

252-0027

Einführung in die Programmierung
Übungen

Woche 10: Verlinkte Objekte, Klassen

Timo Stucki

Departement Informatik
ETH Zürich

Organisatorisches

- Mein Name: *Timo Stucki*
- Bei Fragen: *tistucki@student.ethz.ch*
 - Mails bitte mit «[EProg25]» 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

Discord: timostucki

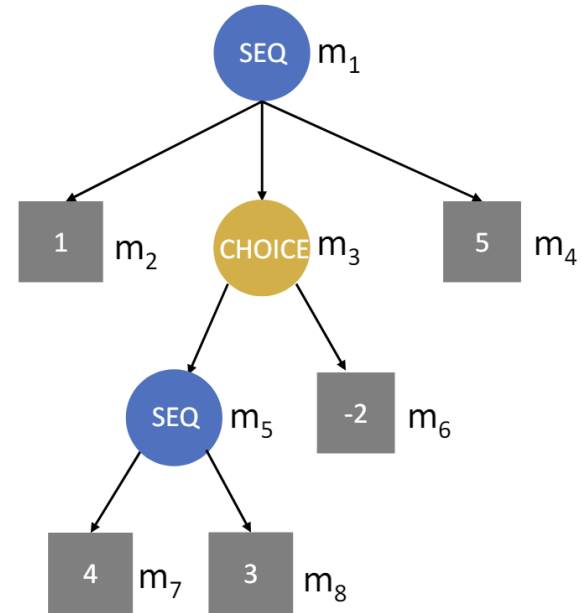
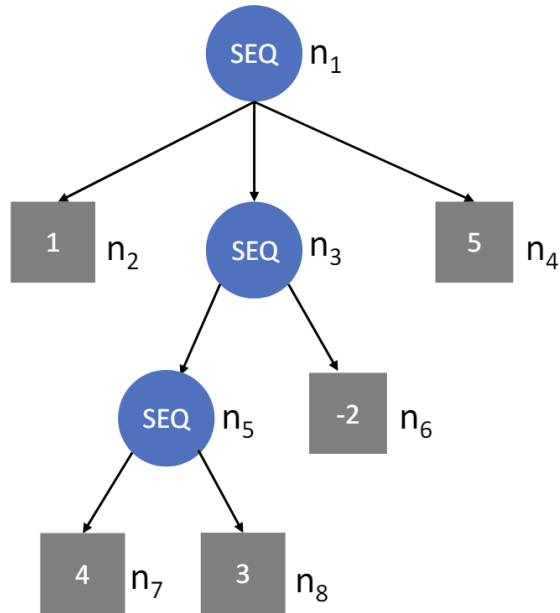
Webseite



■ timostucki.com

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 Programmozustand 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 Programmozustand 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 zurückgegebenen Liste spielt keine Rolle. Wenn das gleiche Resultat durch genau k verschiedene Ausführungen generiert werden kann, dann muss das Resultat k Mal in der zurückgegebenen 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.



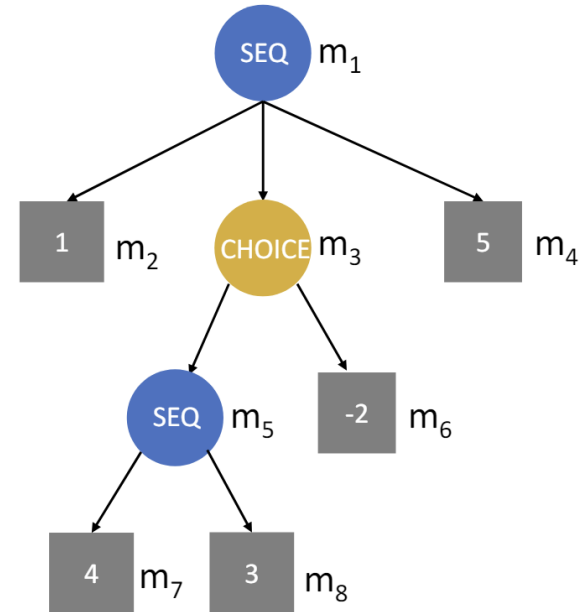
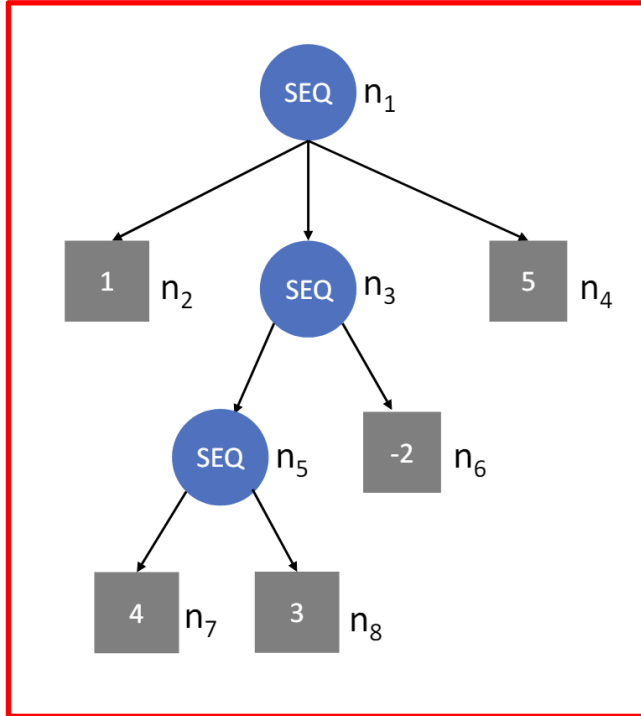
- **SEQ Node:** Kinderknoten werden nacheinander ausgeführt. Reihenfolge ist egal.
Das `value` Attribut wird ignoriert.



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

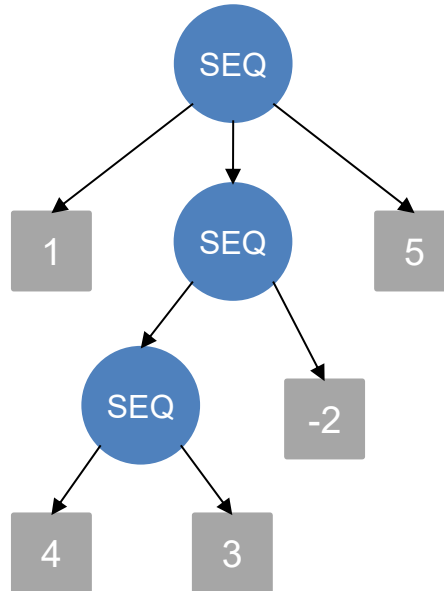


Probleme Lösen: Executable Graph



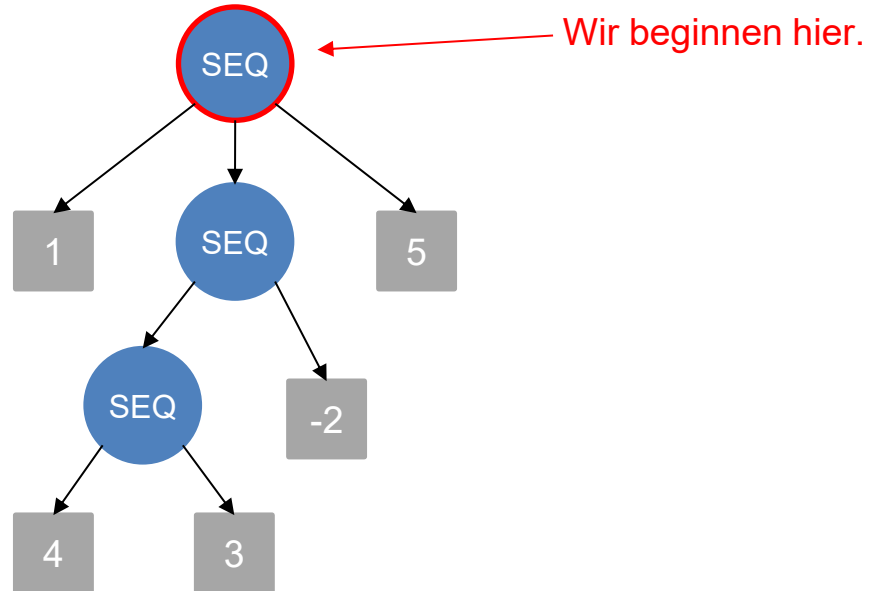
Probleme Lösen: Executable Graph

Startzustand: (1, 2)

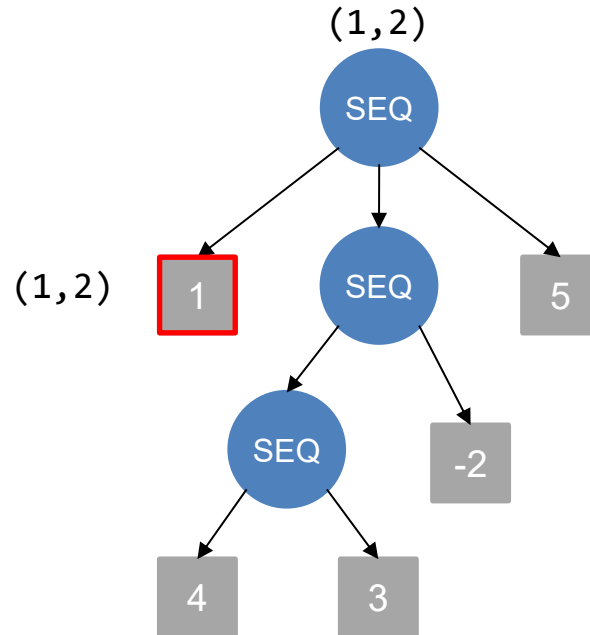


Probleme Lösen: Executable Graph

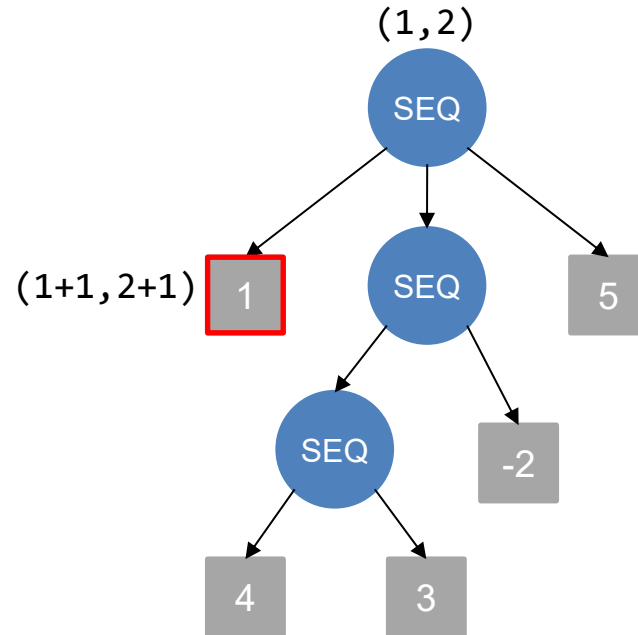
Startzustand: (1, 2)



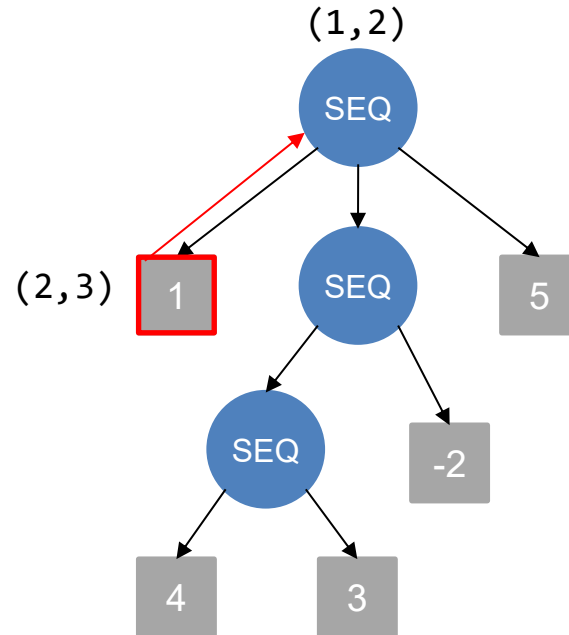
Probleme Lösen: Executable Graph



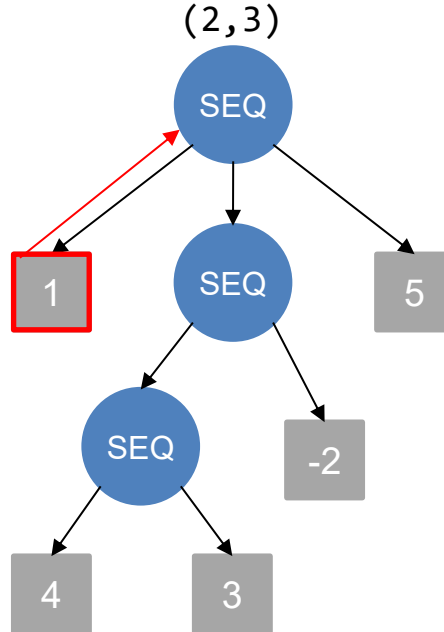
Probleme Lösen: Executable Graph



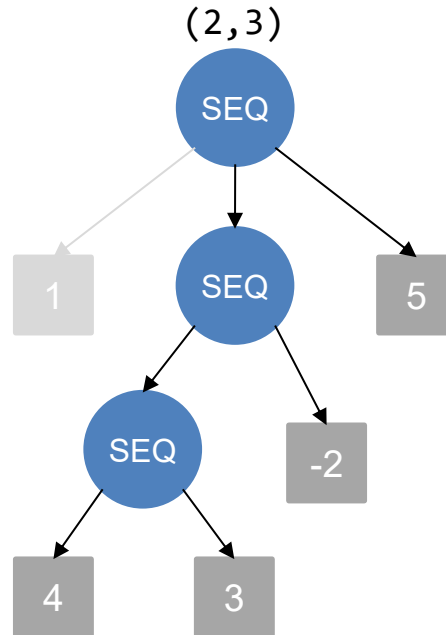
Probleme Lösen: Executable Graph



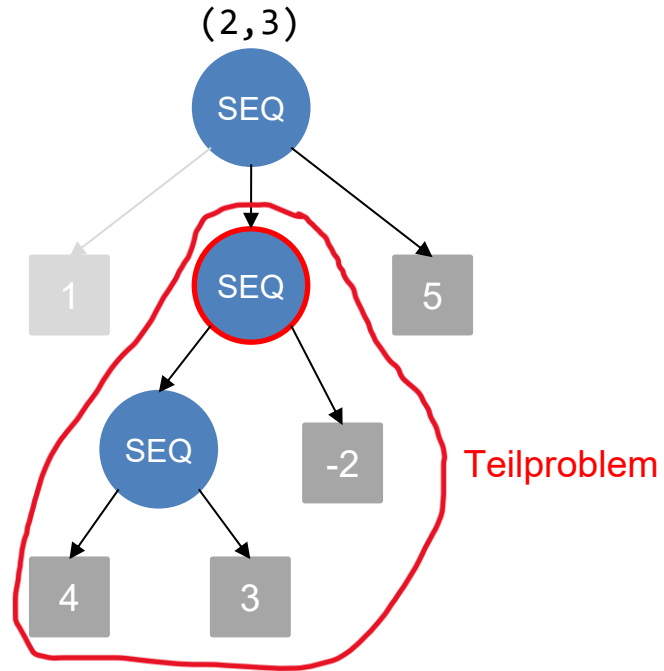
Probleme Lösen: Executable Graph



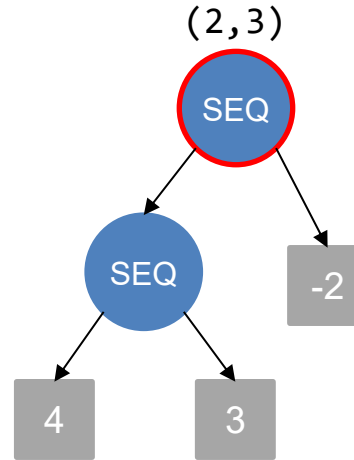
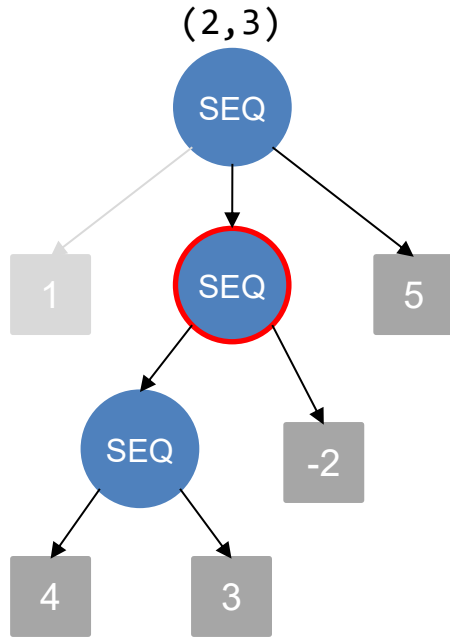
Probleme Lösen: Executable Graph



