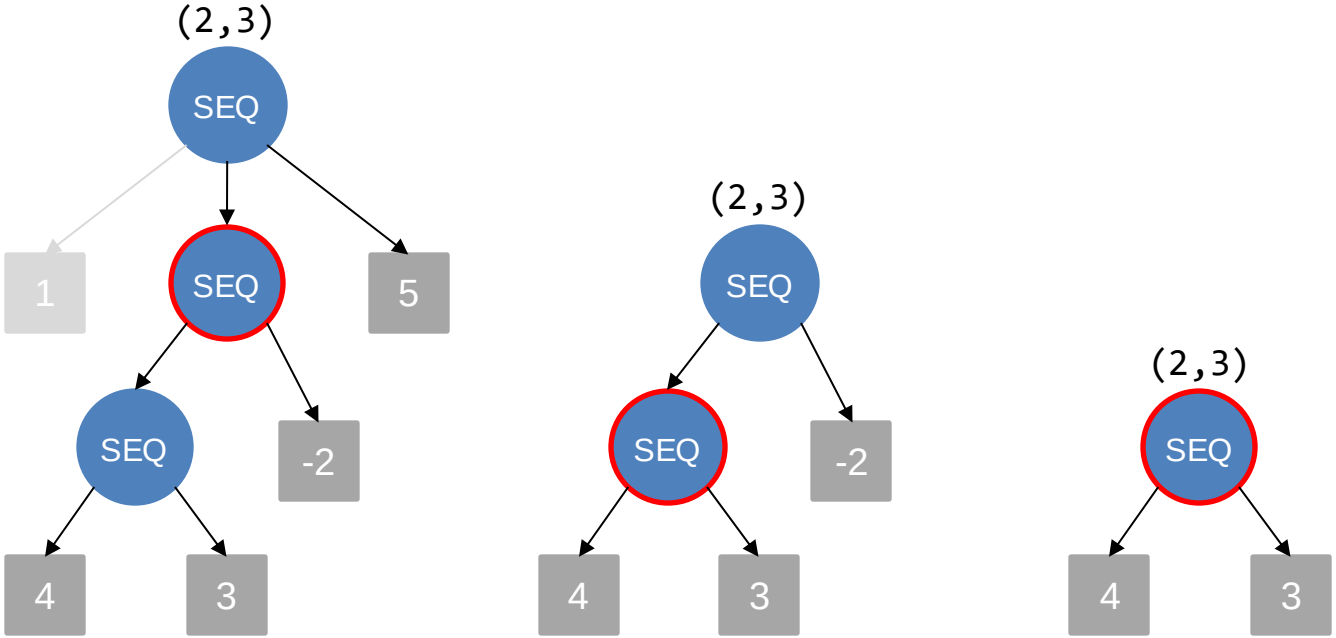




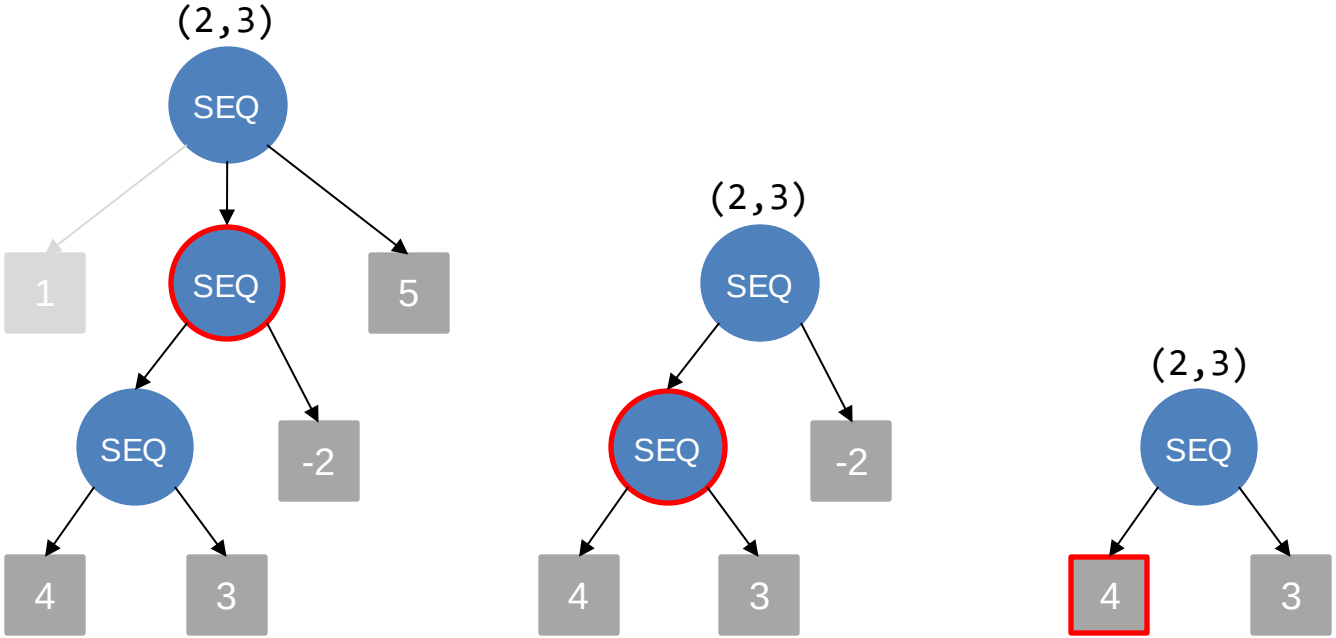
Probleme Lösen: Executable Graph



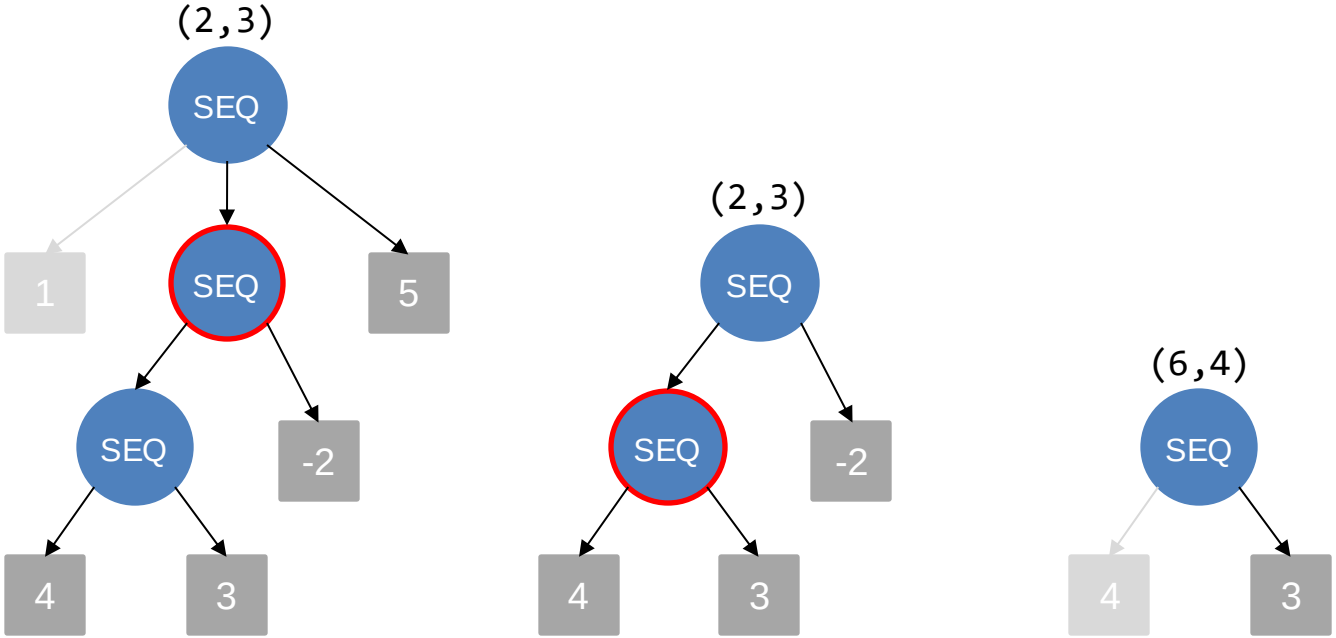
Probleme Lösen: Executable Graph



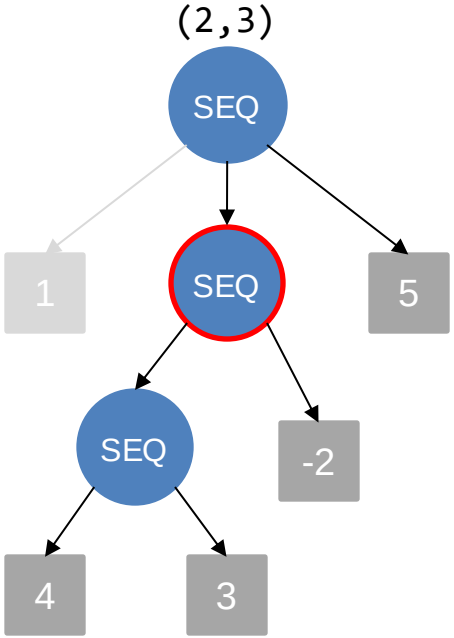
Probleme Lösen: Executable Graph

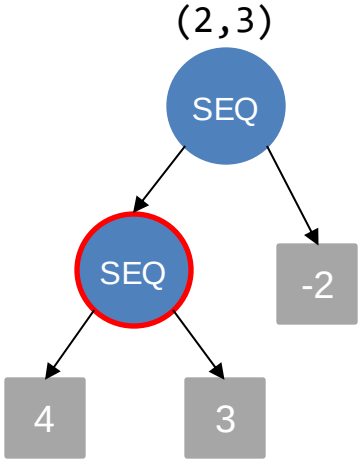


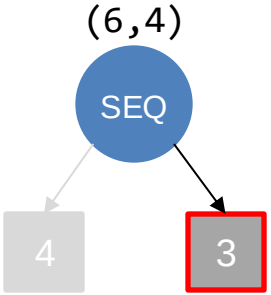
Probleme Lösen: Executable Graph



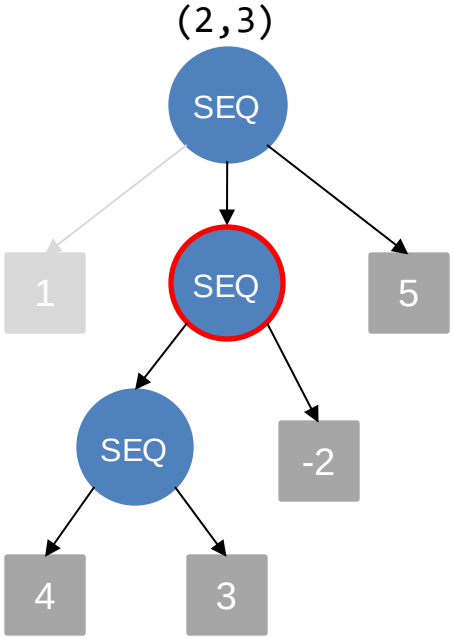
Probleme Lösen: Executable Graph

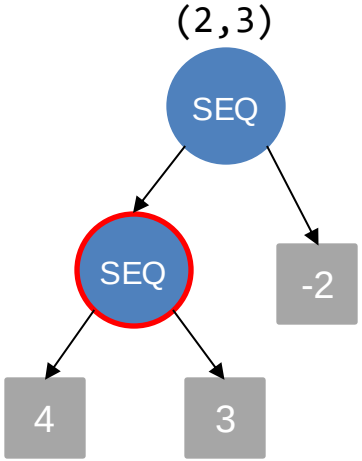


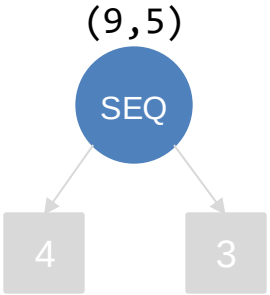




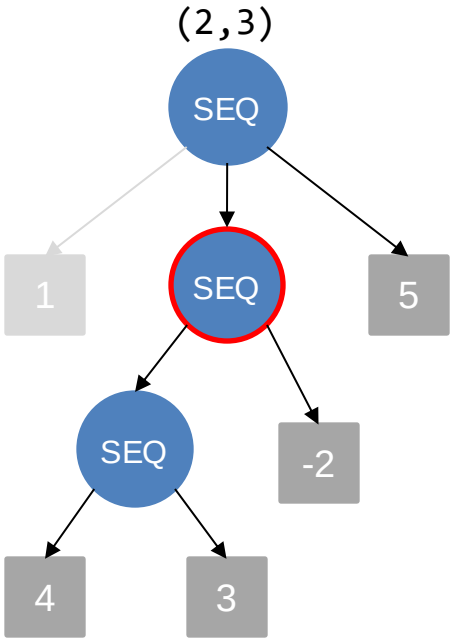
Probleme Lösen: Executable Graph

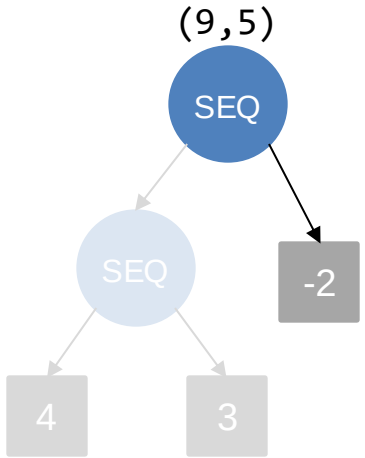