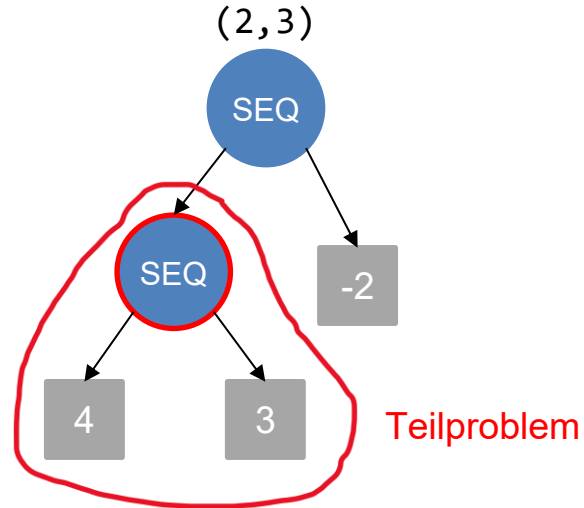
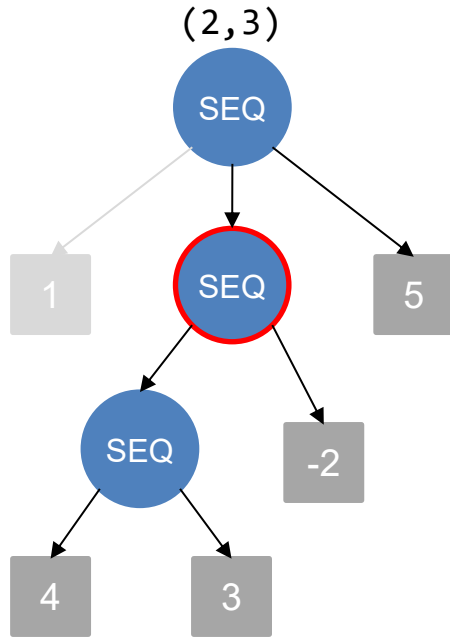
Probleme Lösen: Executable Graph



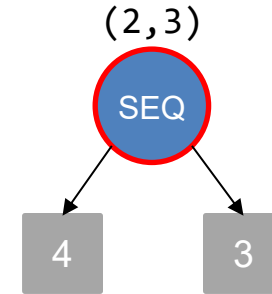
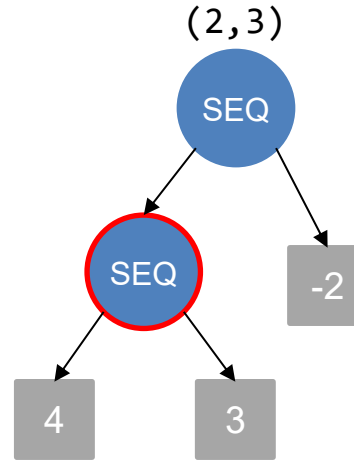
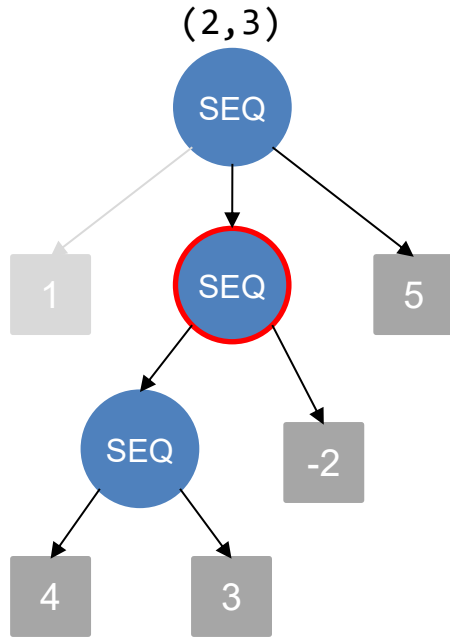
Probleme Lösen: Executable Graph



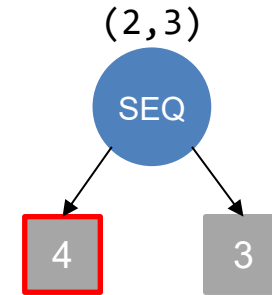
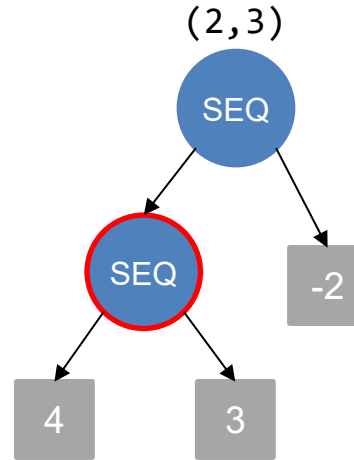
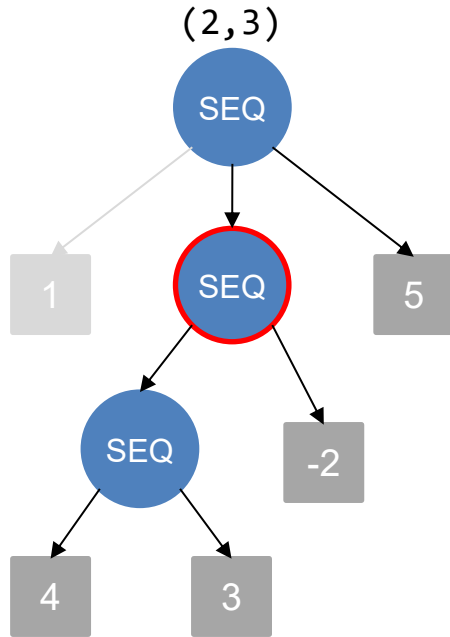
Probleme Lösen: Executable Graph



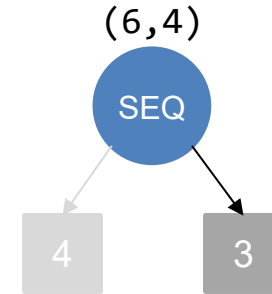
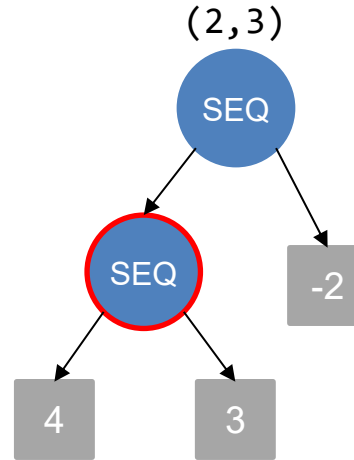
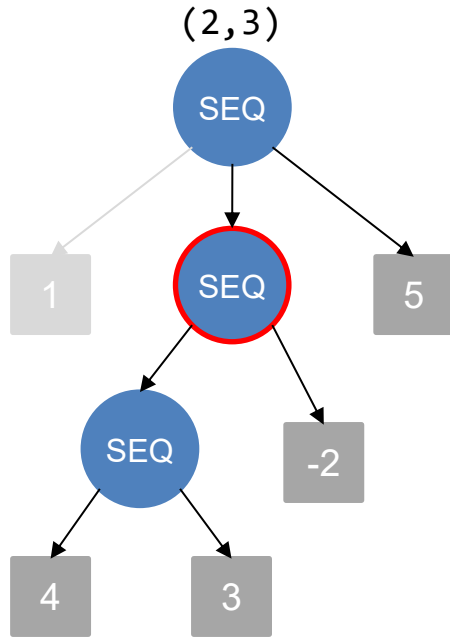
Probleme Lösen: Executable Graph



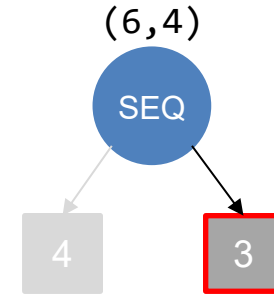
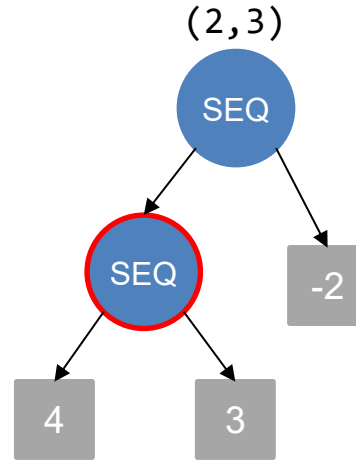
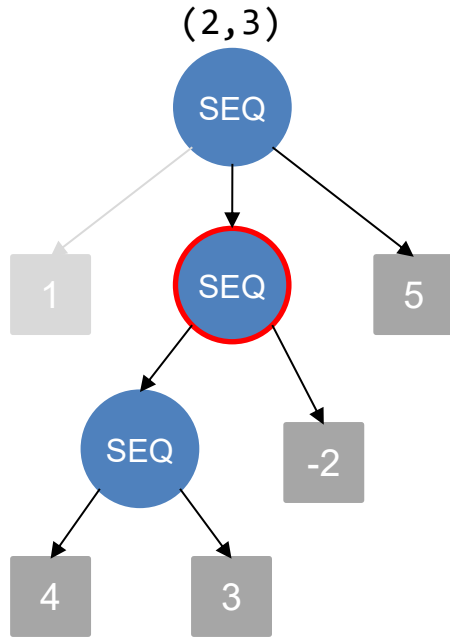
Probleme Lösen: Executable Graph



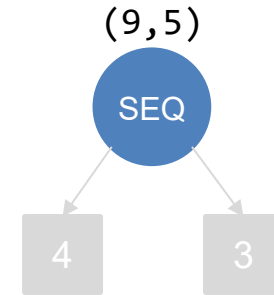
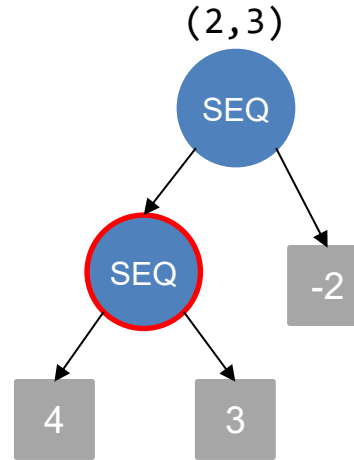
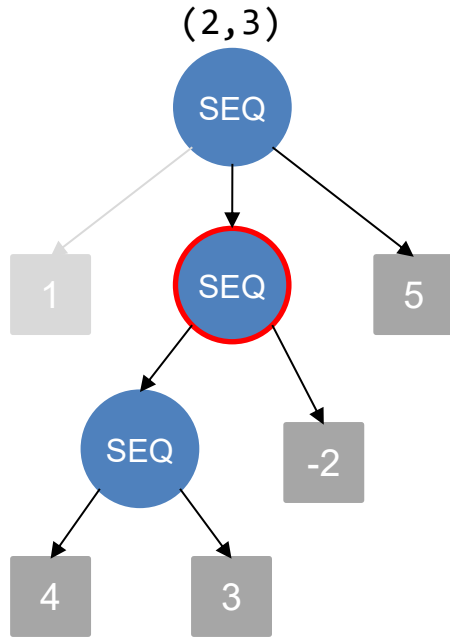
Probleme Lösen: Executable Graph



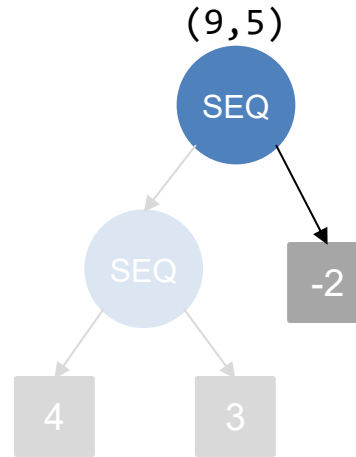
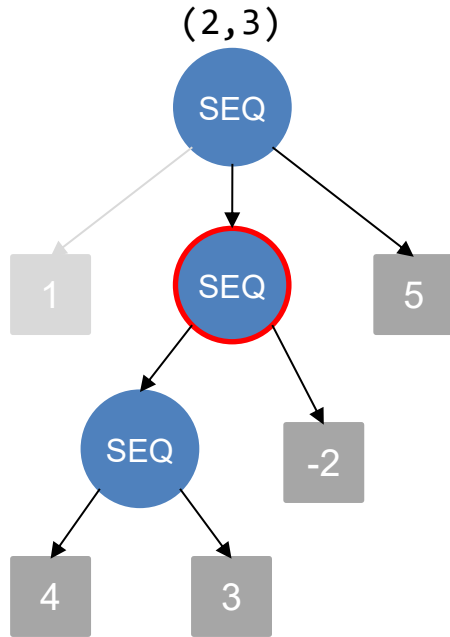
Probleme Lösen: Executable Graph



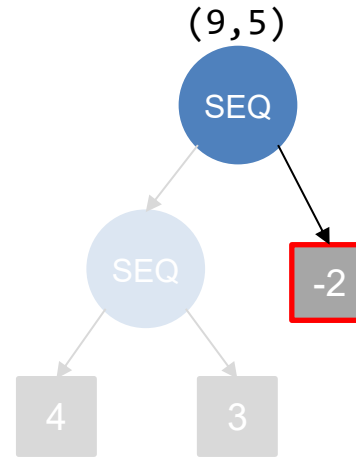
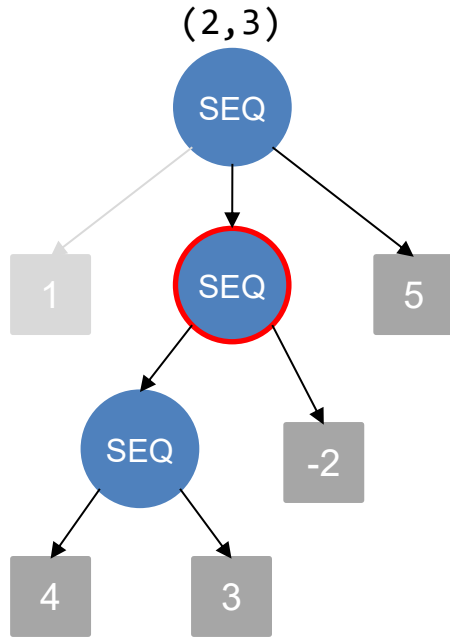
Probleme Lösen: Executable Graph



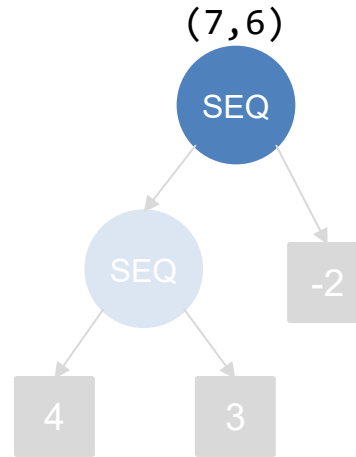
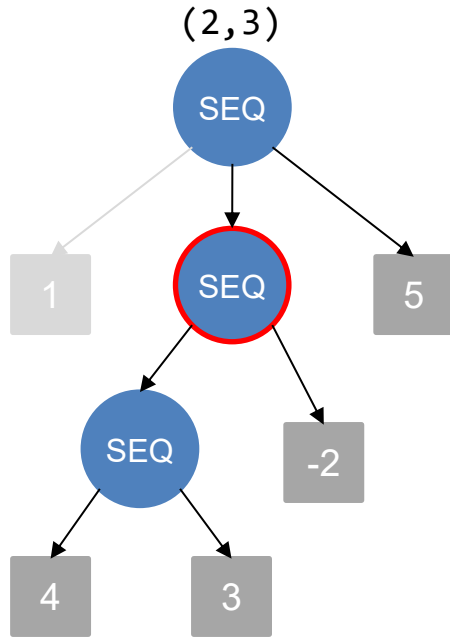
Probleme Lösen: Executable Graph



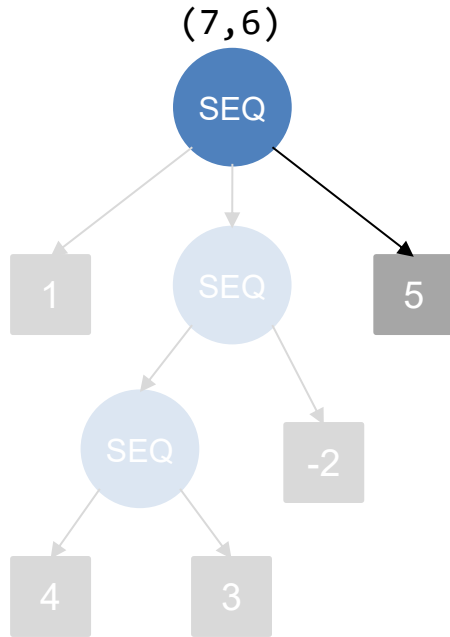
Probleme Lösen: Executable Graph



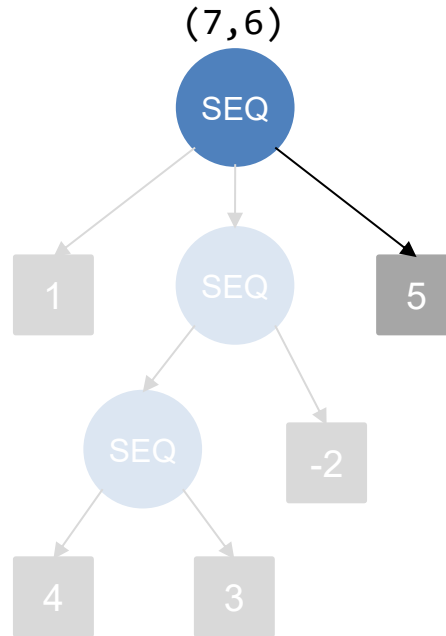
Probleme Lösen: Executable Graph



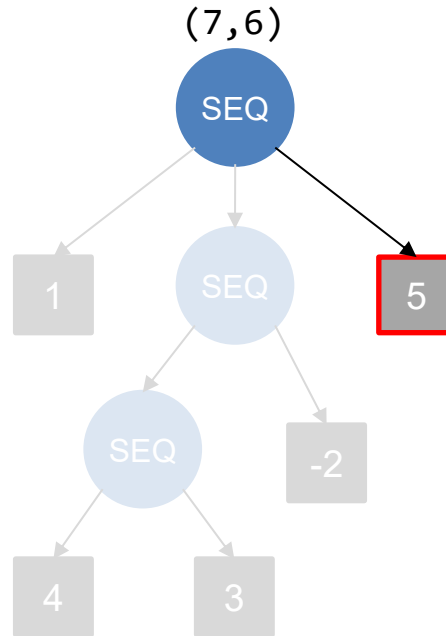
Probleme Lösen: Executable Graph



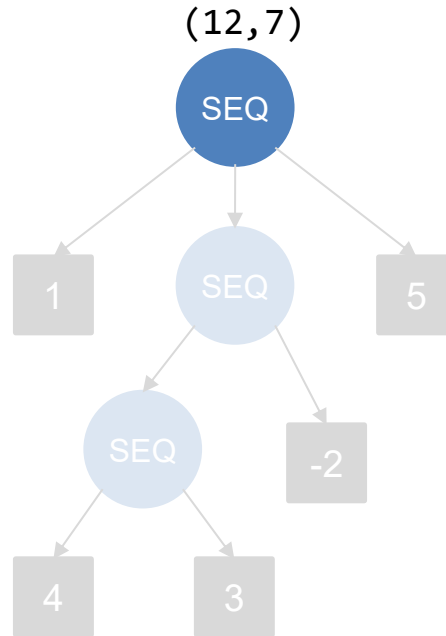
Probleme Lösen: Executable Graph



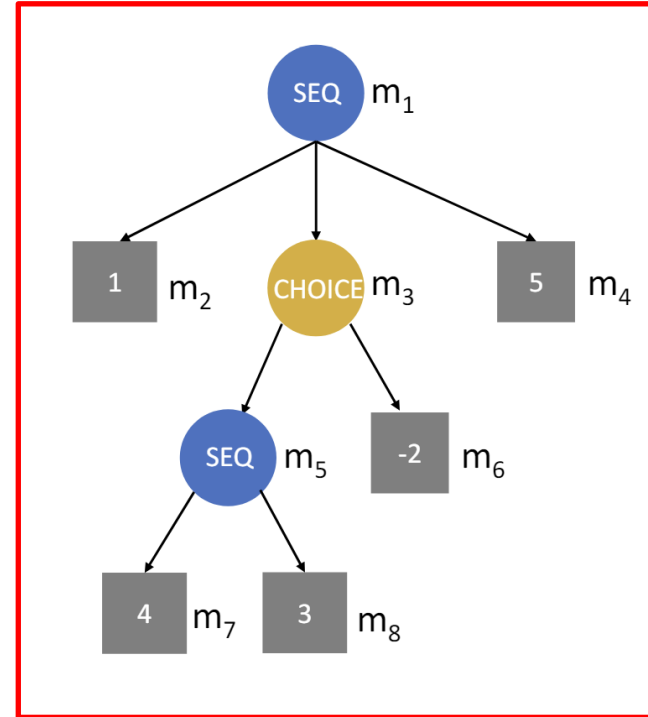
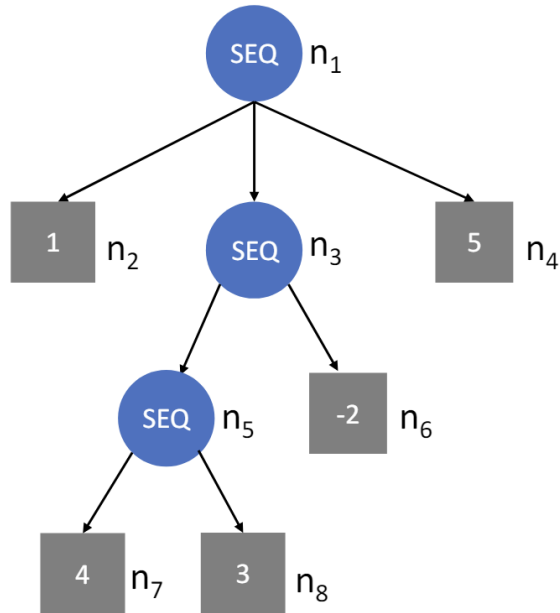
Probleme Lösen: Executable Graph



Probleme Lösen: Executable Graph

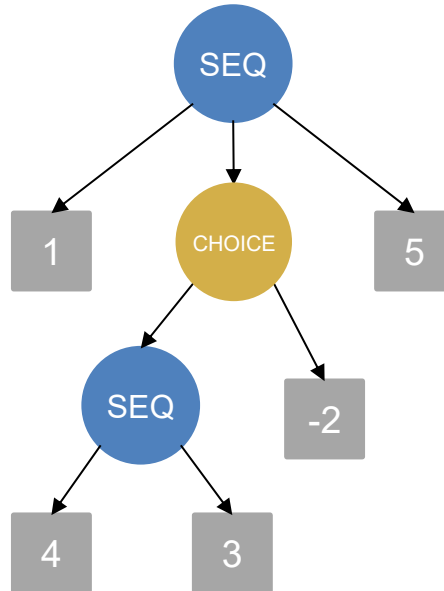


Probleme Lösen: Executable Graph



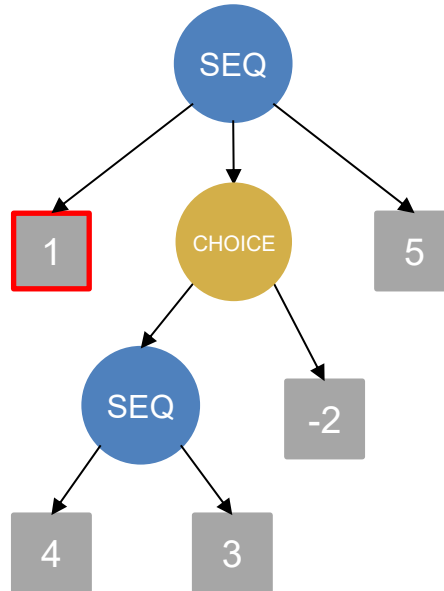
Probleme Lösen: Executable Graph

Startzustand: $(0,0)$



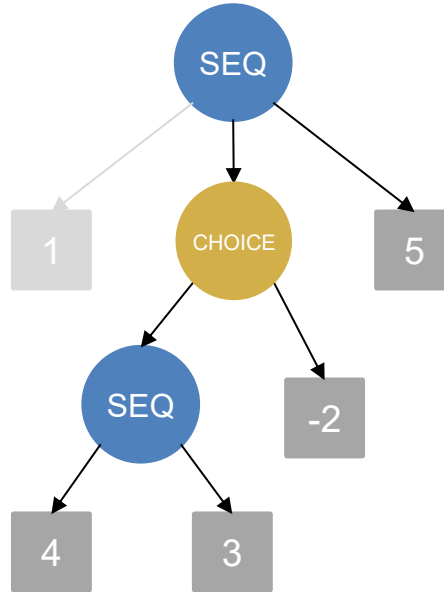
Probleme Lösen: Executable Graph

Startzustand: $(0, 0)$



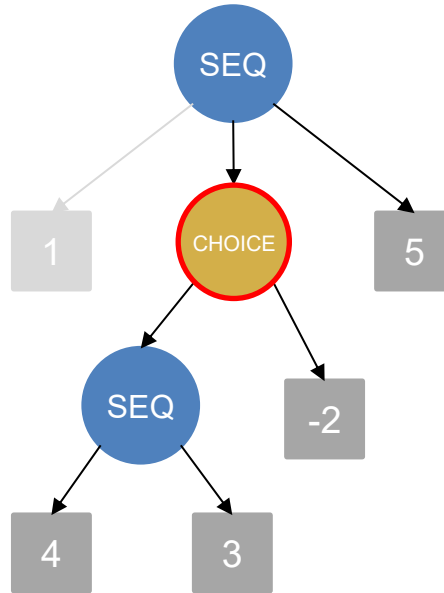
Probleme Lösen: Executable Graph

Startzustand: (1,1)



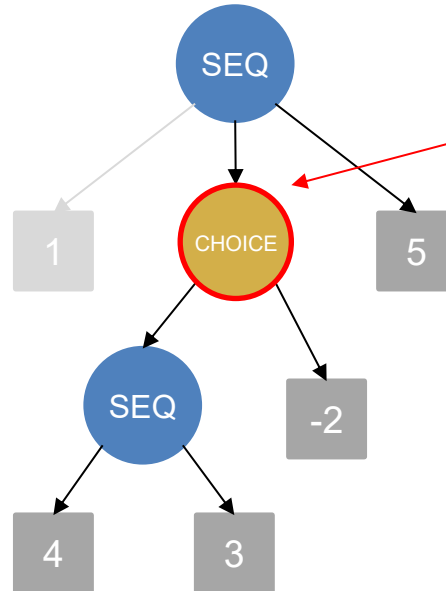
Probleme Lösen: Executable Graph

Startzustand: (1,1)



Probleme Lösen: Executable Graph

Startzustand: (1,1)

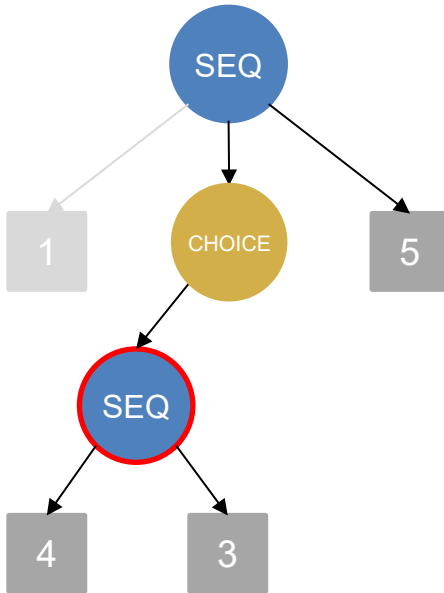


Wir haben jetzt die Wahl zwischen zwei Kinderknoten.

Probleme Lösen: Executable Graph

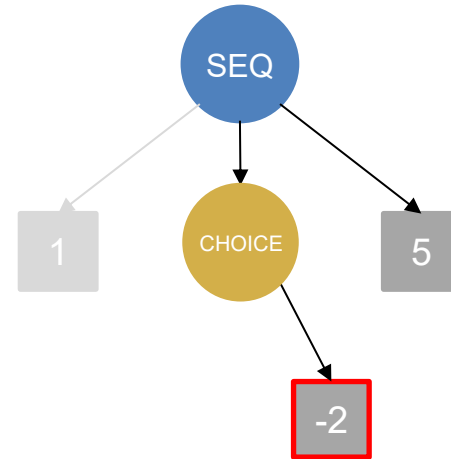
Resultat 1

(1,1)



Resultat 2

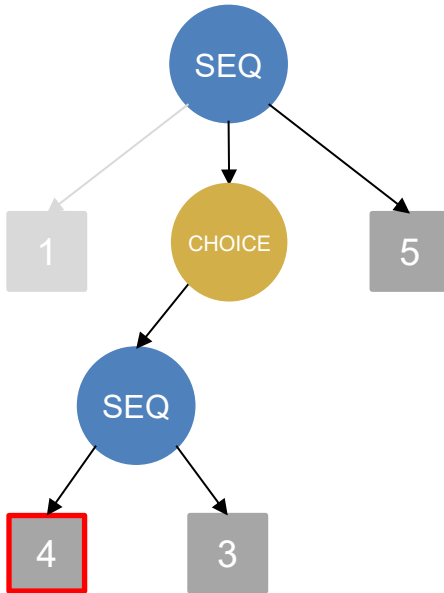
(1,1)



Probleme Lösen: Executable Graph

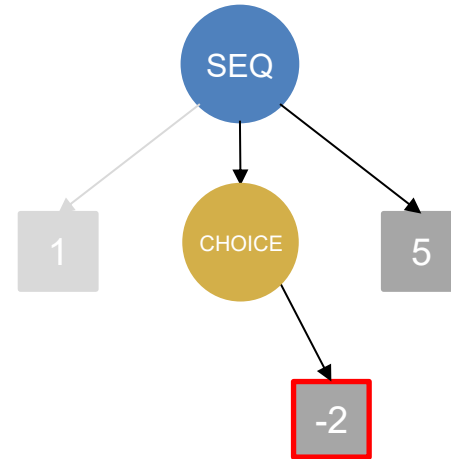
Resultat 1

(1,1)



Resultat 2

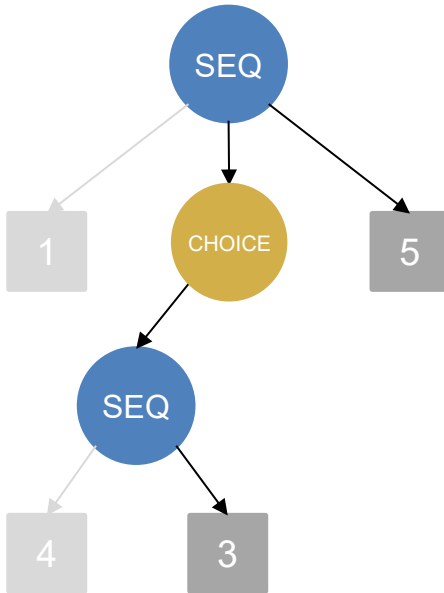
(1,1)



Probleme Lösen: Executable Graph

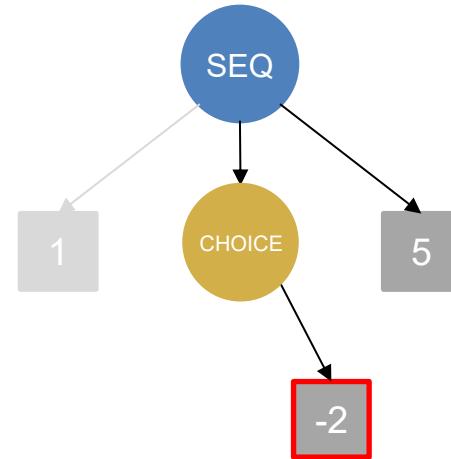
Resultat 1

(5, 2)



Resultat 2

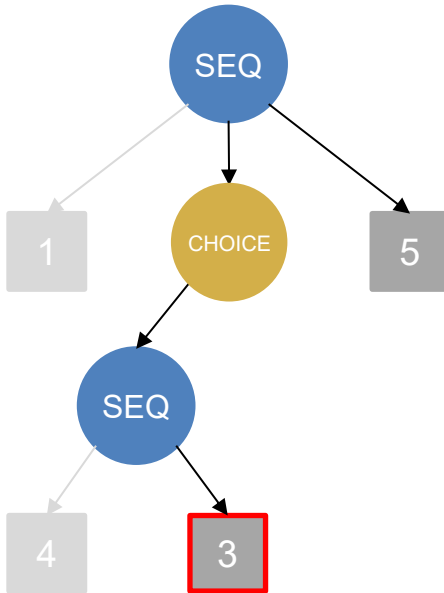
(1, 1)



Probleme Lösen: Executable Graph

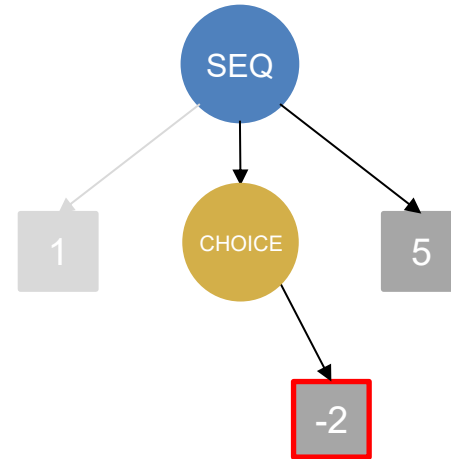
Resultat 1

(5, 2)



Resultat 2

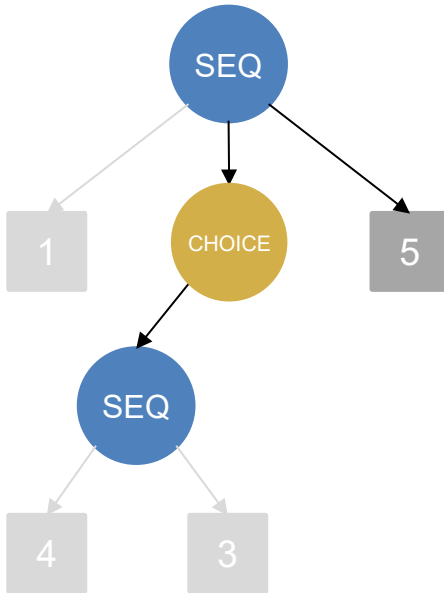
(1, 1)



Probleme Lösen: Executable Graph

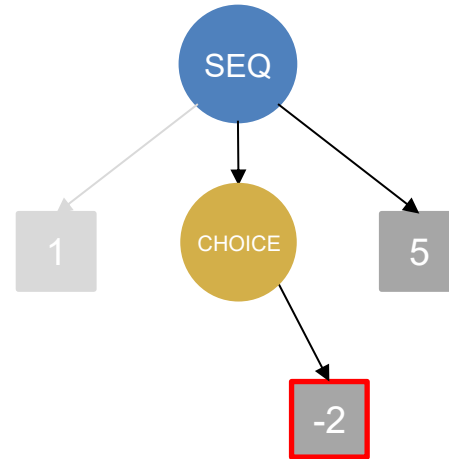
Resultat 1

(8, 3)