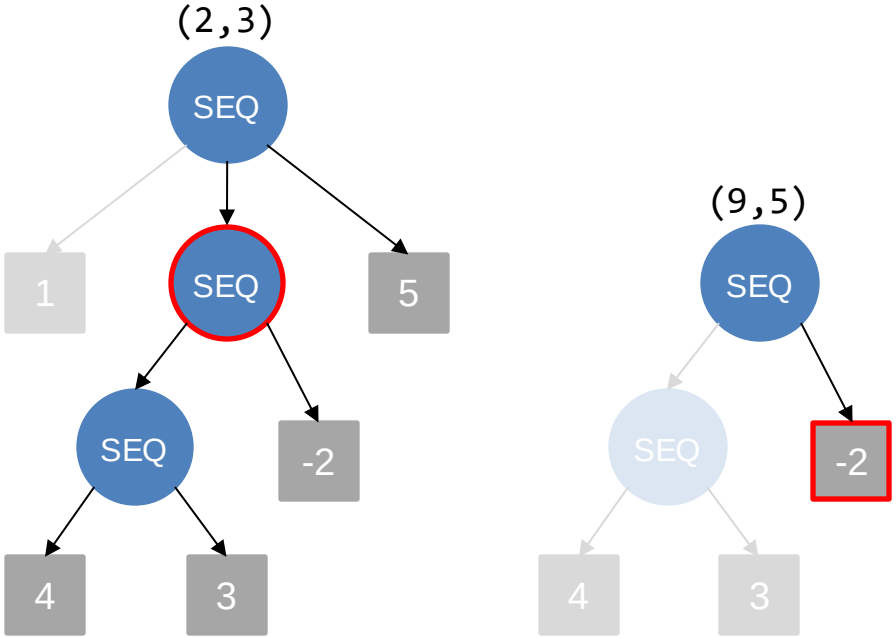


Probleme Lösen: Executable Graph

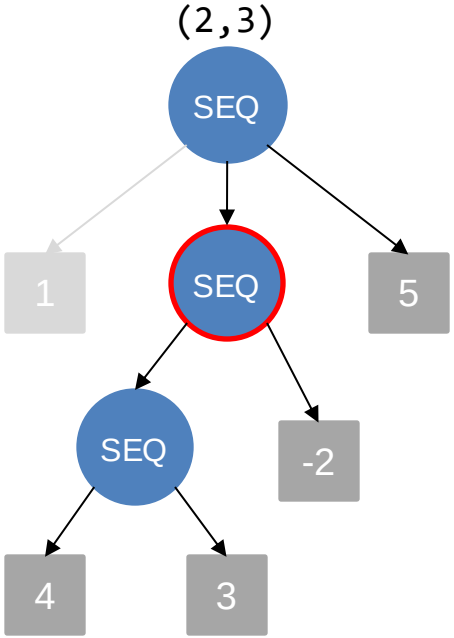


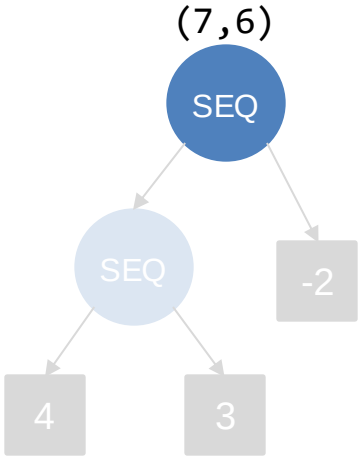


Probleme Lösen: Executable Graph

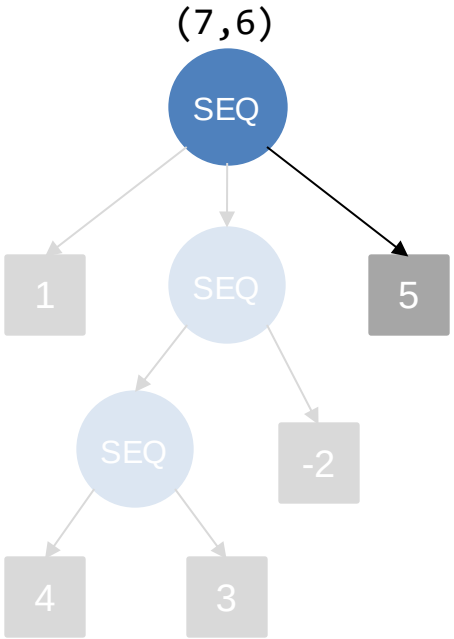


Probleme Lösen: Executable Graph

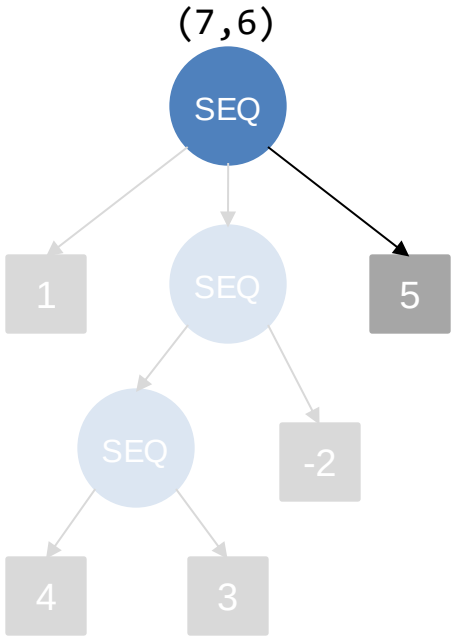




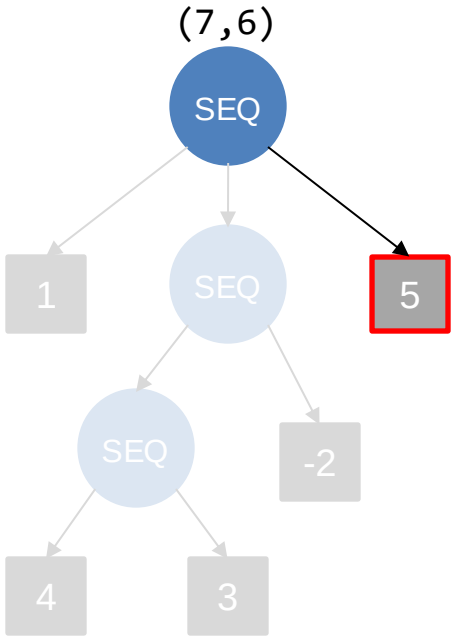
Probleme Lösen: Executable Graph



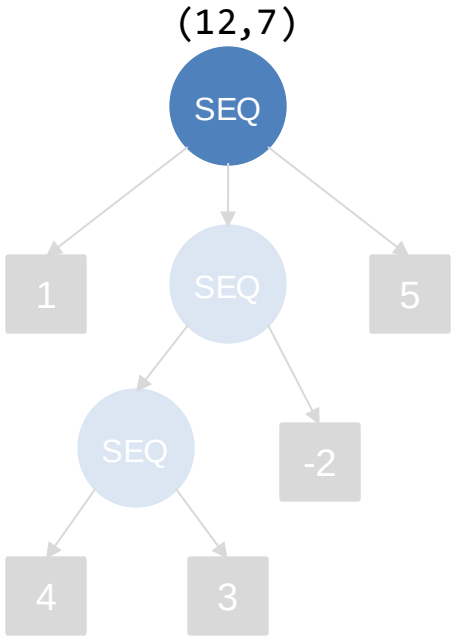
Probleme Lösen: Executable Graph



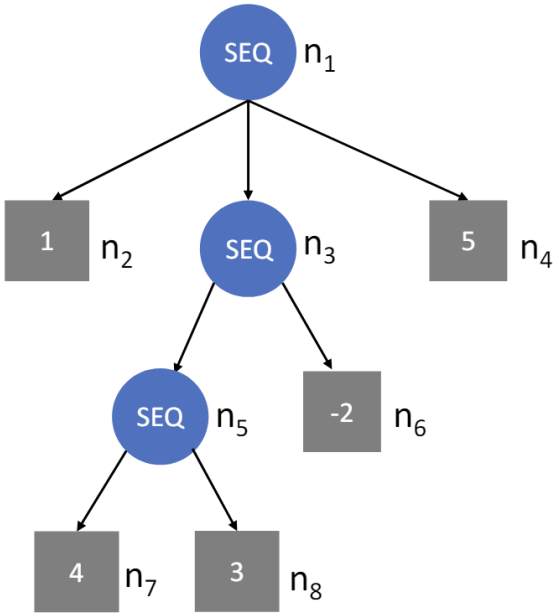
Probleme Lösen: Executable Graph

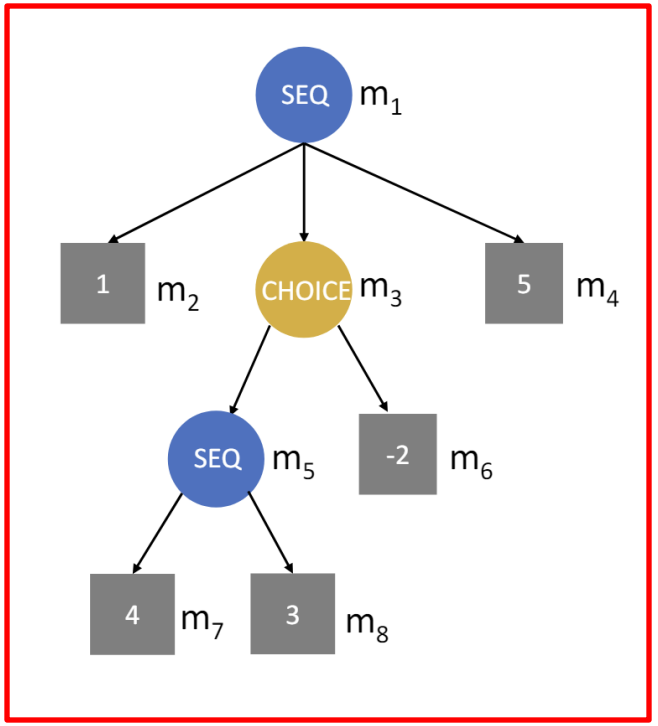


Probleme Lösen: Executable Graph

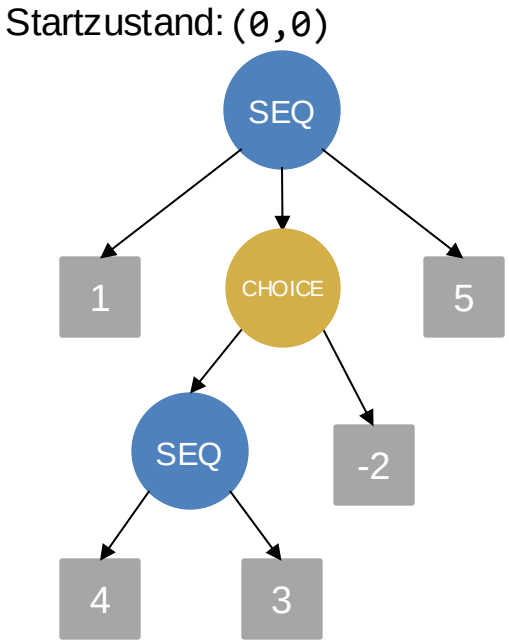


Probleme Lösen: Executable Graph

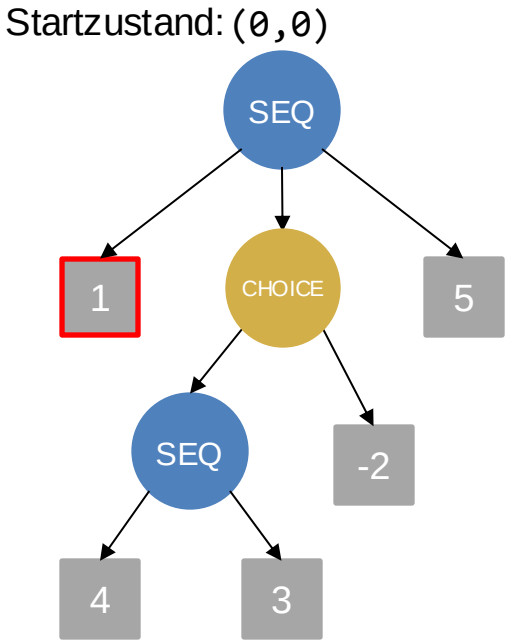




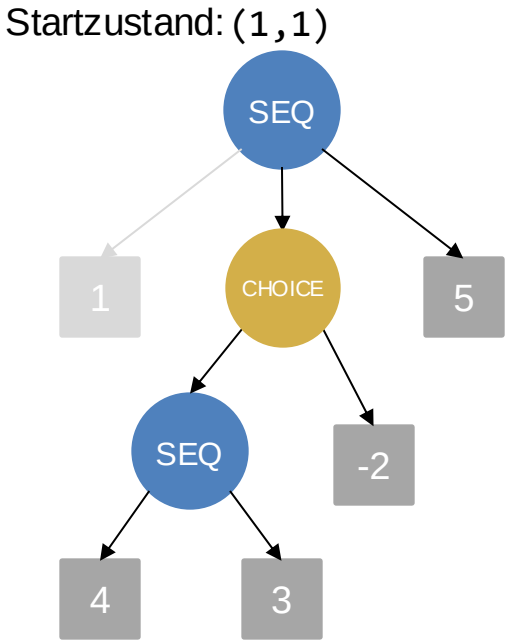
Probleme Lösen: Executable Graph



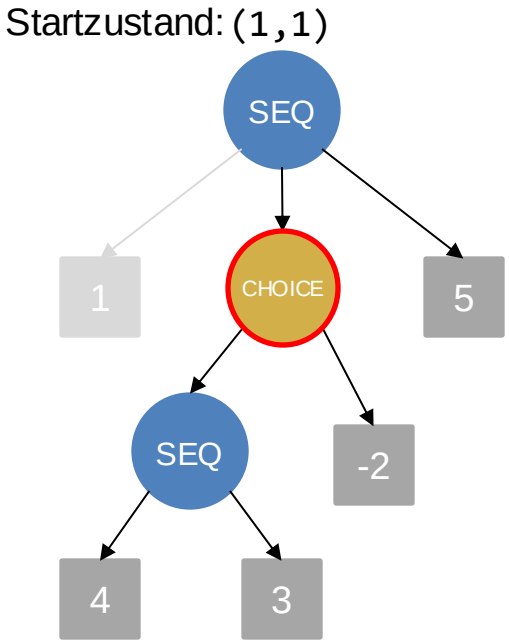
Probleme Lösen: Executable Graph



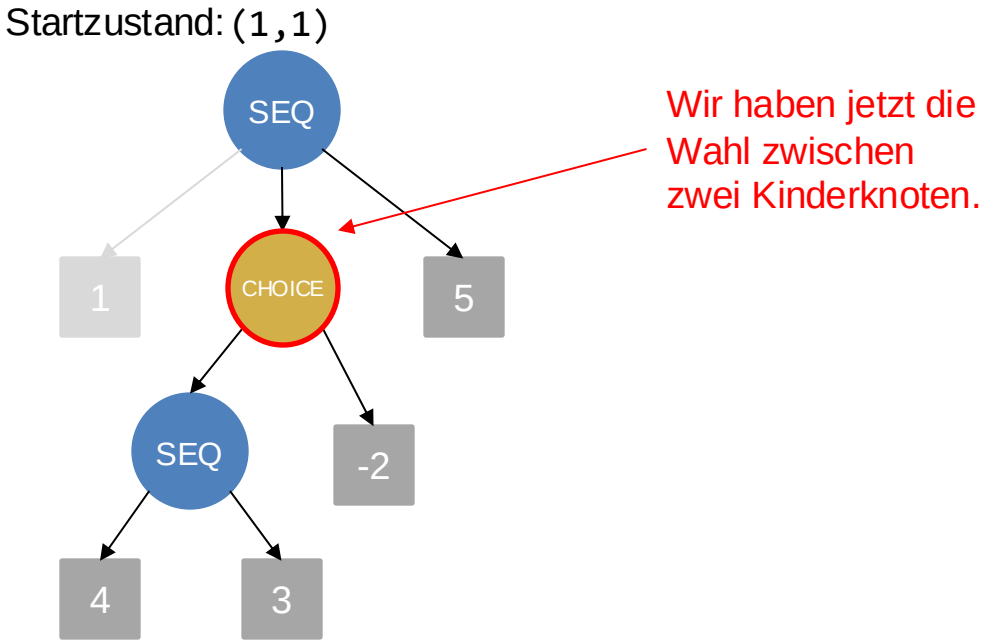
Probleme Lösen: Executable Graph



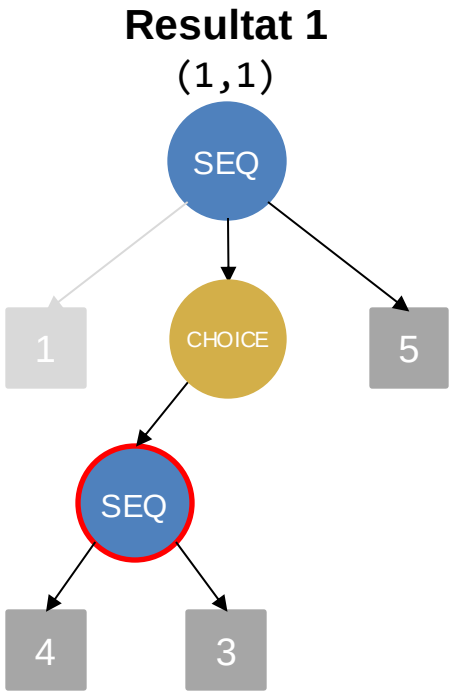
Probleme Lösen: Executable Graph

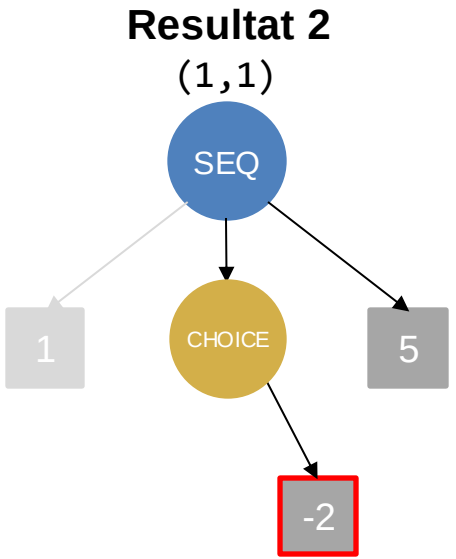


Probleme Lösen: Executable Graph



Probleme Lösen: Executable Graph

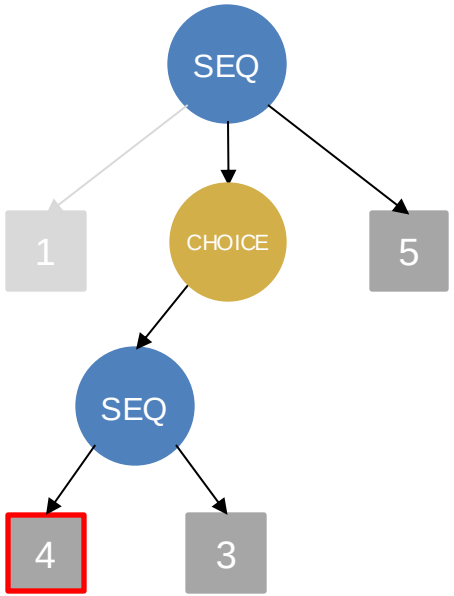




Probleme Lösen: Executable Graph

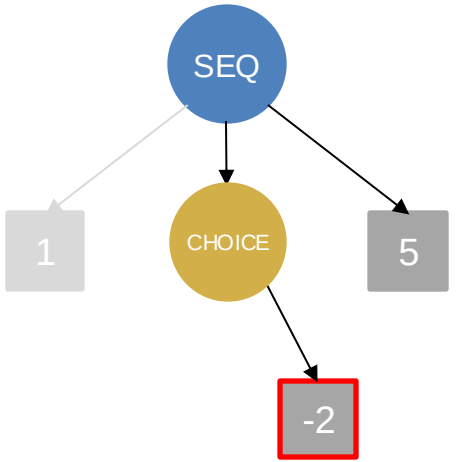
Resultat 1

(1,1)



Resultat 2

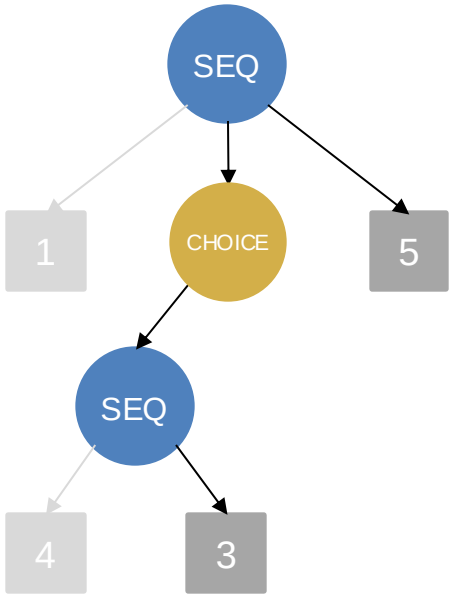
(1,1)