Resultat 2

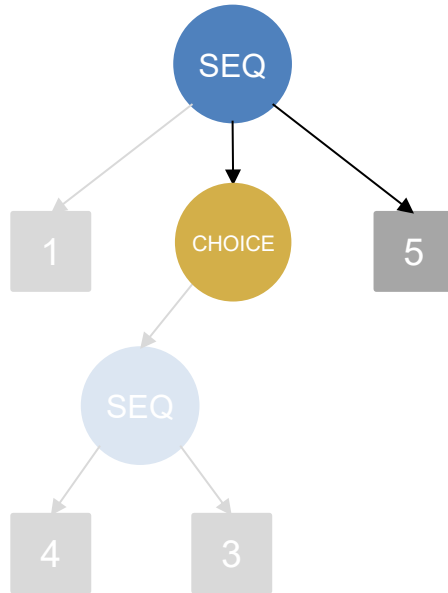
(1, 1)



Probleme Lösen: Executable Graph

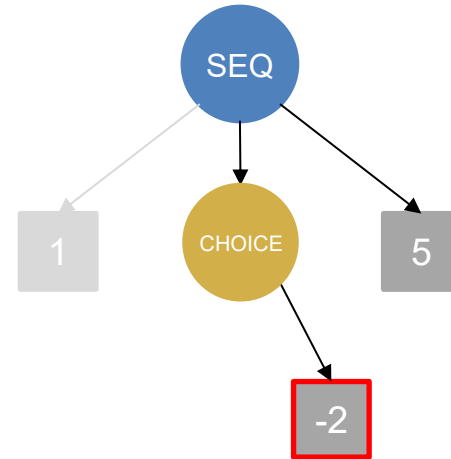
Resultat 1

(8, 3)



Resultat 2

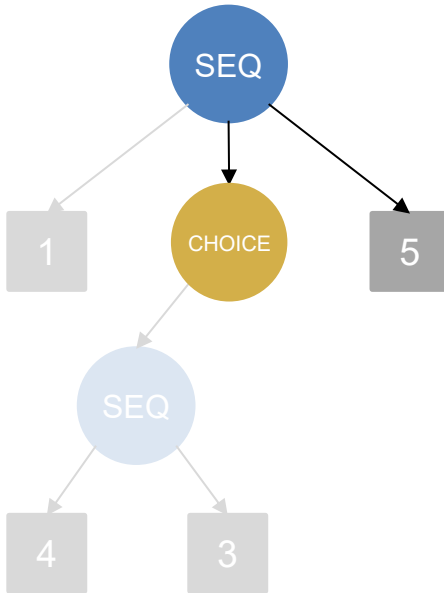
(1, 1)



Probleme Lösen: Executable Graph

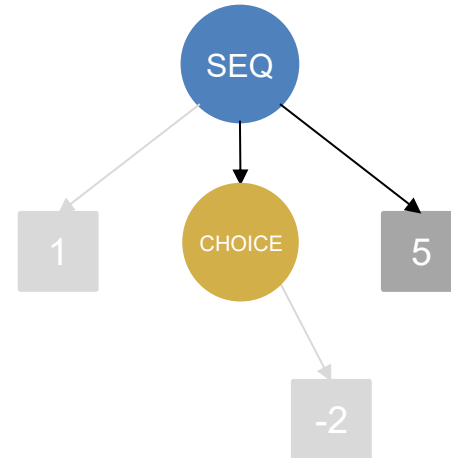
Resultat 1

$(8, 3)$



Resultat 2

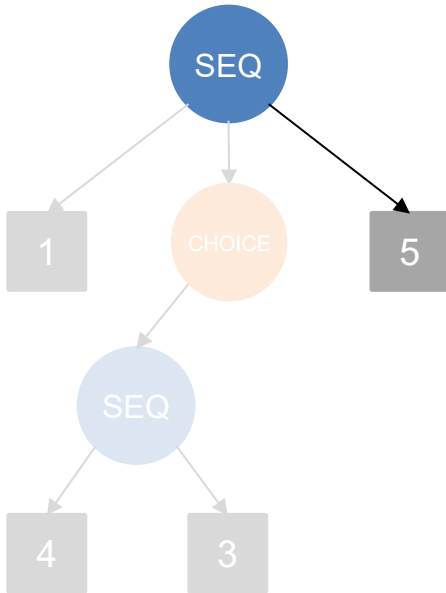
$(-1, 2)$



Probleme Lösen: Executable Graph

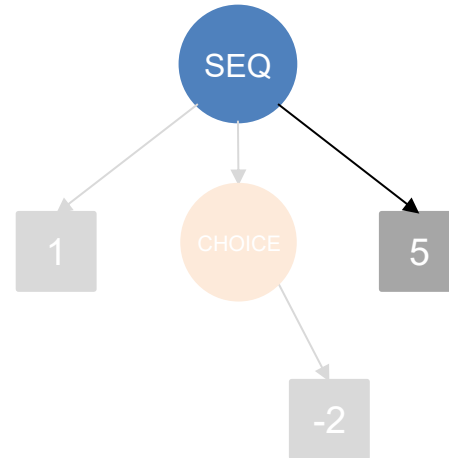
Resultat 1

$(8, 3)$

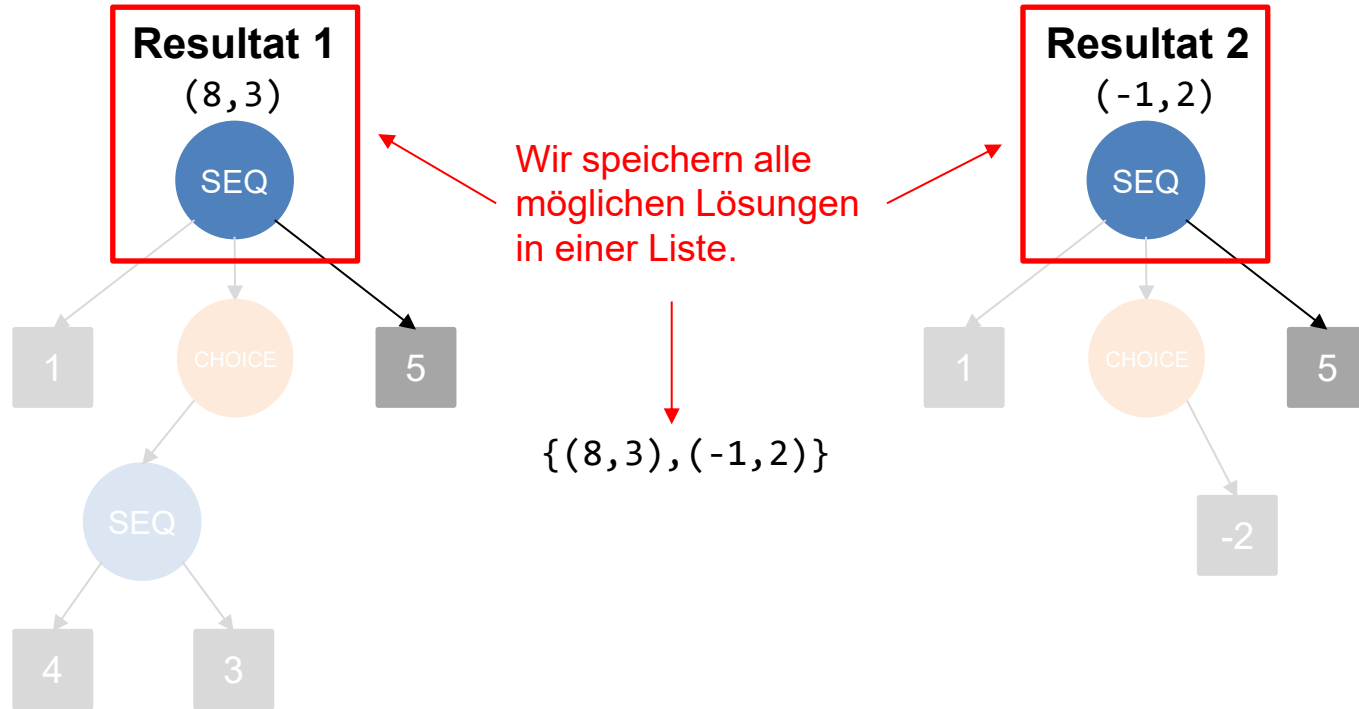


Resultat 2

$(-1, 2)$

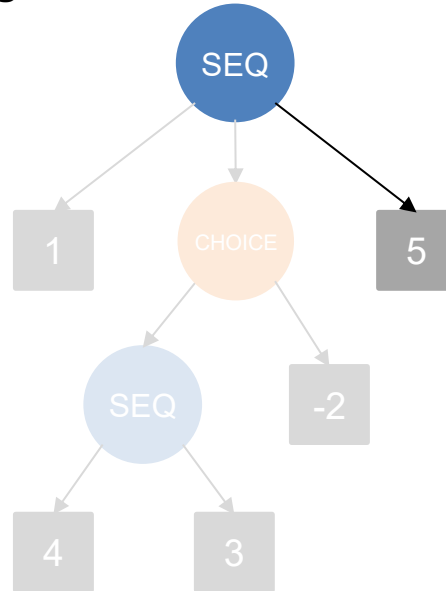


Probleme Lösen: Executable Graph



Probleme Lösen: Executable Graph

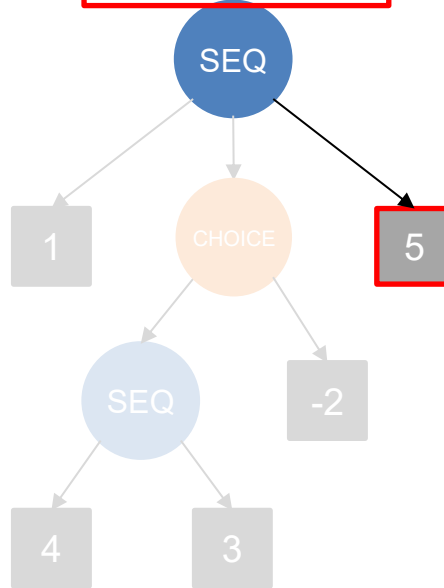
Lösungen: $\{(8, 3), (-1, 2)\}$



Probleme Lösen: Executable Graph

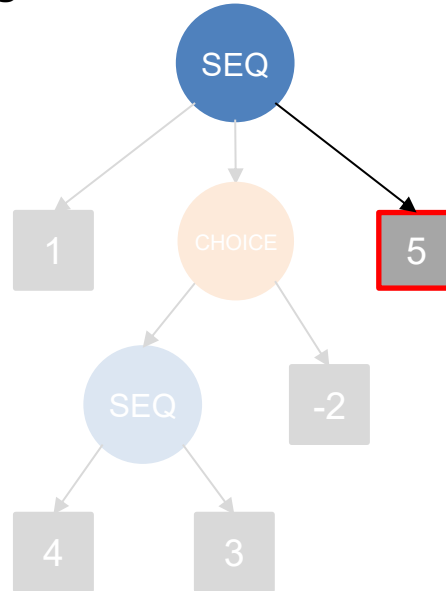
Lösungen: $\{(8, 3), (-1, 2)\}$

Wir müssen jedes
Zwischenresultat in
der Liste updaten.



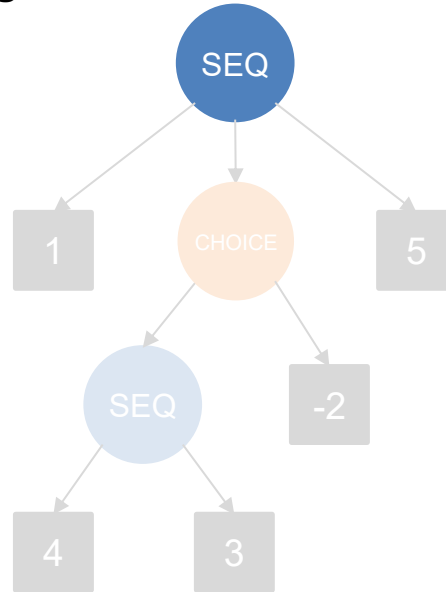
Probleme Lösen: Executable Graph

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;  
    }  
    (...)  
}
```

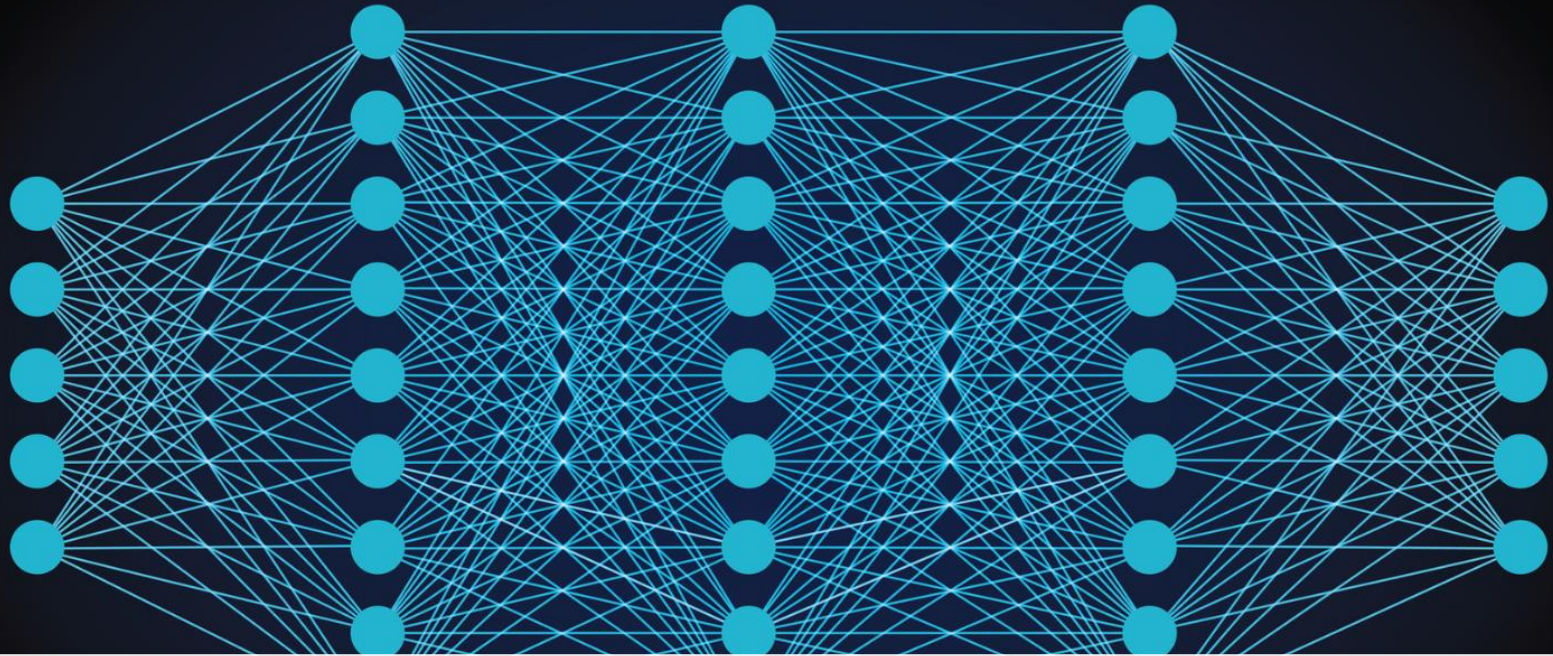
Wir rufen für jeden
Kinderknoten die
Methode rekursiv auf.

```

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.