Probleme Lösen: Executable Graph

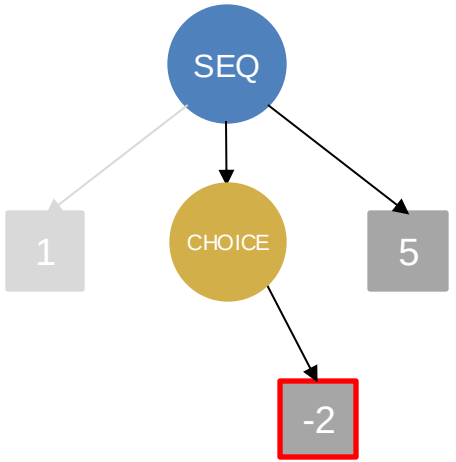
Resultat 1

(5,2)

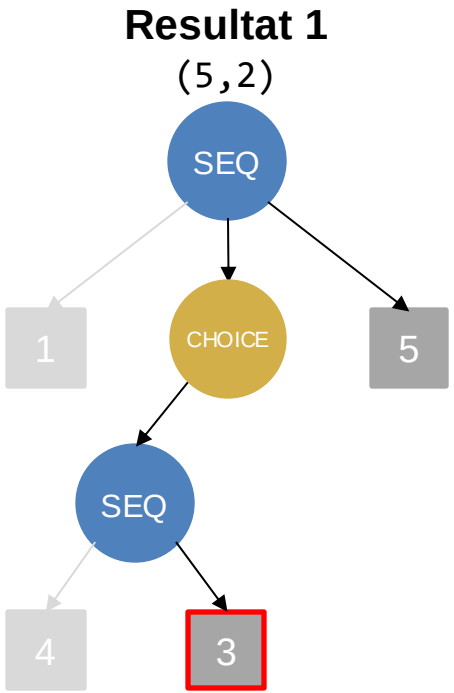


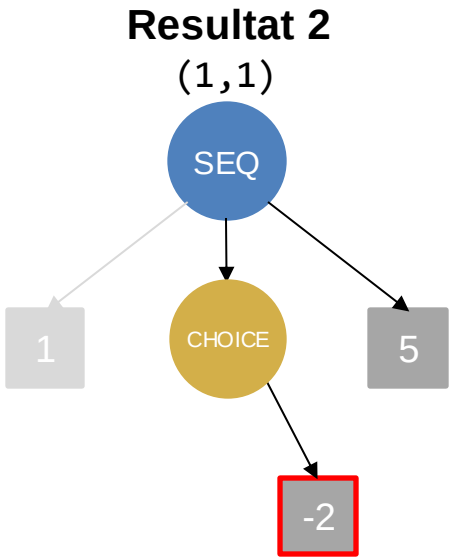
Resultat 2

(1,1)

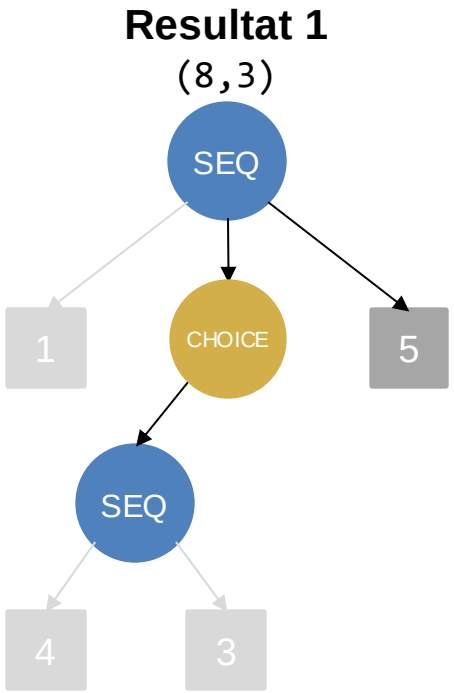


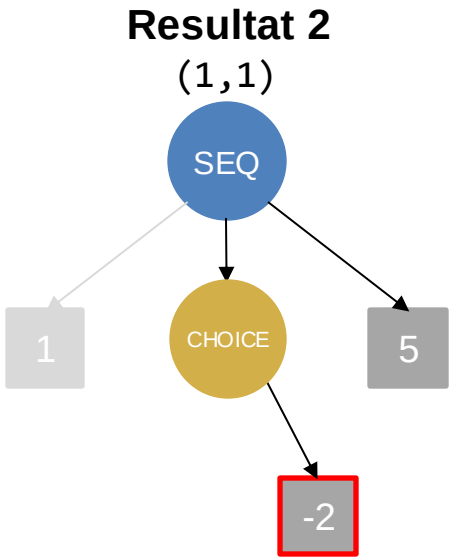
Probleme Lösen: Executable Graph





Probleme Lösen: Executable Graph

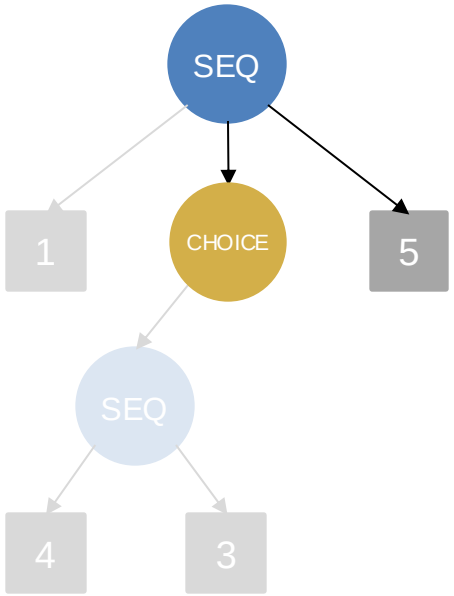




Probleme Lösen: Executable Graph

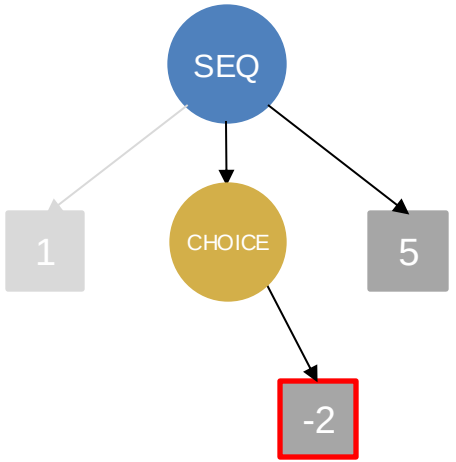
Resultat 1

(8,3)



Resultat 2

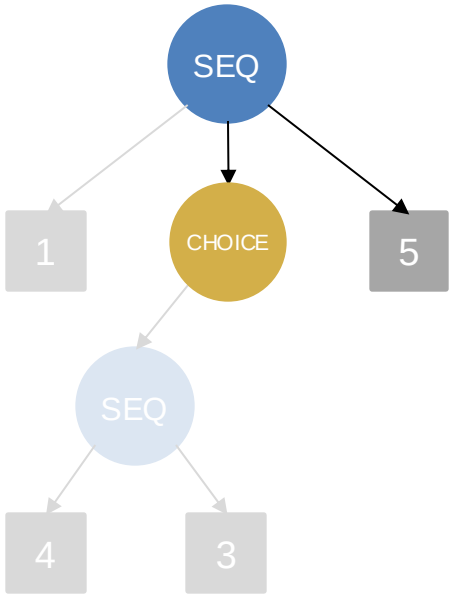
(1,1)



Probleme Lösen: Executable Graph

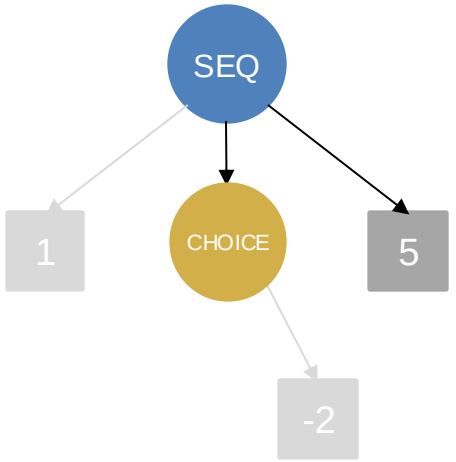
Resultat 1

(8,3)



Resultat 2

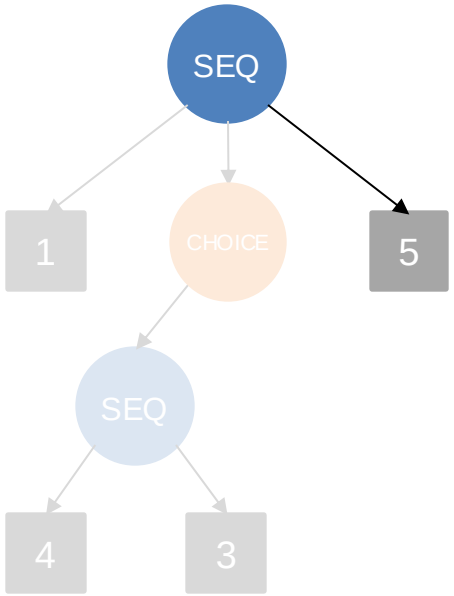
(-1,2)



Probleme Lösen: Executable Graph

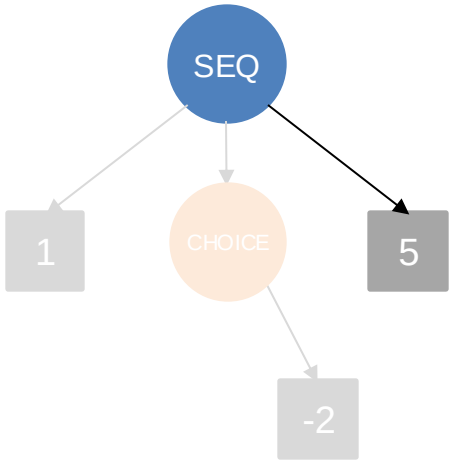
Resultat 1

(8,3)

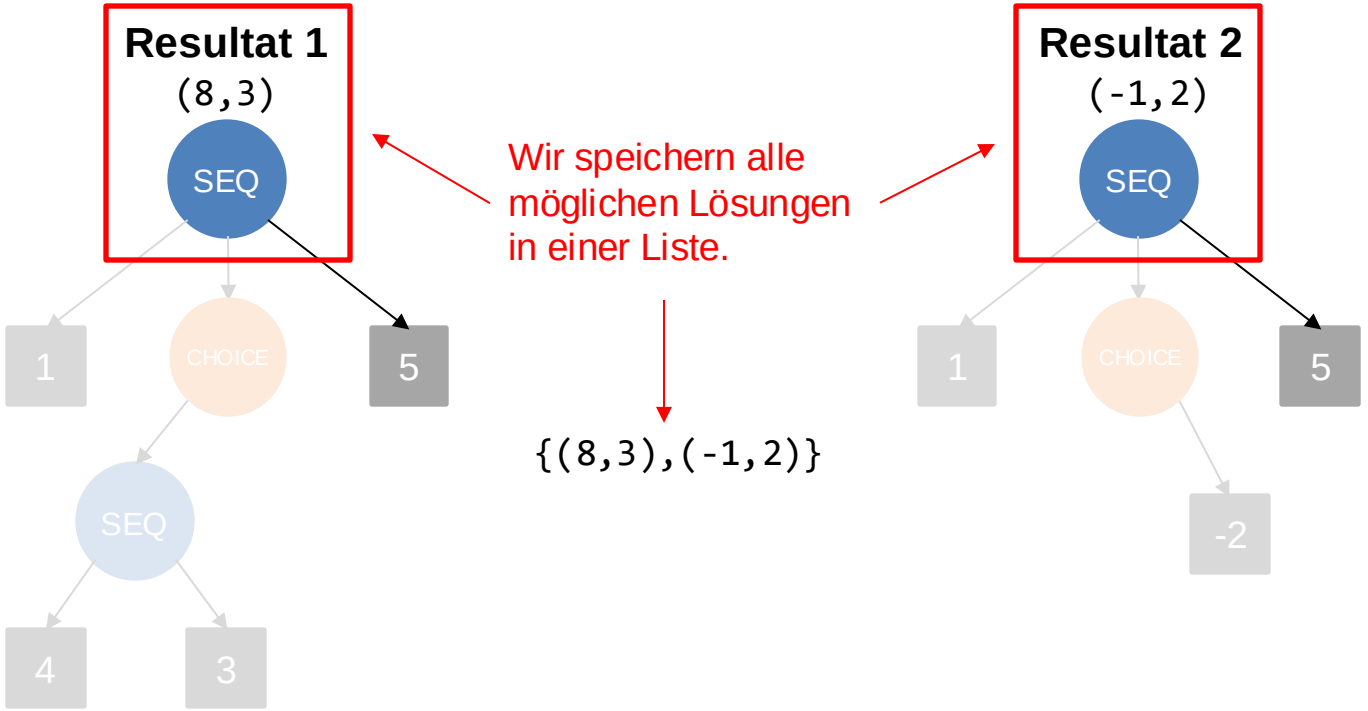


Resultat 2

(-1,2)

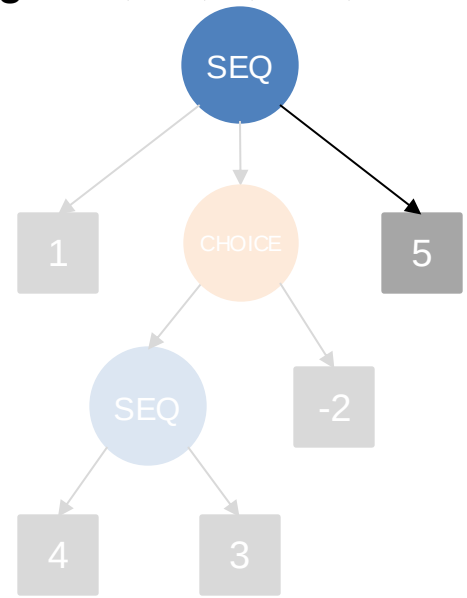


Probleme Lösen: Executable Graph



Probleme Lösen: Executable Graph

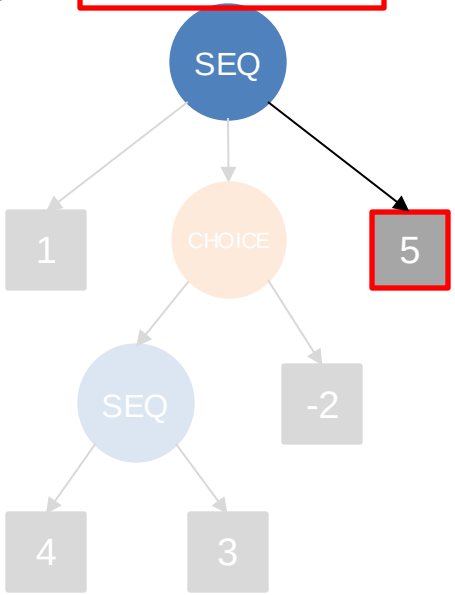
Lösungen: `{(8,3),(-1,2)}`



Probleme Lösen: Executable Graph

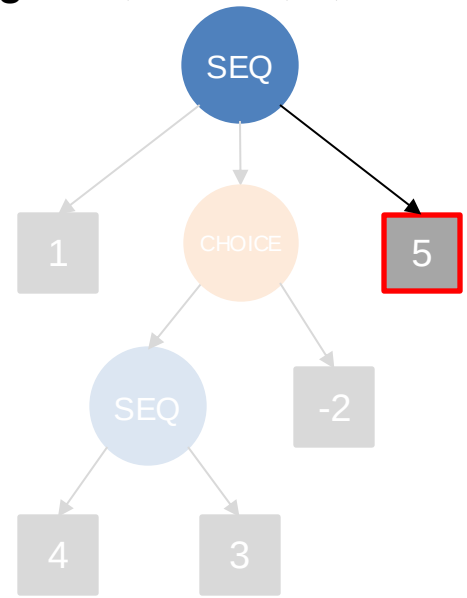
Lösungen: `{(8,3),(-1,2)}`

Wir müssen jedes Zwischenresultat in der Liste updaten.



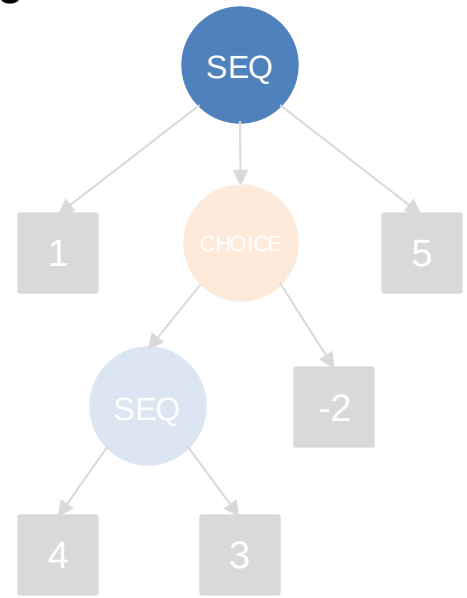
Probleme Lösen: Executable Graph

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.

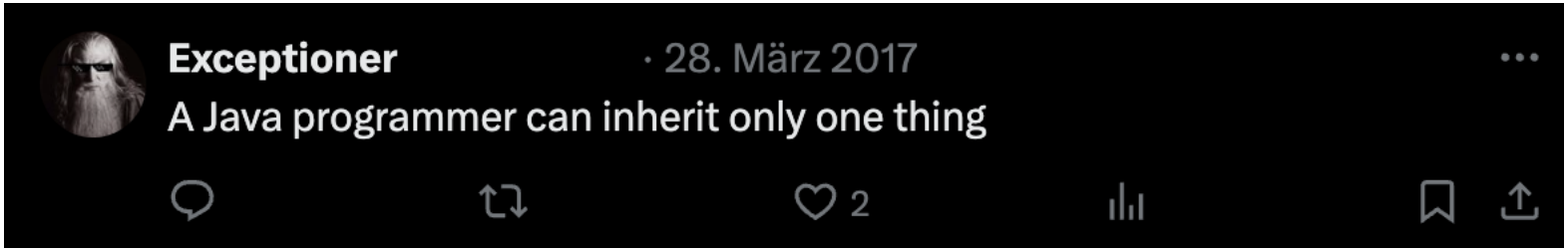
```
public static LinkedProgramStateList allResultsGo(Node n, LinkedProgramStateList states) {
(…)
    } else if (n.getType().equals("CHOICE")) {
        LinkedProgramStateList next = new LinkedProgramStateList();
        for (Node ch : n.getSubnodes()) {
            LinkedProgramStateList results =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

You have to use a new linked list for every recursive call

Just use a single linked list

```
private static void getResults(LinkedProgramStateList distinctRoutes, Node n) {
    switch (n.getType()) {
    case "ADD":
        for (ProgramState state : distinctRoutes.toArray()) {
            state.sum+=n.getValue();
            state.counter++;
        }
        break;
    case "CHOICE":
        ProgramState[] currentStates = distinctRoutes.toArray();
        distinctRoutes.clear();
        for (Node sn : n.getSubnodes()) {
            LinkedProgramStateList list = LinkedProgramStateList.of(new ProgramState(0, 0));
            getResults(list, sn);
            for (ProgramState currentState : currentStates) {
                for (ProgramState newState : list.toArray()) {
                    int sum = currentState.getSum() + newState.getSum();
                    int counter = currentState.getCounter() + newState.getCounter();
                    distinctRoutes.addLast(new ProgramState(sum, counter));
                }
            }
        }
        break;
    case "SEQ":
        for (Node sn : n.getSubnodes()) {
            getResults(distinctRoutes, sn);
        }
        break;
    }
}
```



Vererbung in Java

Was wird vererbt?

- Fields (gleiche Felder in Superklasse werden versteckt)
- Instance Methods (gleiche Methoden in Superklasse werden überschrieben)
- Überschreiben von Methoden wird auch Runtime Polymorphismus genannt
- Überladen von Methoden wird Compile-Time Polymorphismus genannt
- Static Methods (gleiche Methoden in Superklasse werden versteckt)
- Methoden / Felder werden nur verebt, wenn sie public bzw. protected sind

```java
// A ist Subclass B und B ist Subclass C und C ist Subclass D
A a = new A();
B b = new B();

C bc = b;
bc = a;

D d = bc;
((A) d).methodFromA();
```

```java
ArrayList<Integer> coolList = new ArrayList<>();
ArrayList<Object> froozenList = (ArrayList<Object>) coolList;

Integer[] coolArray = new Integer[3];
Object[] frozenArray = coolArray;
frozenArray[0] = new ArrayList<Integer>();
```

Compile-Time Error

Runtime Exception aber kompiliert

Was ist eigentlich super in Java?

```java
class A {
    void method1() {
        System.out.println("A");
        this.method2(); // dynamic dispatch
    }

    void method2() {
        System.out.println("A");
    }
}

class B extends A {
    @Override
    void method2() {
        System.out.println("B");
        super.method2(); // was ist in diesem Kontext das super keyword?
    }
}

class C extends B {
    @Override
    void method2() {
        System.out.println("C");
        super.method2()
    }
}

public class Test {
    public static void main(String[] args) {
        A obj = new C();
        obj.method1();
    }
}
```

super Keyword ist immer die Superklasse der aktuellen Klasse (da wo die Methode definiert ist) super ist abhängig von der Code Struktur. Somit ist das Verhalten von super lexikalisch (ohne Berücksichtigung des Zusammenhangs)

```
class SchwanzLurche extends Lurche {
   void info() {
      System.out.print("Salamander&Molche ");
   }
   void zuerst() {
      // Larven entwickeln ...
      System.out.println("Arme zuerst");
   }
}

class FroschLurche extends Lurche {
   public String toString() {
      return "FroschLurche " + super.toString();
   }
   void zuerst() {
      // Larven entwickeln ...
      System.out.println("Beine zuerst");
   }
}

class Lurche extends Amphibien  {
    void info() {
       super.info();
       System.out.print("Amphibien ");
    }