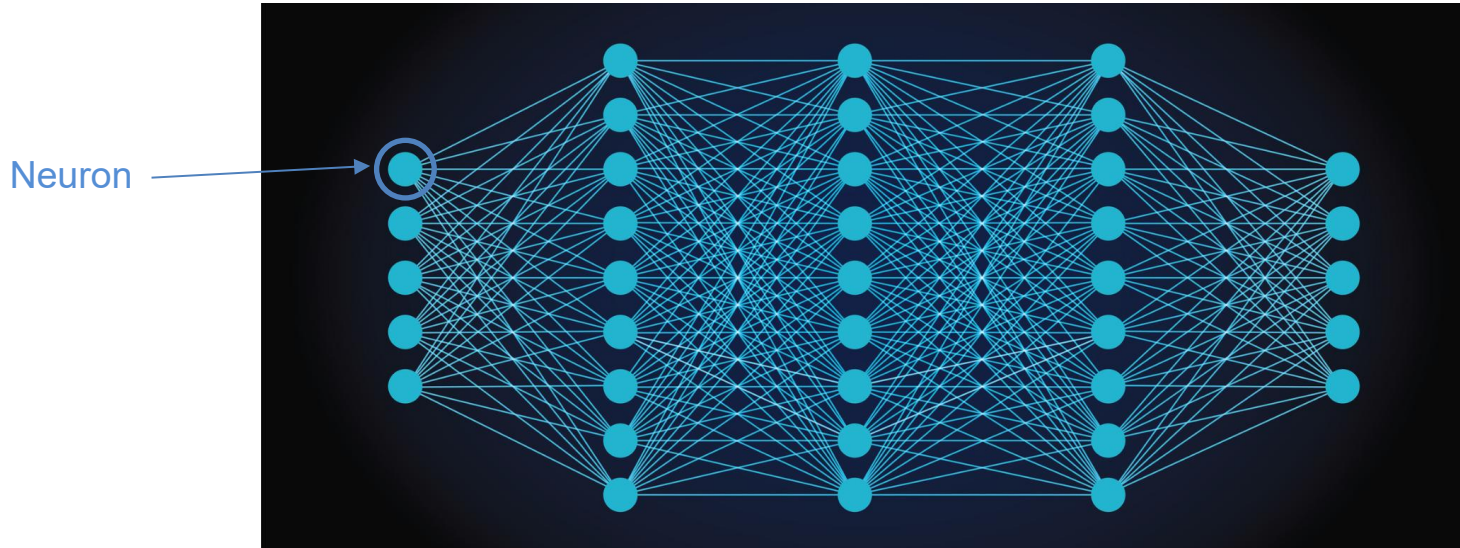


Klassen: Neural Network

Klassen: Neural Network

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

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

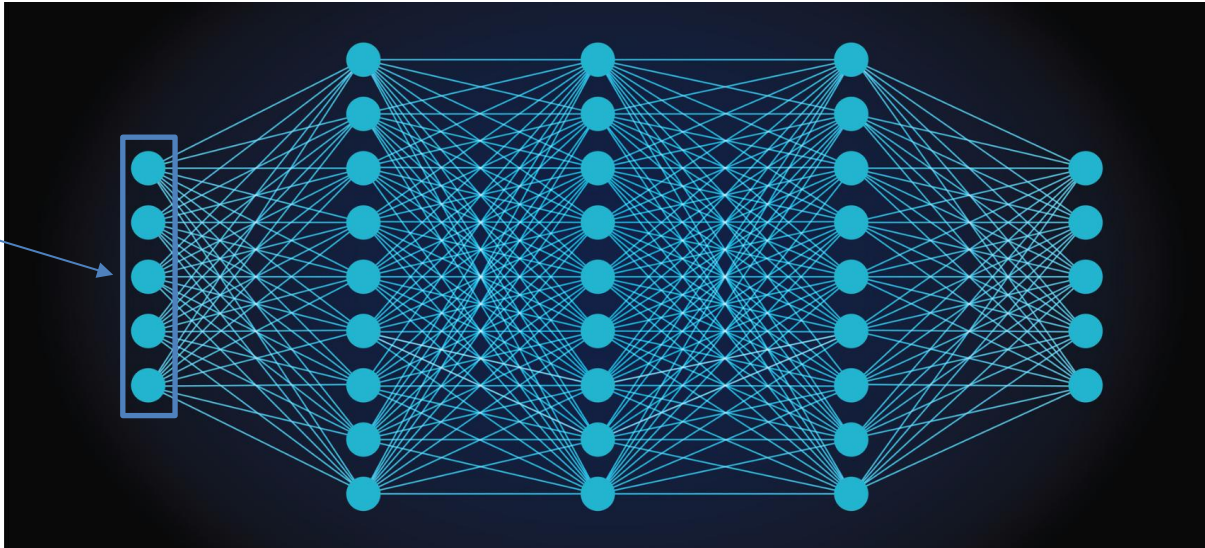


Klassen: Neural Network

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

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

Ein "Layer"
von einem
NN.

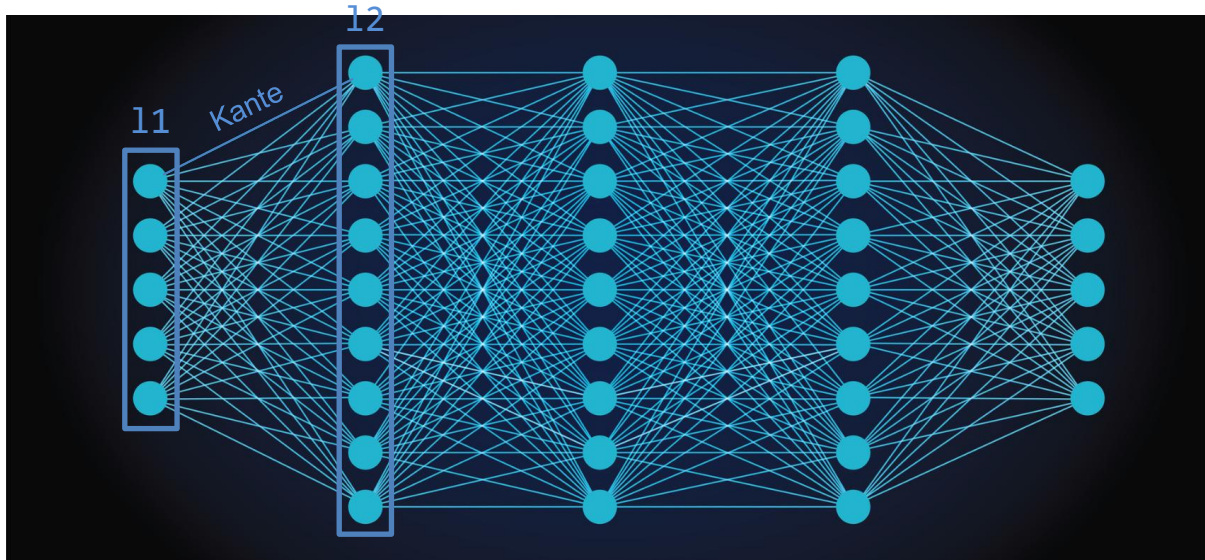


Klassen: Neural Network

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

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

Zwei Layers 11 und 12 welche durch Kanten verbunden sind, welche jeweils ein Gewicht besitzen.

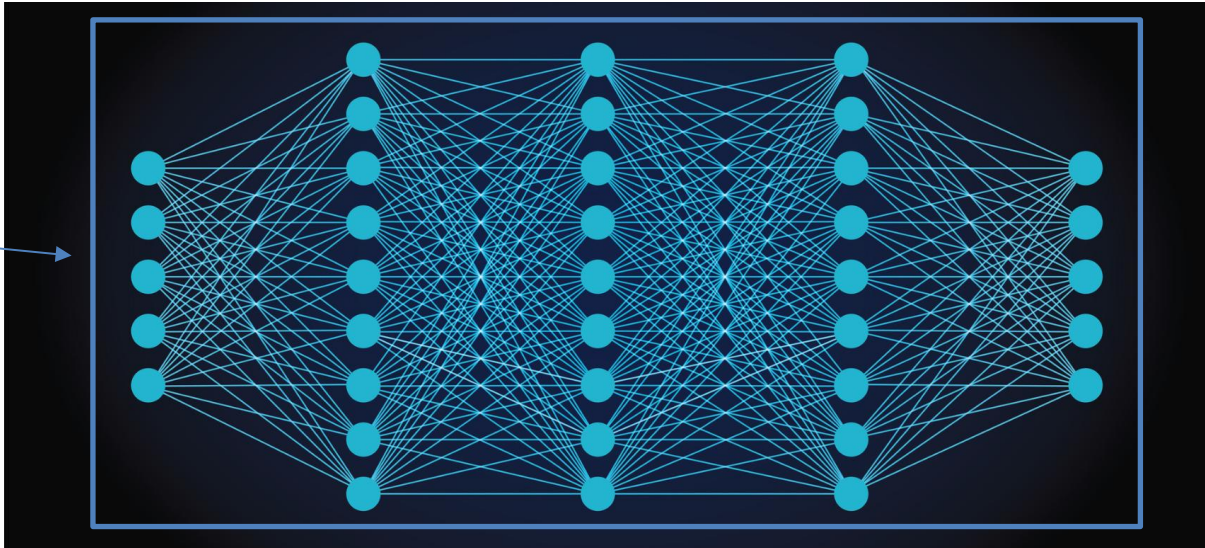


Klassen: Neural Network

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

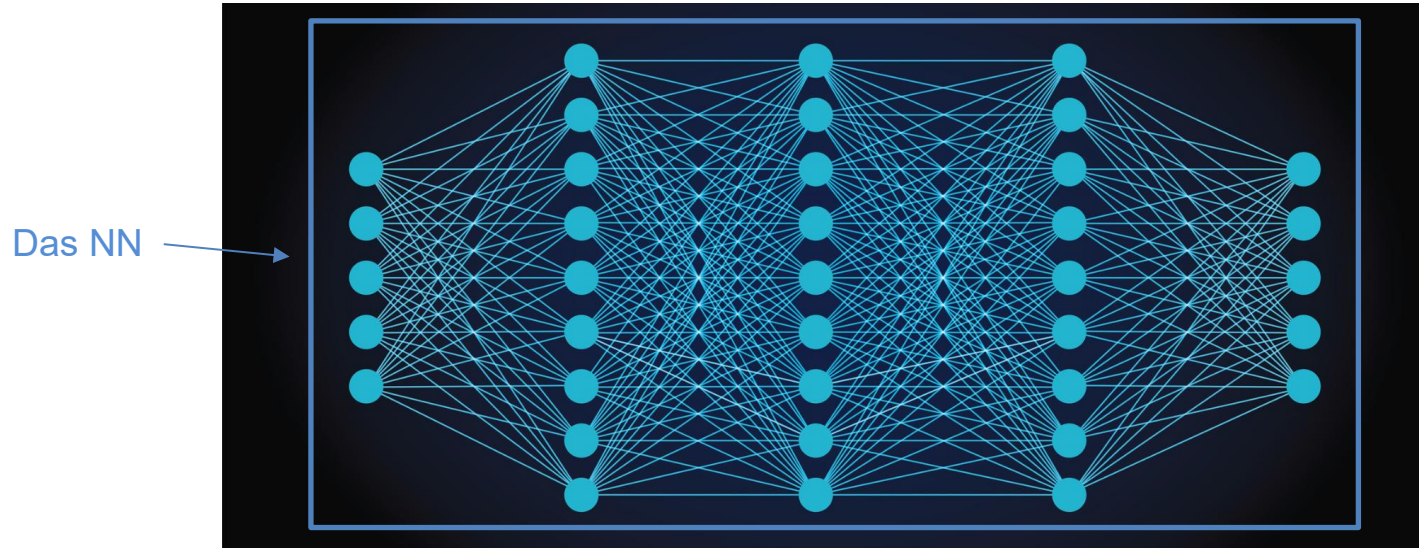
- Wir schreiben also eine Klasse `NeuralNetwork`, welche ein neuronales Netzwerk modelliert.

Das NN →



Klassen: Neural Network

Das Neural Network 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.





```
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

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



Attribut – private access modifier

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

```
    public Neuron() {  
        this(0);  
    }
```

```
    public int getValue() {  
        return value;  
    }
```

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

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



```
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

```
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 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;
    }
}
```



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

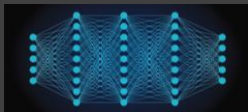
public void setWeights(double[] weights) {
    this.weights = weights;
}
```

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

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

← Klassenmethode – auch Member-Methode

```
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
    }
```

```
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

```
public class Neuron {
    private int value;

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

    public Neuron() {
        this(0);
    }

    public int getValue() {
        return value;
    }

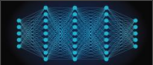
    public void setValue(int value) {
        this.value = value;
    }
}
```



Neuron.java


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

```
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

```
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

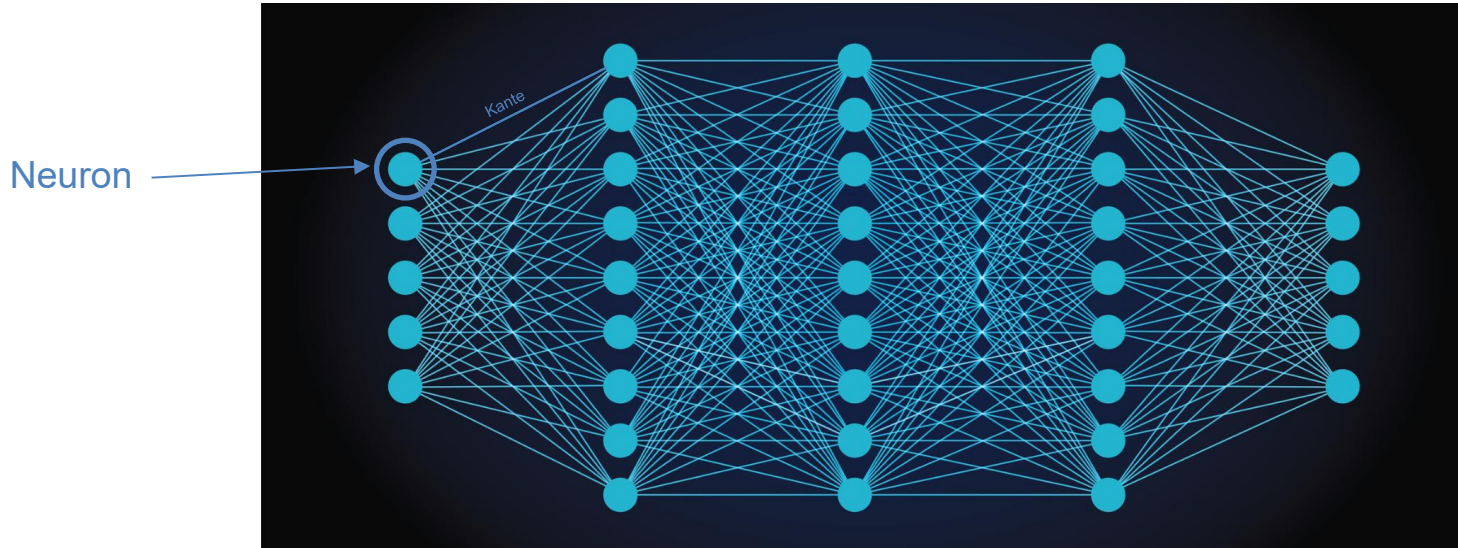
```
public class Neuron {  
    private int value;  
  
    public Neuron(int value) {  
        this.value = value;  
    }  
  
    public Neuron() {  
        this(0);  
    }  
  
    public int getValue() {  
        return value;  
    }  
  
    public void setValue(int value) {  
        this.value = value;  
    }  
}
```



Neuron.java

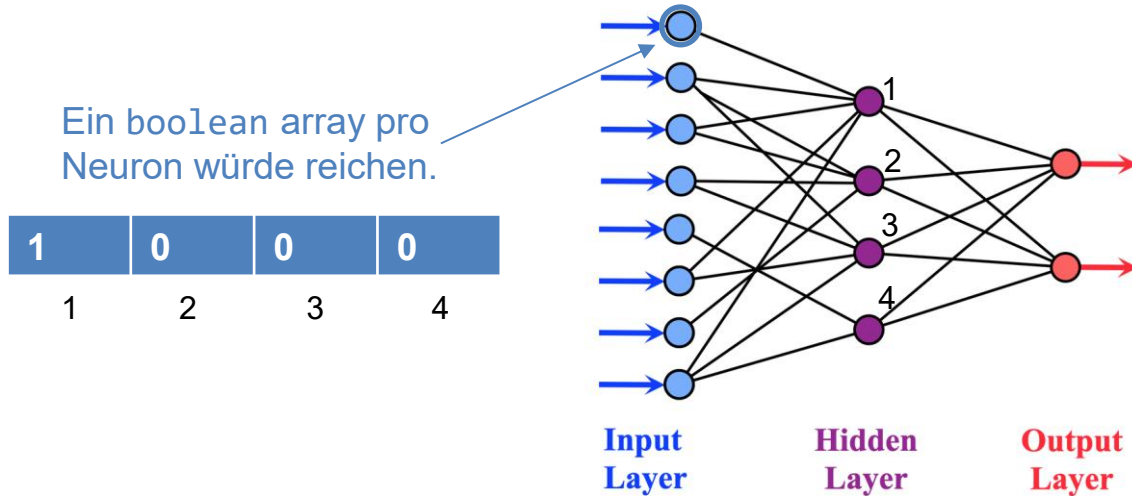
Klassen: Neural Network

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



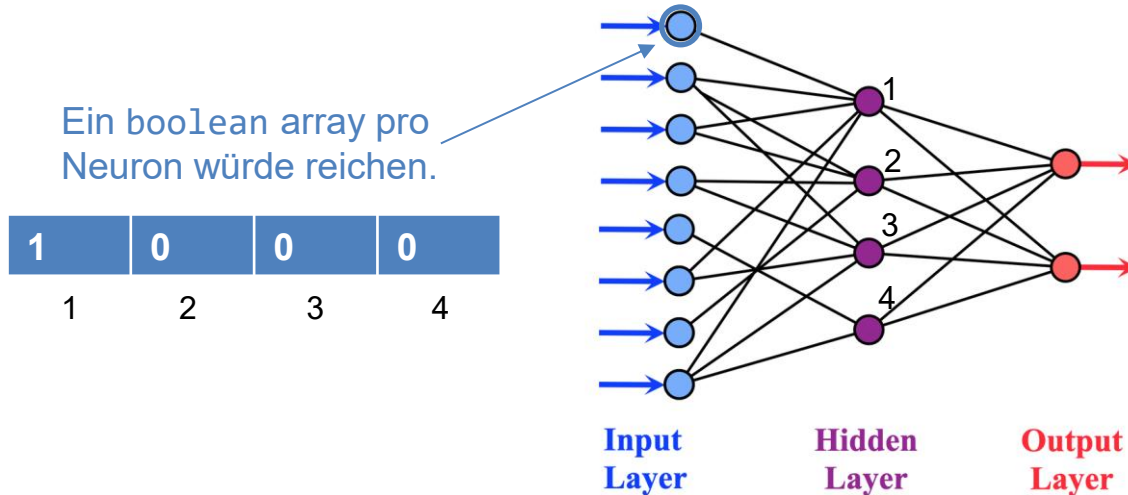
Klassen: Neural Network

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



Klassen: Neural Network

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



✨ Vererbung! ✨

```

public class SparseNetwork extends NeuralNetwork {

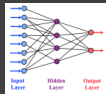
    private boolean[][] connections;

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

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

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

```



SparseNetwork.java

```

public class NeuralNetwork {
    private Layer[] layers;

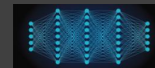
    public NeuralNetwork(Layer[] layers) {
        this.layers = layers;
    }

    public NeuralNetwork() {
        this(null);
    }

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

    public void train() {
        // TODO
    }
}

```



NeuralNetwork.java

```

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

```

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

```

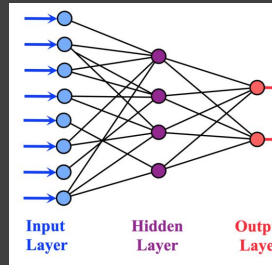
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][] connections;

    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.