    void zuerst() {
    }

    public String toString() {
       return "Lurche ";
    }


}
```

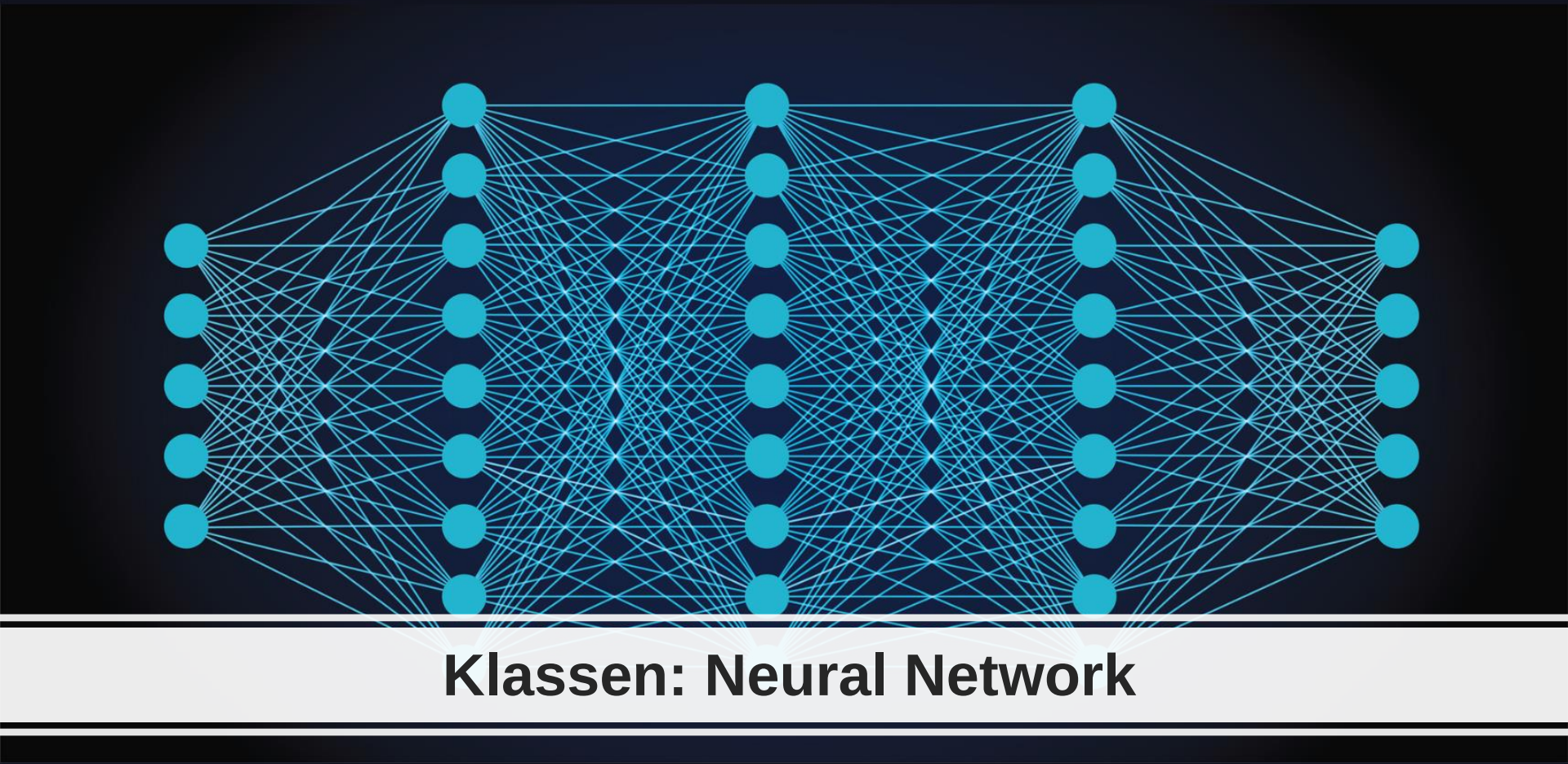
```
class Amphibien {
   void info() {
      System.out.print("Amphibien ");
   }

   void status() {
      System.out.print("bedroht ");
   }

   public String toString() {
      return "Amphibien ";
   }
}
```

1. Schritt: Überblick verschaffen, welche Methoden gibt es?
2. Schritt: Welche Methoden werden überschrieben?
3. Schritt: Welche Felder werden versteckt? (bei diesem Bsp. nicht relevant)

```
Amphibien[] meinTerrarium = { new Amphibien(), new Lurche(),
                              new FroschLurche(), new SchwanzLurche()};
for (int i = 0; i < meinTerrarium.length; i++) {
    System.out.println(meinTerrarium[i]);
    meinTerrarium[i].info();
    System.out.println();
    meinTerrarium[i].status();
    System.out.println();
}
```

Lösung

```
1  Amphibien
2  Amphibien
3  bedroht
4  Lurche
5  Amphibien Amphibien
6  bedroht
7  FroschLurche Lurche
8  Amphibien Amphibien
9  bedroht
10 Lurche
11 Salamander&Molche
12 bedroht
```

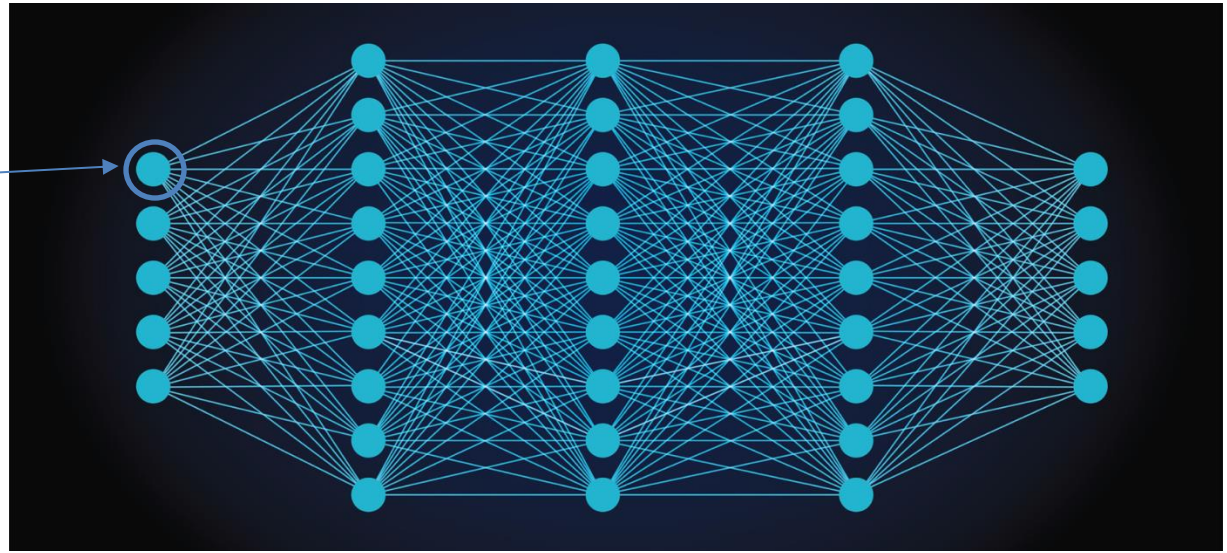


Klassen: Neural Network

Klassen: Neural Network

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

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

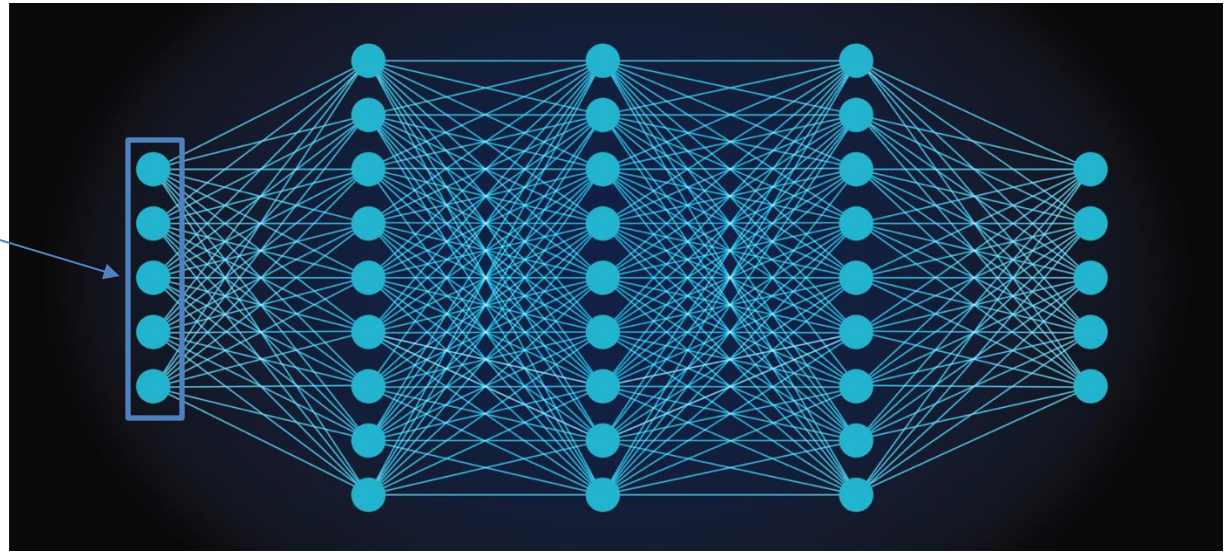


Klassen: Neural Network

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.

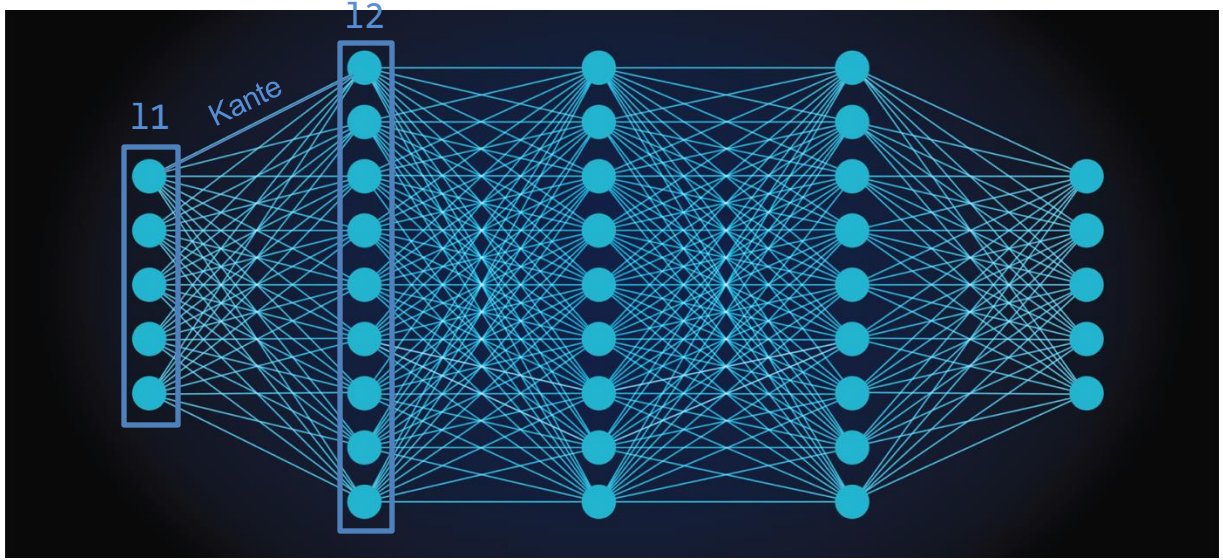


Klassen: Neural Network

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.

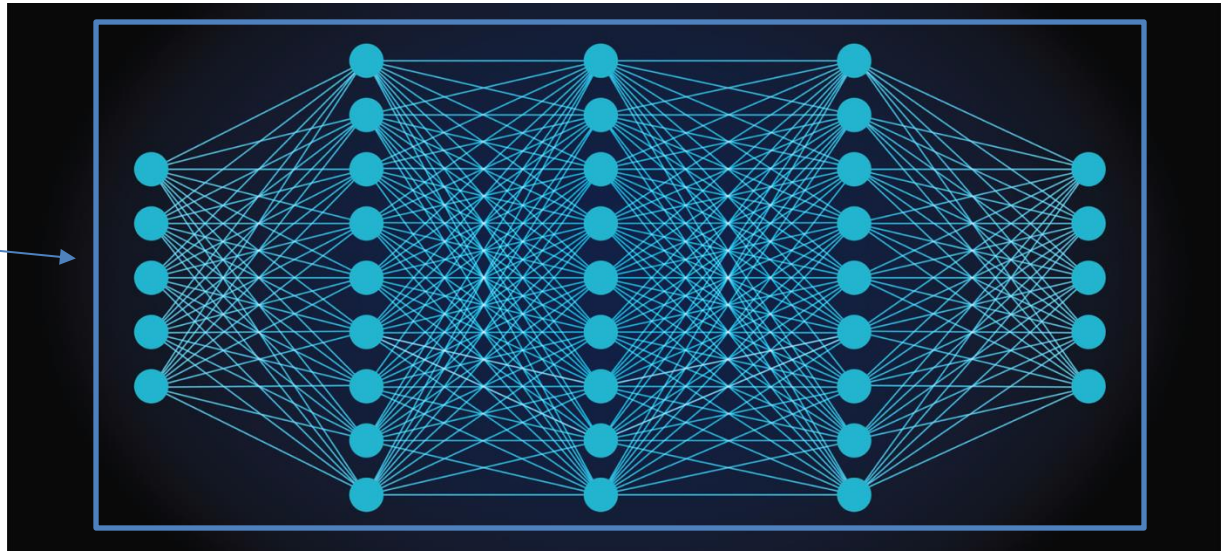


Klassen: Neural Network

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

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

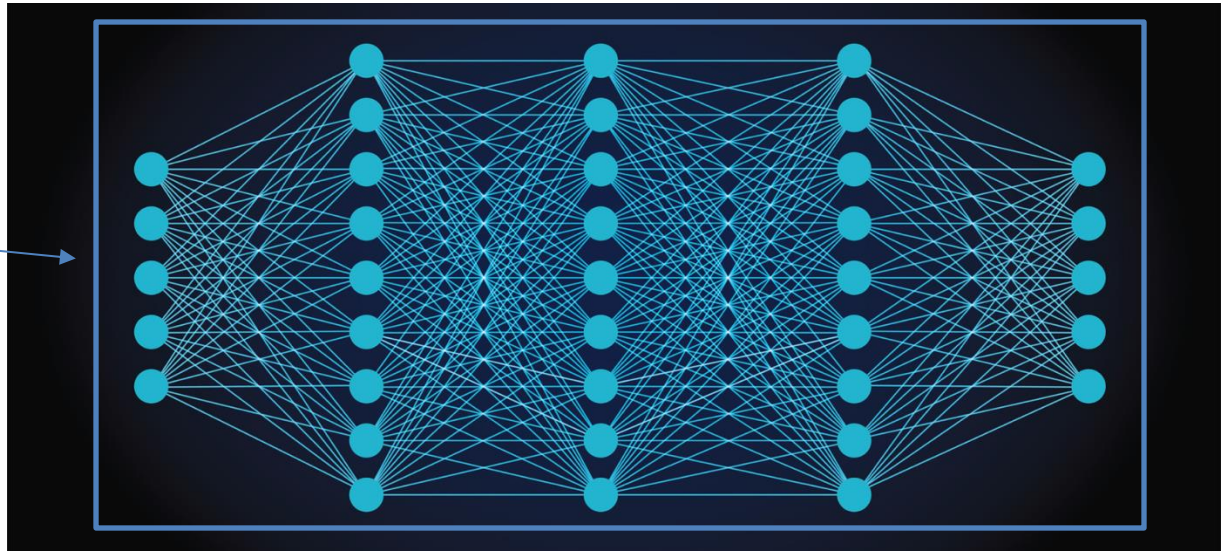
Das NN



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;
    }
}
```

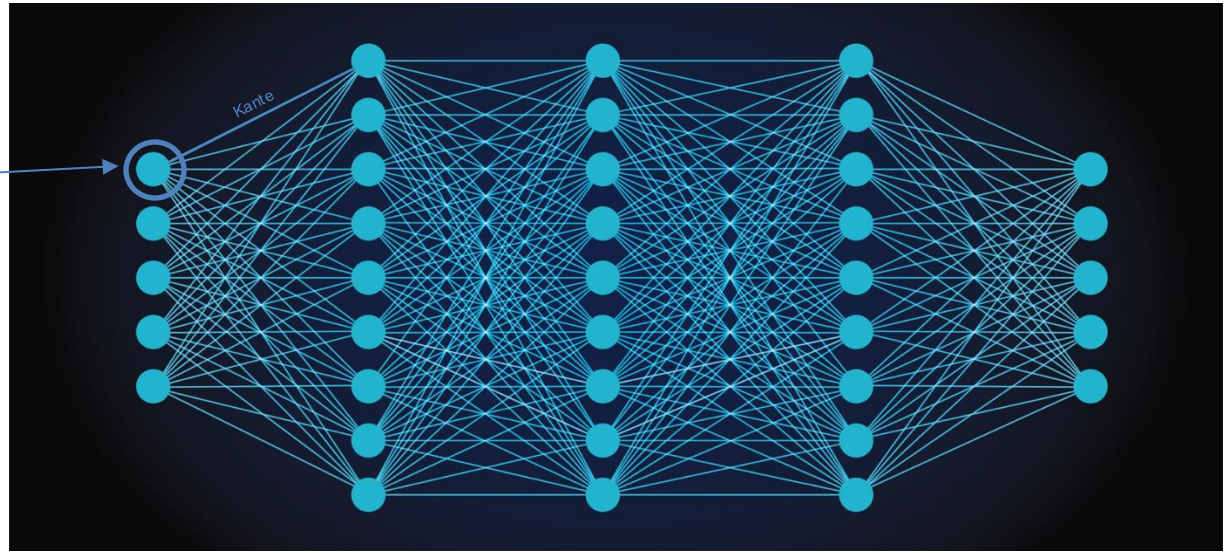
Layer.java

```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?

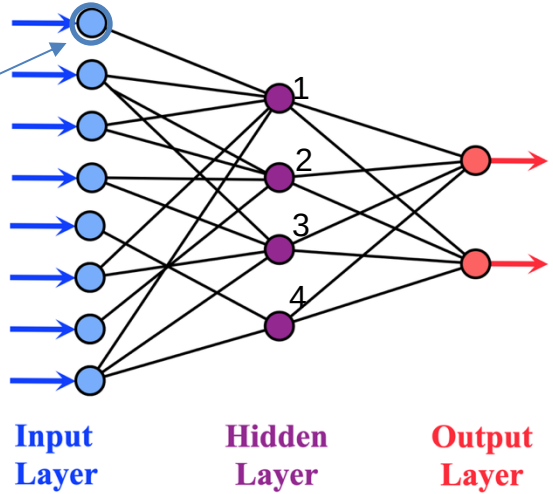


Klassen: Neural Network

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

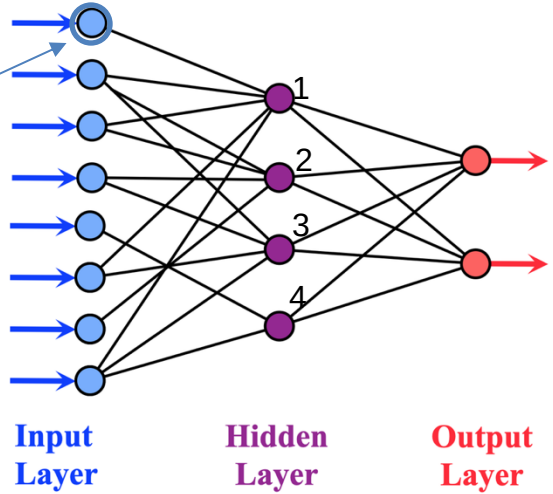


Klassen: Neural Network

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

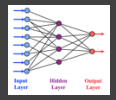
Ein `boolean` array pro Neuron würde reichen.

1	0	0	0
1	2	3	4



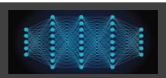
✨ **Vererbung!** ✨

```java
public class SparseNetwork extends NeuralNetwork {

    private boolean[][][] connections;

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

    public SparseNetwork() {
        this(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

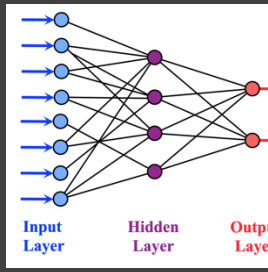
```java
public class Neuron {
    private int value;

    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

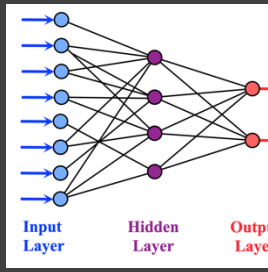
```java
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][][] connections;

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

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

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



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

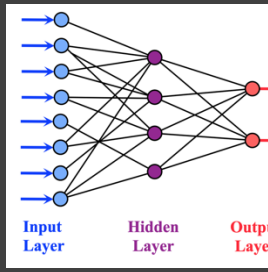
Bedeutung von `connections[i][j][k]`
`connections[i][j][k]` := Im Layer i gibt es von Node j eine Verbindung zu Layer (i+1) mit dem Node k

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

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

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

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



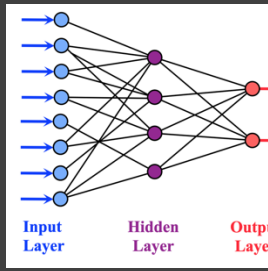
Die restlichen Attribute werden von NeuralNetwork geerbt.

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

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

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

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



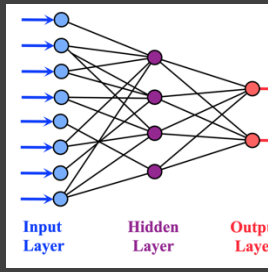
Konstruktoren werden nie geerbt!

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

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

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

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



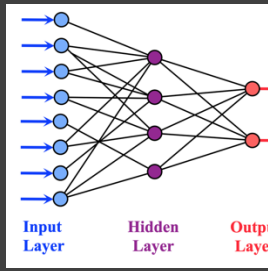
Wir rufen den Konstruktor der Superklasse und initialisieren das Connections-Array zusätzlich.

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

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

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

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

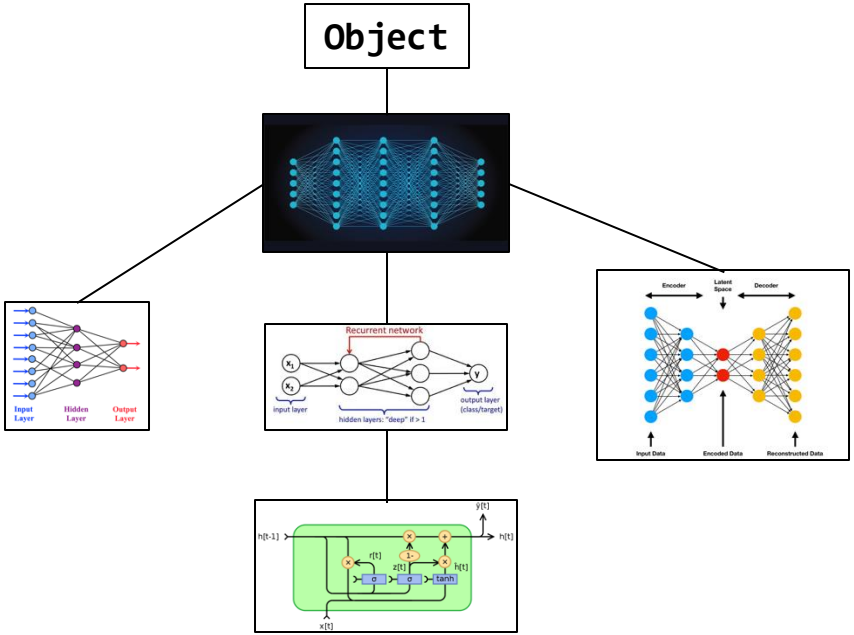


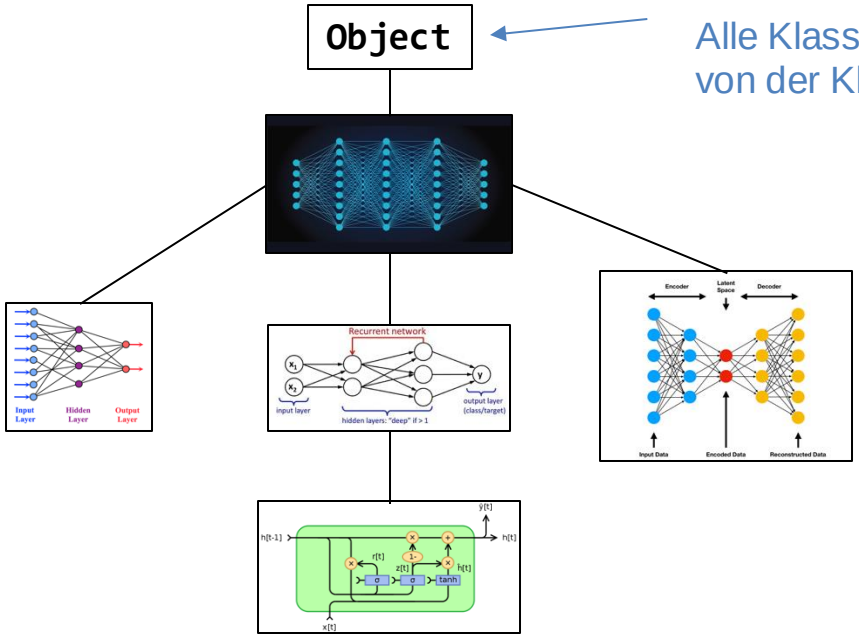
@Override stellt sicher, dass dies wirklich eine Überschreibung ist. Ansonsten gibt es einen Fehler bei der Ausführung.