```

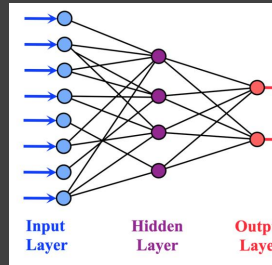
private static class SparseNetwork extends NeuralNetwork {
    private boolean[][] connections;

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

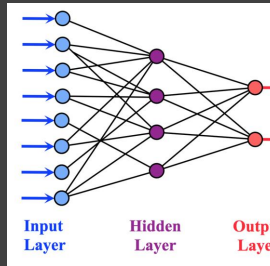
    public SparseNetwork() {
        SparseNetwork(null, null);
    }

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

```



Die restlichen Attribute
werden von NeuralNetwork
geerbt.



```
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!

```

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

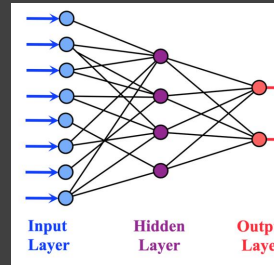
    public SparseNetwork(Layer[] layers, boolean[][] connections) {
        super(layers);
        this.connections = connections;
    }

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

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

```

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



```

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

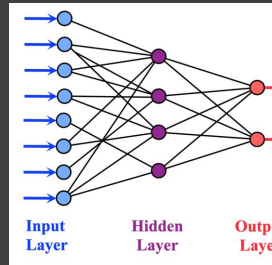
    public SparseNetwork(Layer[] layers, boolean[][] connections) {
        super(layers);
        this.connections = connections;
    }

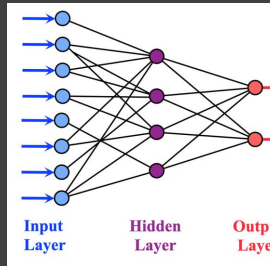
    public SparseNetwork() {
        SparseNetwork(null, null);
    }

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

```

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



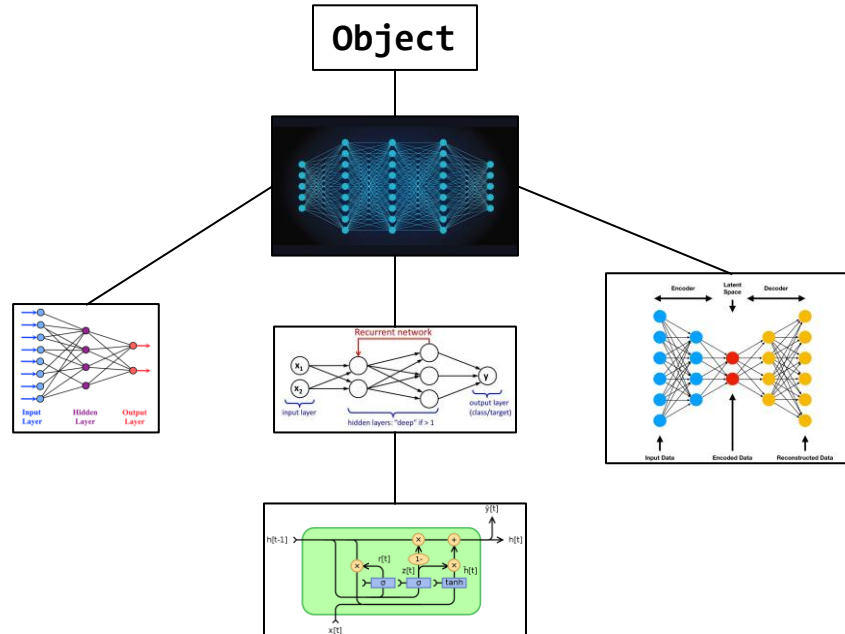


```
private static class SparseNetwork extends NeuralNetwork {  
    private boolean[][] connections;  
  
    public SparseNetwork(Layer[] layers, boolean[][] connections) {  
        super(layers);  
        this.connections = connections;  
    }  
  
    public SparseNetwork() {  
        SparseNetwork(null, null);  
    }  
  
    @Override  
    public void train() {  
        // TODO  
    }  
}
```

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

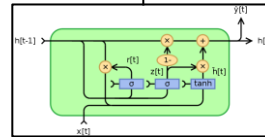
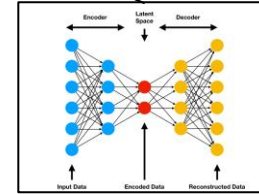
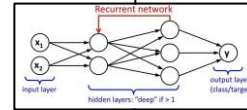
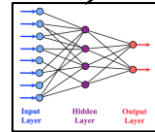
Klassen: Neural Network

Wir könnten diverse neuronale Netzwerke so durch eine Klasse beschreiben...

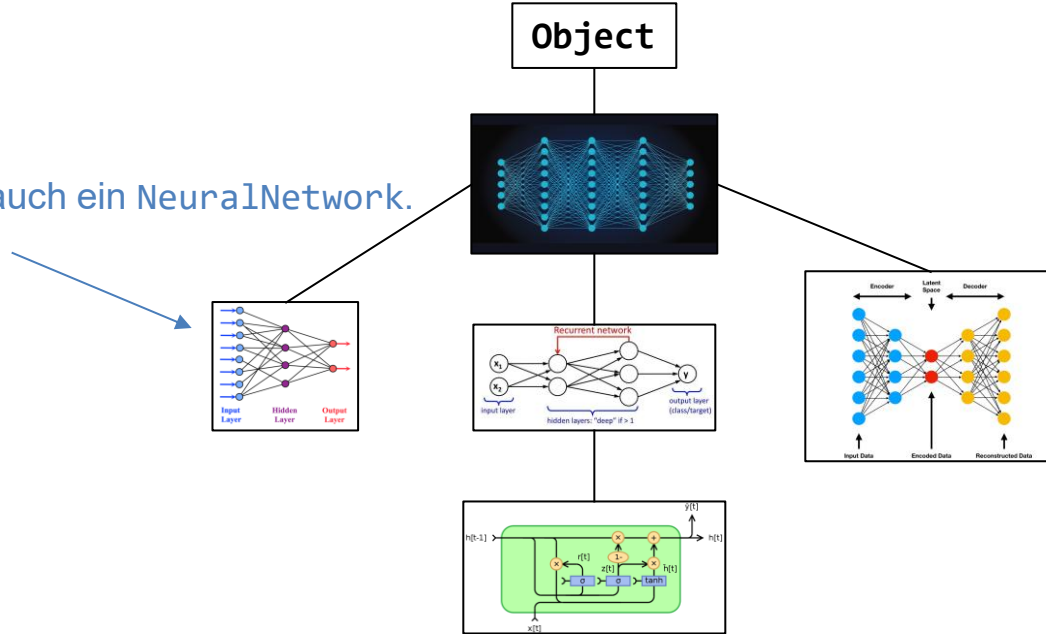


Object

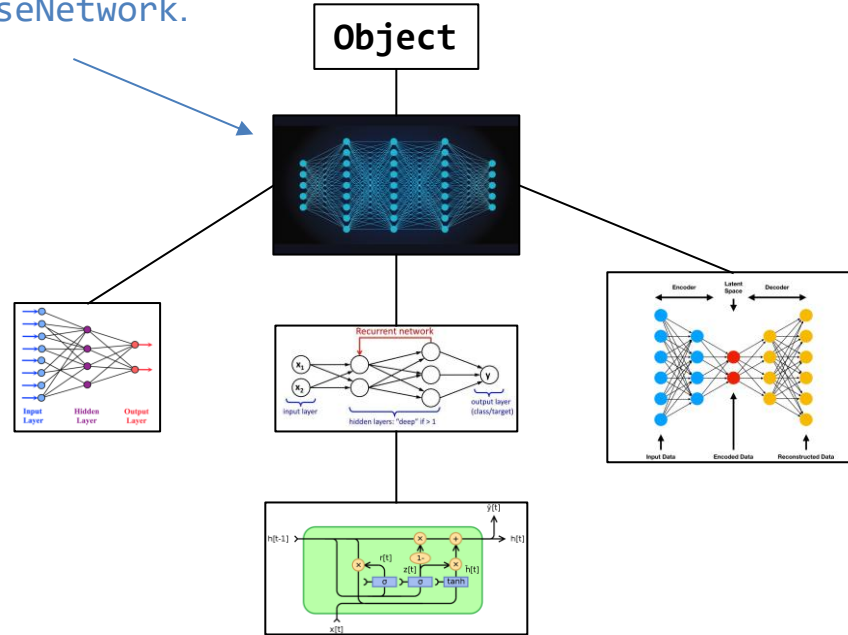
Alle Klassen sind Subklassen von der Klasse Object.



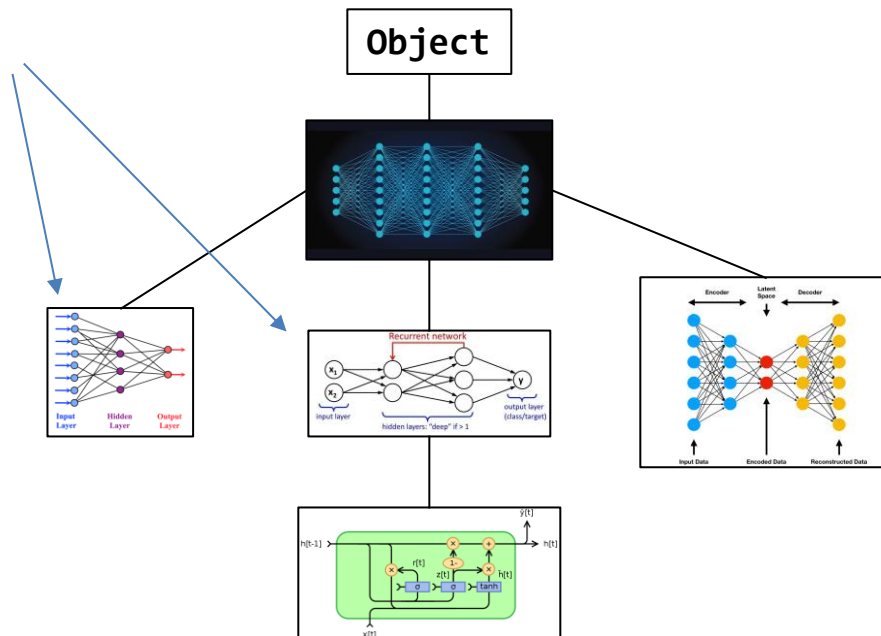
SparseNetwork ist auch ein NeuralNetwork.



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

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 <code>arr</code>	ArrayList <code>arrList</code>
Lese Element an Index <code>i</code>	<code>arr[i]</code>	<code>arrList.get(i)</code>
Element an Index <code>i</code> auf <code>e</code> setzen	<code>arr[i] = e;</code>	<code>arrList.set(i, e);</code>
Erstes Element	<code>arr[0]</code>	<code>arrList.getFirst()</code>
Letztes Element	<code>arr[arr.length - 1]</code>	<code>arrList.getLast()</code>
Länge	<code>arr.length</code>	<code>arrList.size()</code>

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	<code>arrList.add(i, e)</code>
Füge Element e am Anfang der Liste hinzu	<code>arrList.addFirst(e);</code>
Füge Element e am Ende der Liste hinzu	<code>arrList.addLast(e)</code>
Prüfe ob Element e enthalten ist	<code>arrList.contains(e)</code>
In Array umwandeln	<code>arrList.toArray()</code>

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

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	<code>LinkedList list</code>
Liste als Stack (Element entfernen)	<code>list.pop()</code>
Liste als Stack (Element e hinzufügen)	<code>list.push(e)</code>
Liste als Queue (Element entfernen)	<code>list.poll()</code>
Liste als Queue (Element e hinzufügen)	<code>list.add(e)</code>

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	<code>LinkedList list</code>
Liste als Stack (Element entfernen)	<code>list.removeFirst()</code>
Liste als Stack (Element e hinzufügen)	<code>list.addFirst(e)</code>
Liste als Queue (Element entfernen)	<code>list.removeLast()</code>
Liste als Queue (Element e hinzufügen)	<code>list.addFirst(e)</code>

Listen

LinkedLists

LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

LinkedLists

Referenz auf Objekt des selben Typs!

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

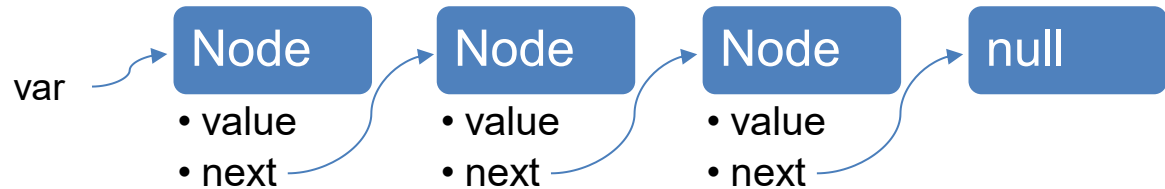
single linked list Knotenaufbau

LinkedLists

Referenz auf Objekt des selben Typs!

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau



LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel

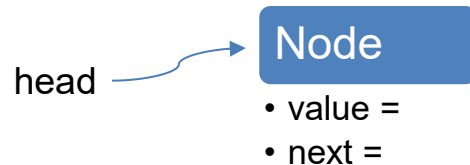
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



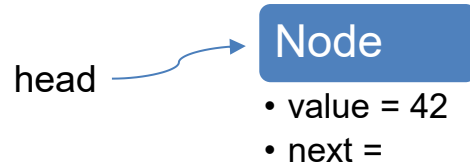
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



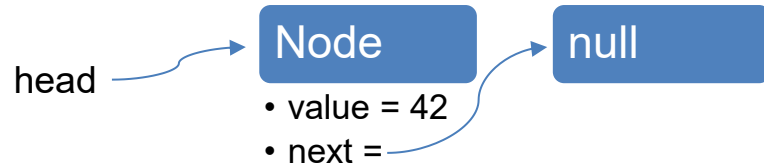
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



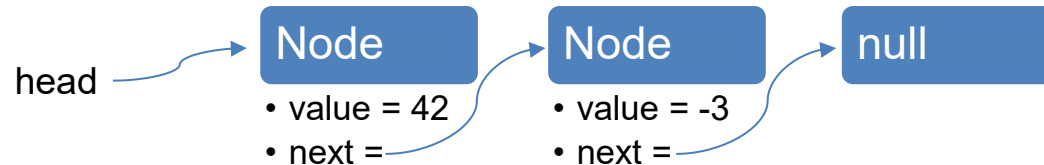
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



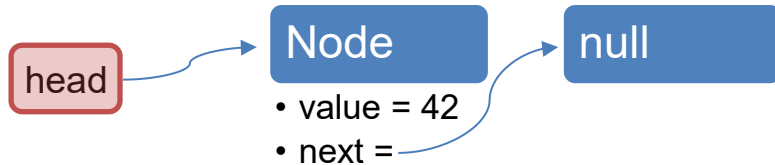
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



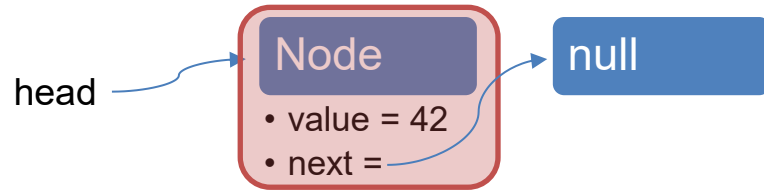
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



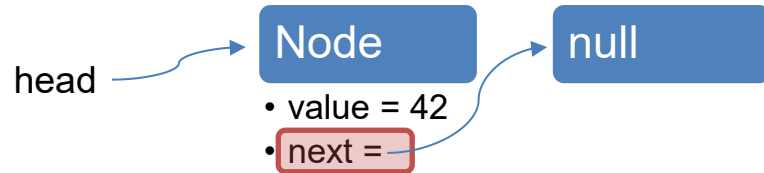
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



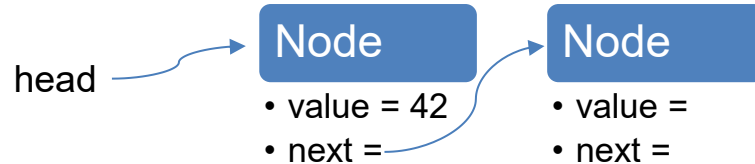
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



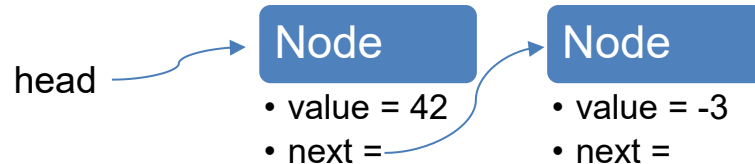
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



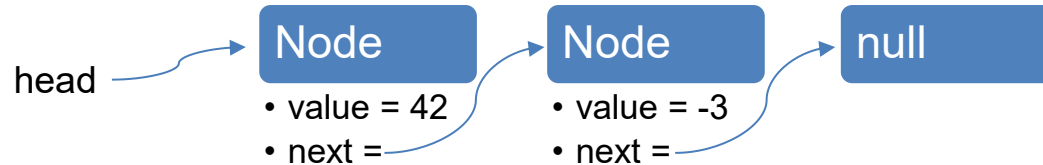
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



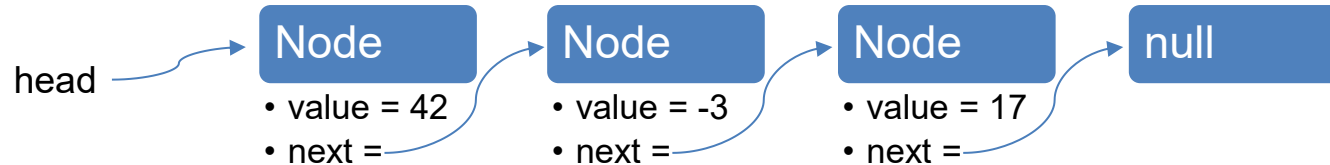
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel



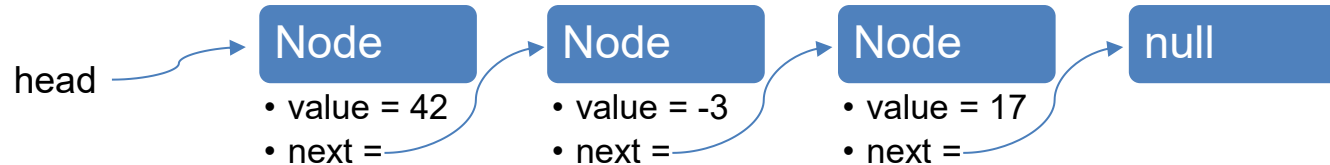
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

Beispiel

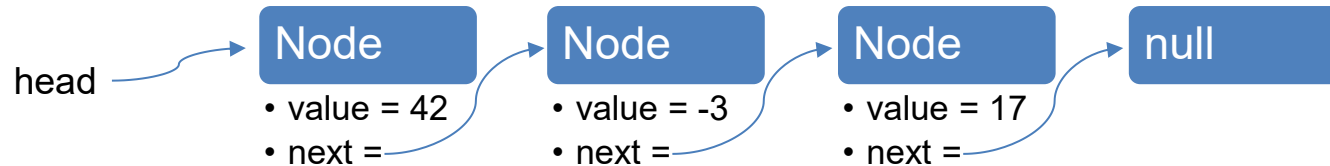


LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

head.next.next = new Node(15);
was passiert nun?

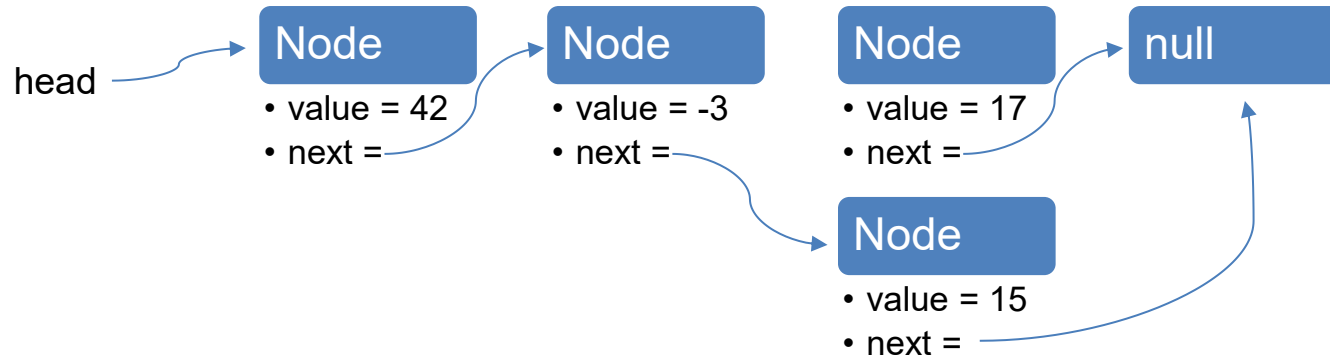


LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
head.next.next = new Node(15);  
was passiert nun?
```



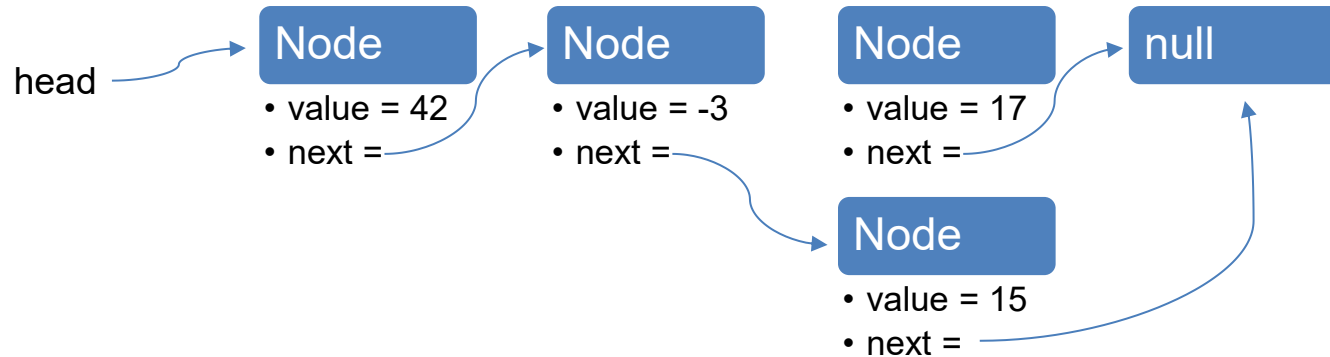
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



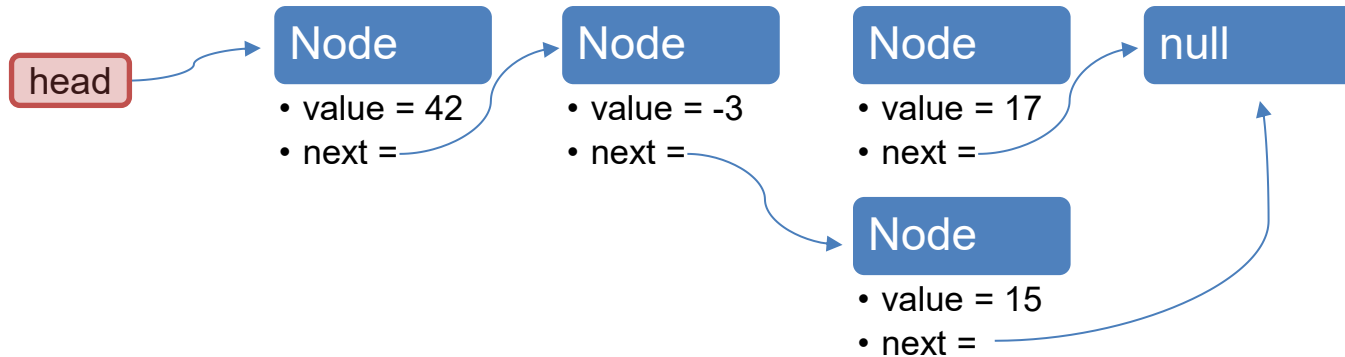
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



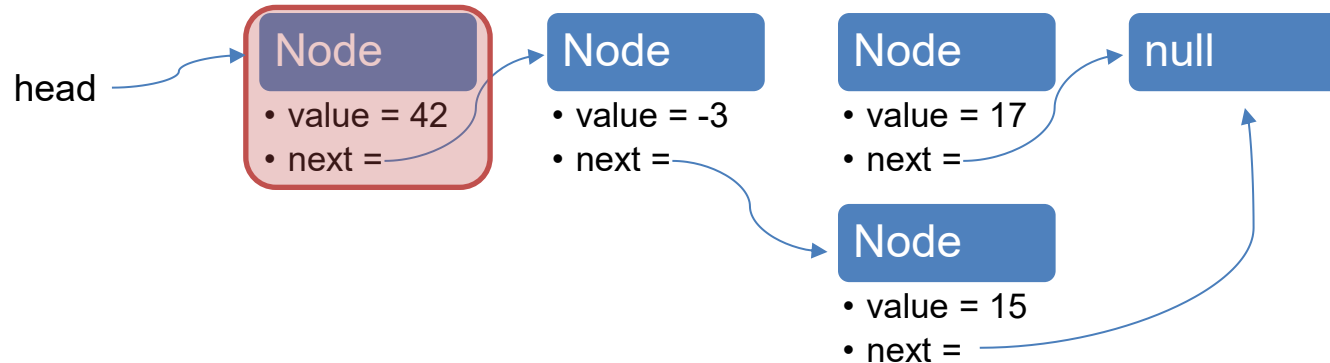
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