`@Override` muss nicht geschrieben werden

Klassen: Neural Network

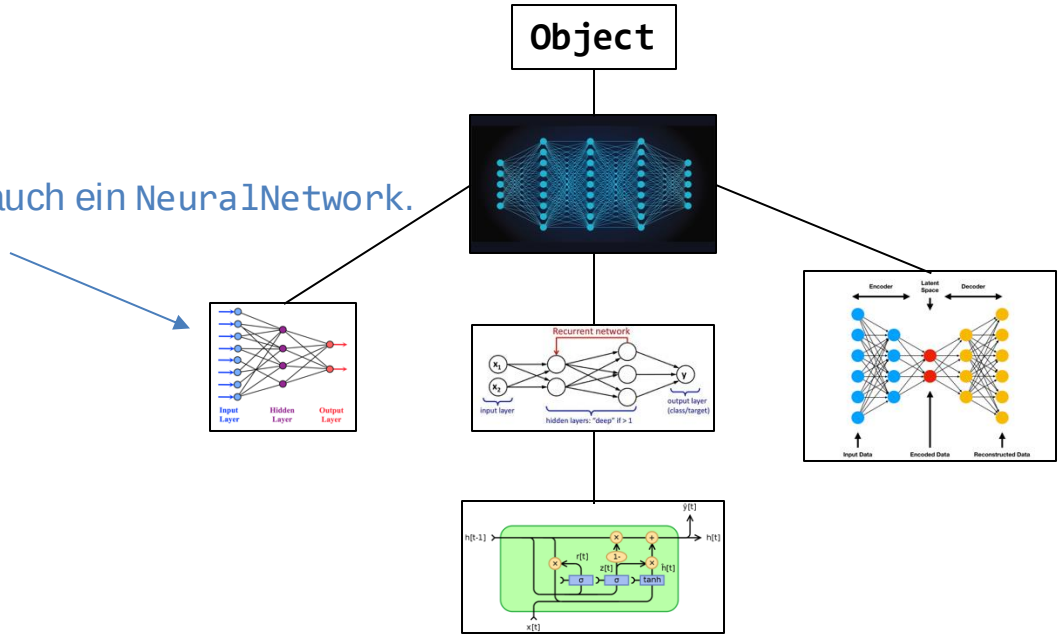
Wir könnten diverse neuronale Netzwerke so durch eine Klasse beschreiben…



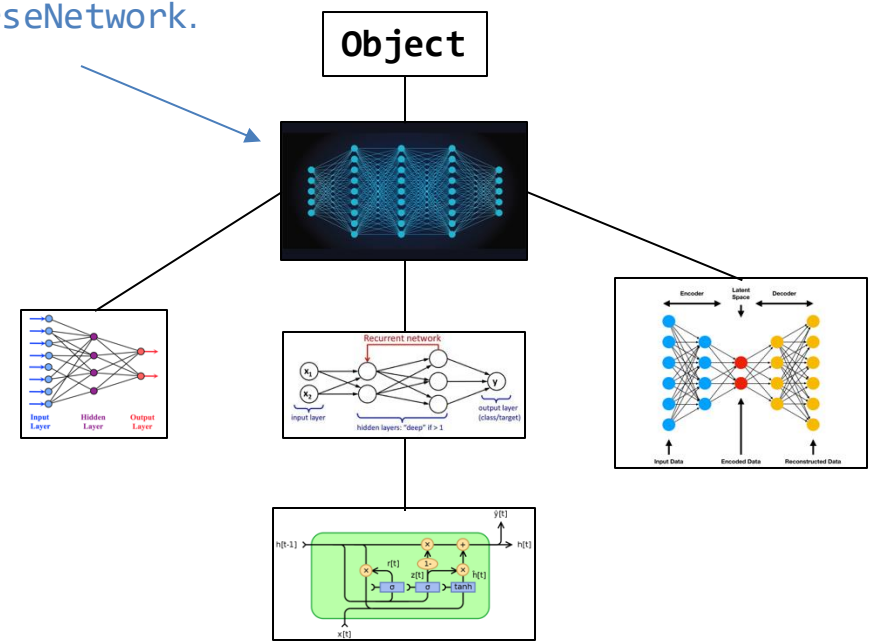


Alle Klassen sind Subklassen von der Klasse `Object`.

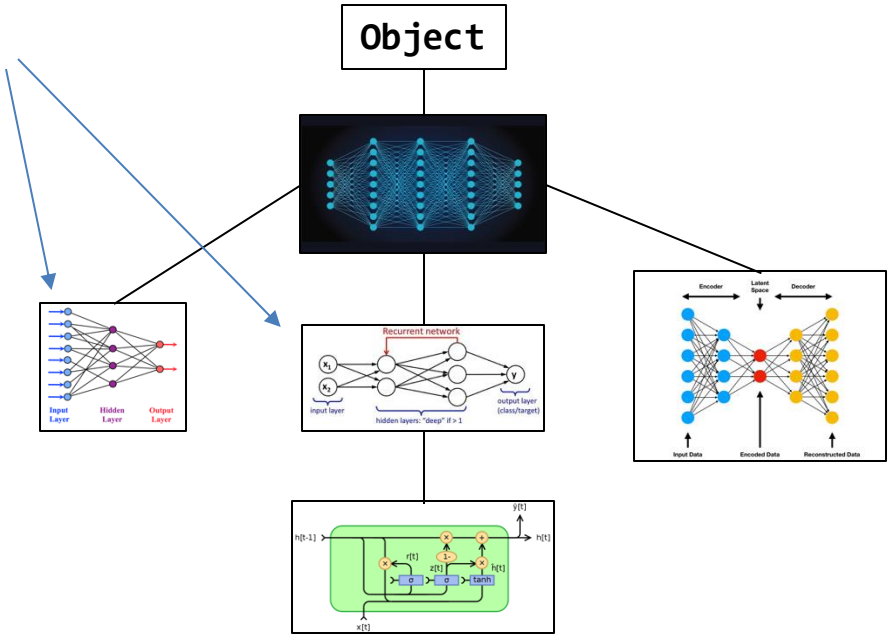
SparseNetwork ist auch ein NeuralNetwork.



NeuralNetwork ist kein SparseNetwork.



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



Teaser: 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 (erstellt neuen Array)	`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)`

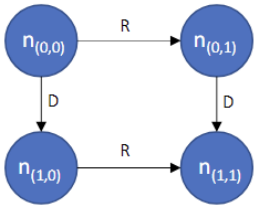
Vorbesprechung

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

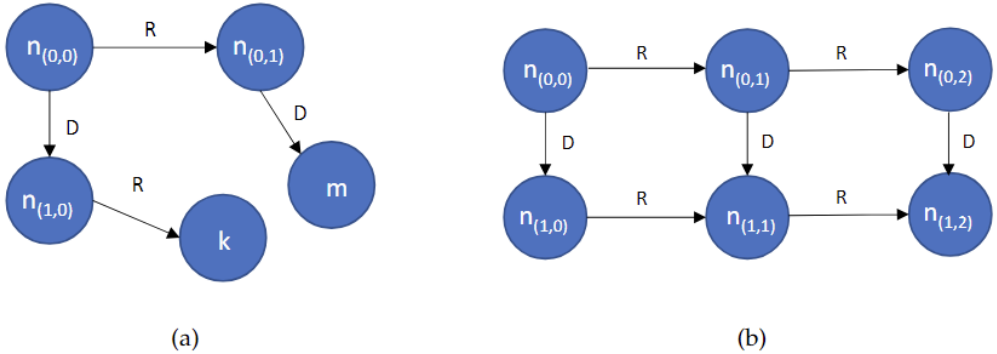


Abbildung 2: Graphen mit quadratischen Gittern als Teilgraphen

Referenzen vs Objekte

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.

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.

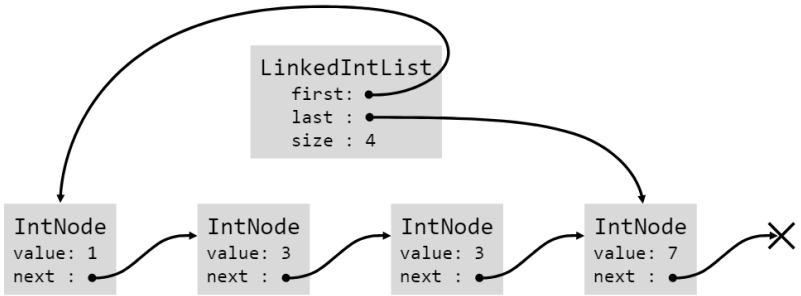


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: Linked List

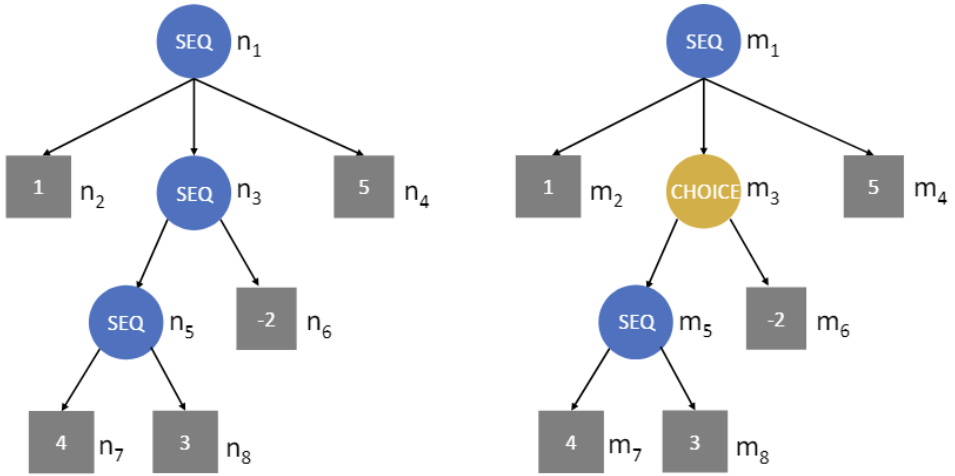
Name	Parameter	Rückg.-Typ	Beschreibung
`addLast`	`int value`	`void`	fügt einen Wert am Ende der Liste ein
`addFirst`	`int value`	`void`	fügt einen Wert am Anfang der Liste ein
`removeFirst`		`int`	entfernt den ersten Wert und gibt ihn zurück
`removeLast`		`int`	entfernt den letzten Wert und gibt ihn 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



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.

Kahoot

https://create.kahoot.it/share/21-u9-eprog-mschoeb/46d2df0a-83de-4a10-a5b0-43c58fd202c9