```
head.next.next = new Node(15);  
was passiert nun?
```

```
head.next = head.next.next;  
was passiert nun?
```



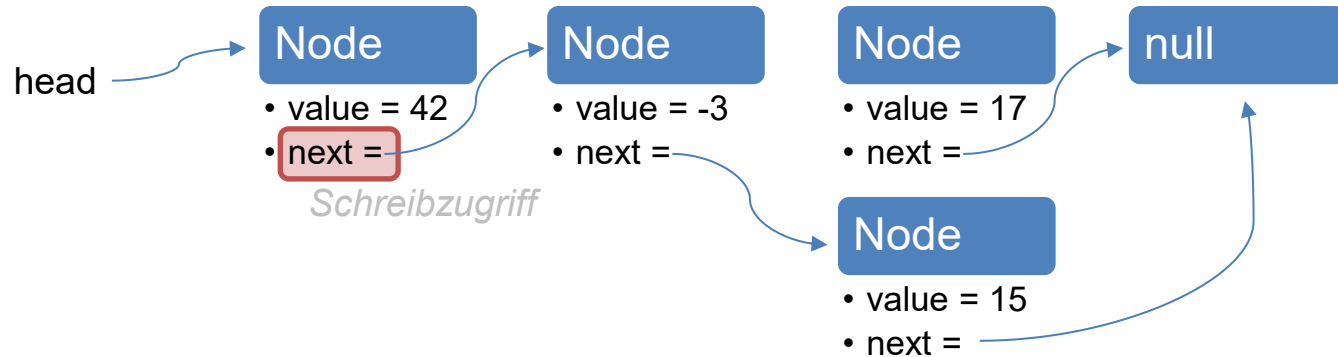
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

head.next.next = new Node(15);
was passiert nun?

head.next = head.next.next;
was passiert nun?



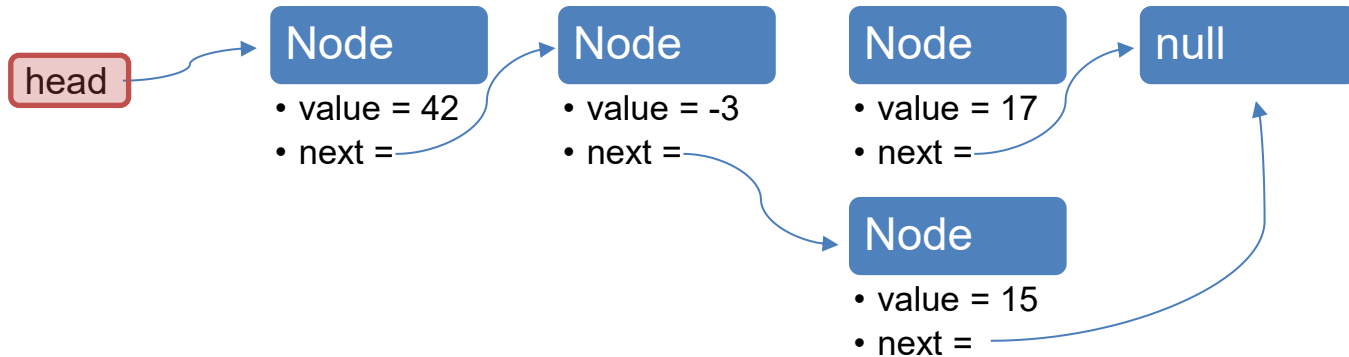
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



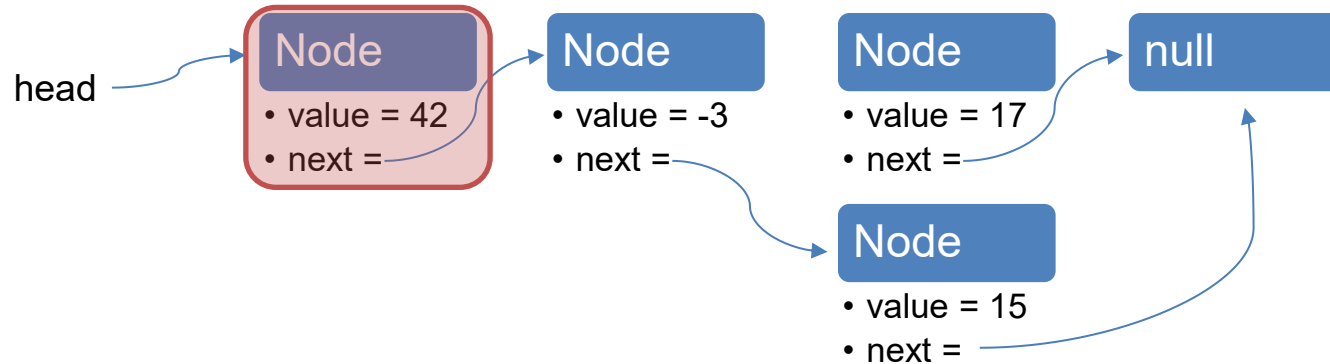
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



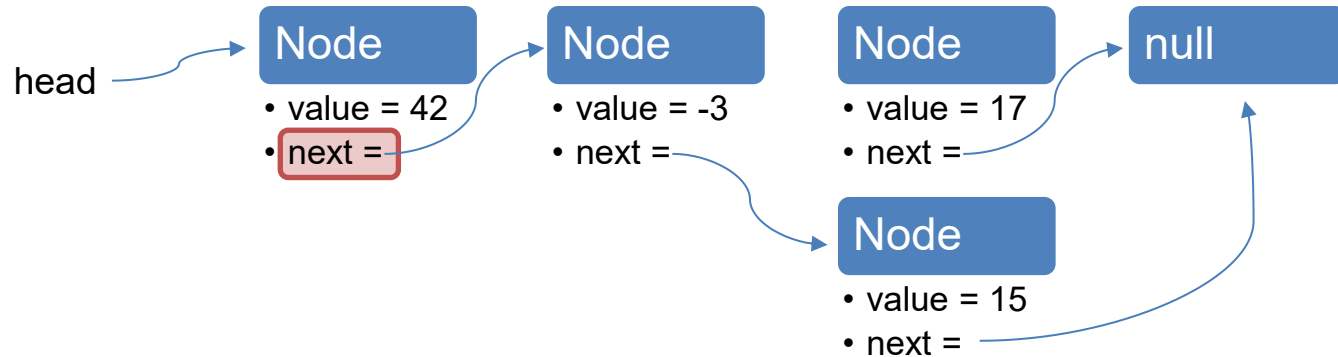
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



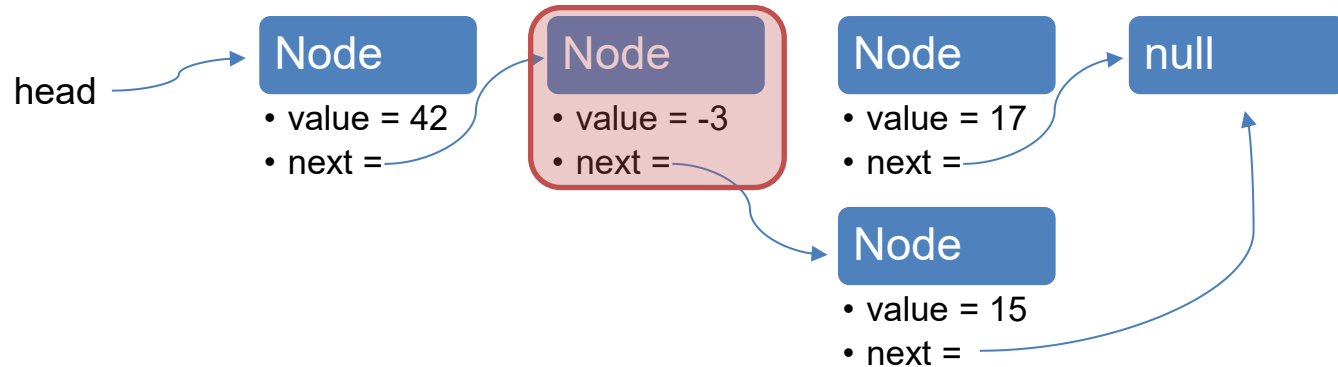
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



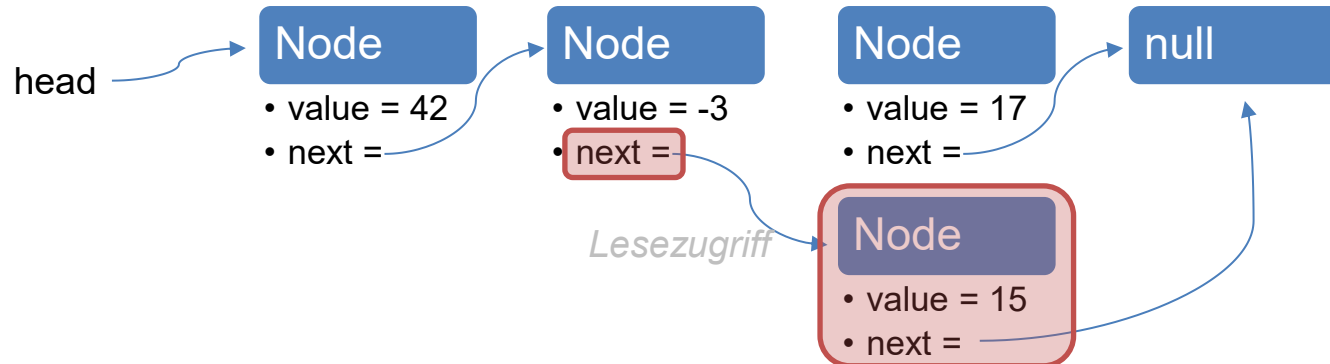
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



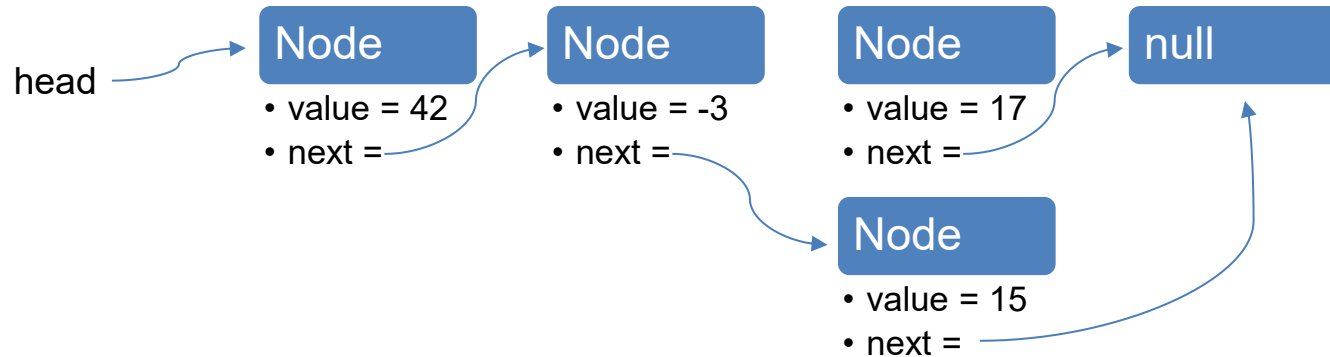
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?



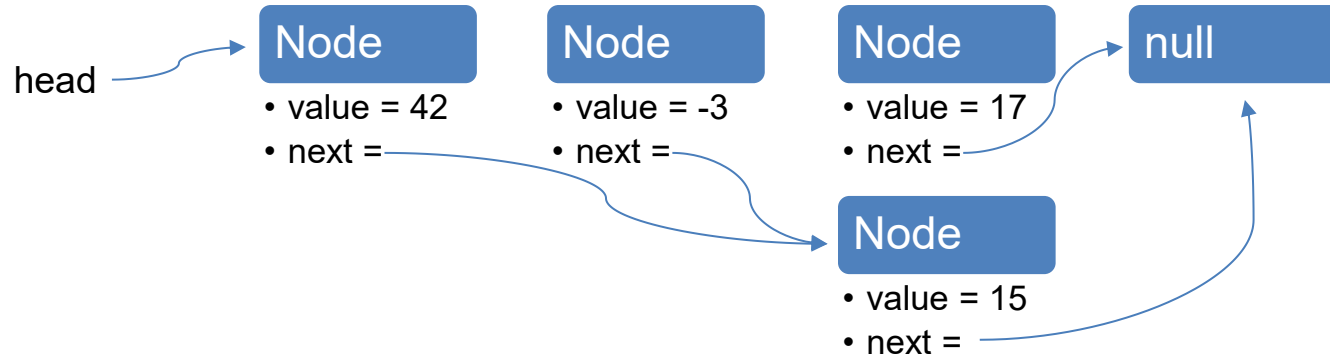
LinkedLists

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

single linked list Knotenaufbau

`head.next.next = new Node(15);`
was passiert nun?

`head.next = head.next.next;`
was passiert nun?

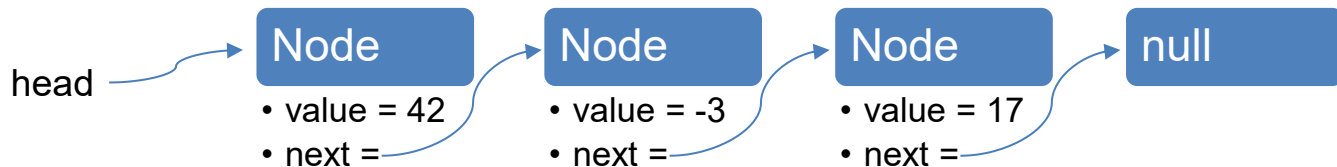


LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



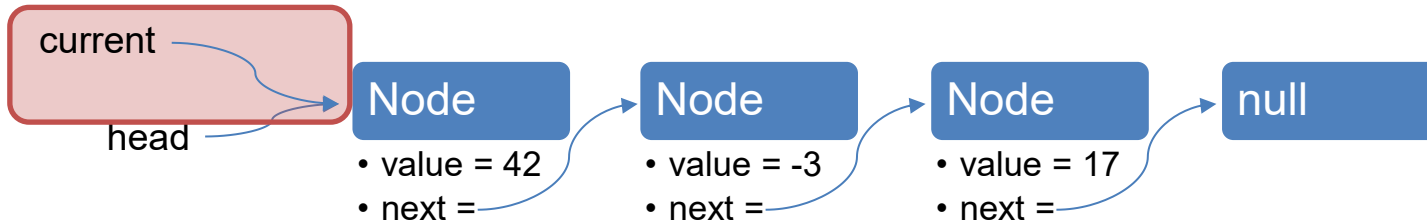
Output:

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



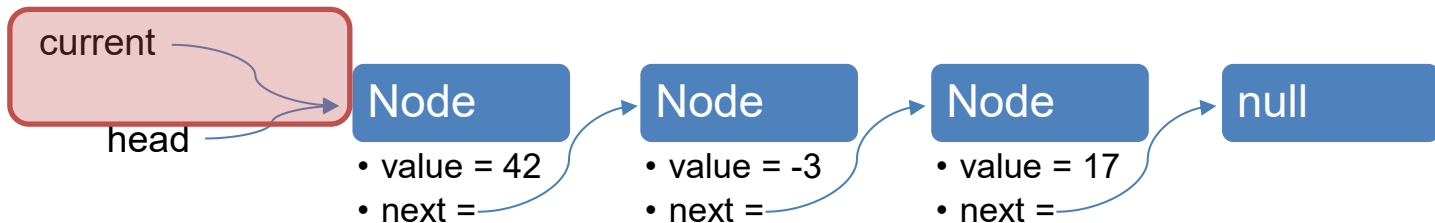
Output:

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



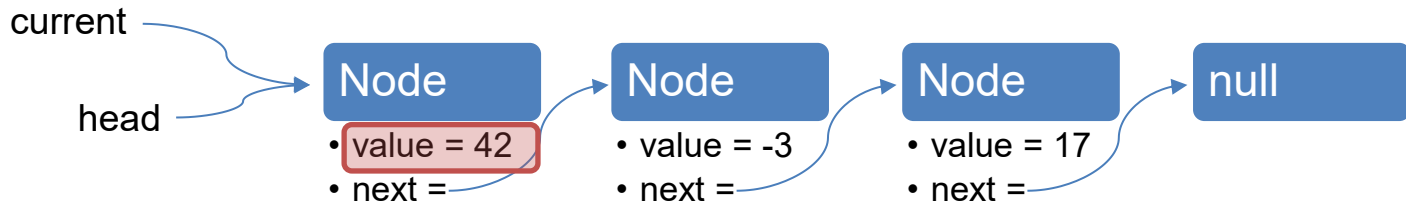
Output:

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



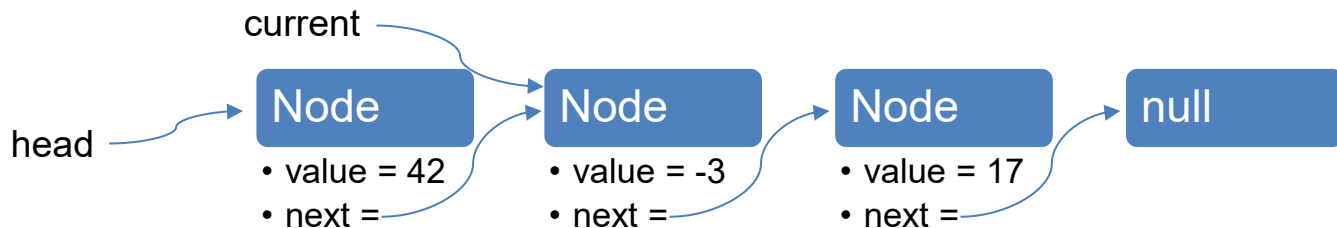
Output: 42

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



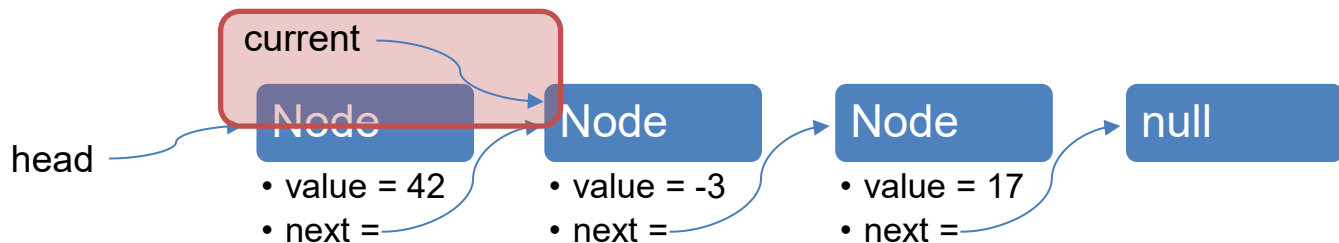
Output: 42

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



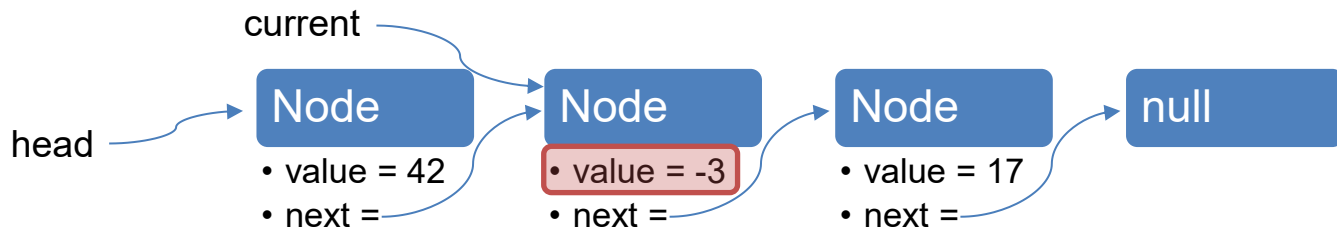
Output: 42

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



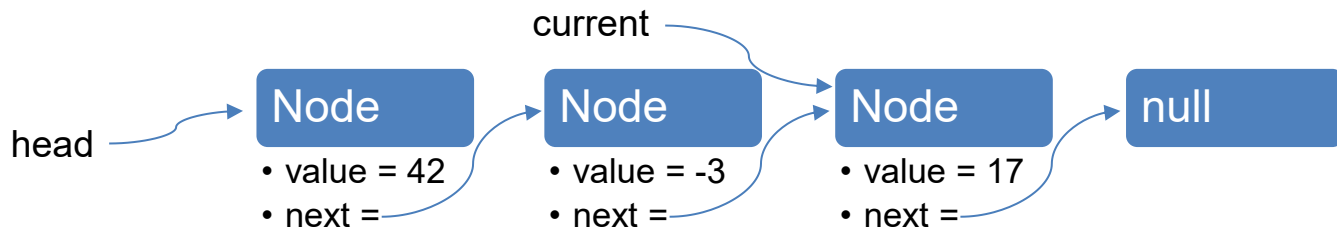
Output: 42 -3

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



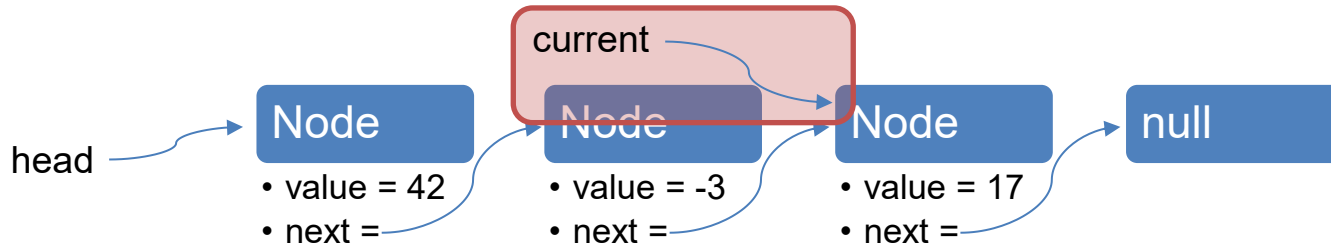
Output: 42 -3

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



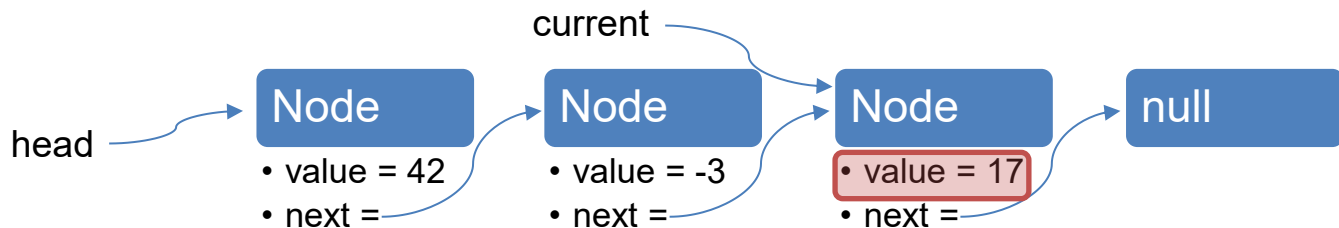
Output: 42 -3

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



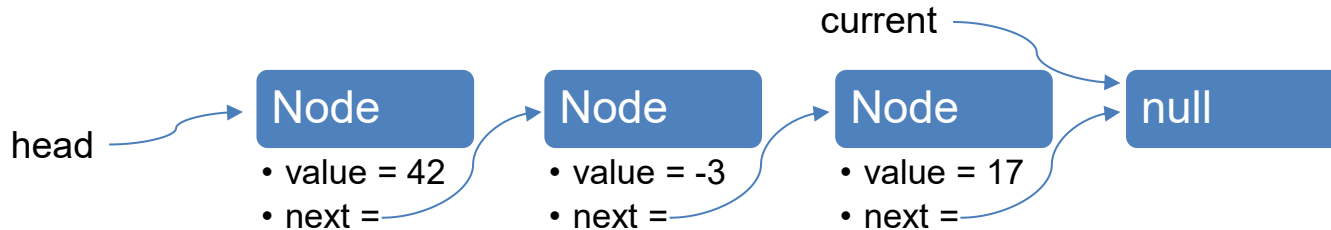
Output: 42 -3 17

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



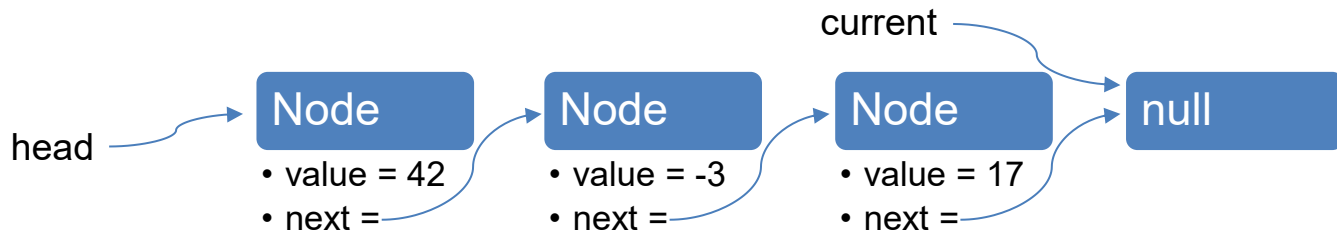
Output: 42 -3 17

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



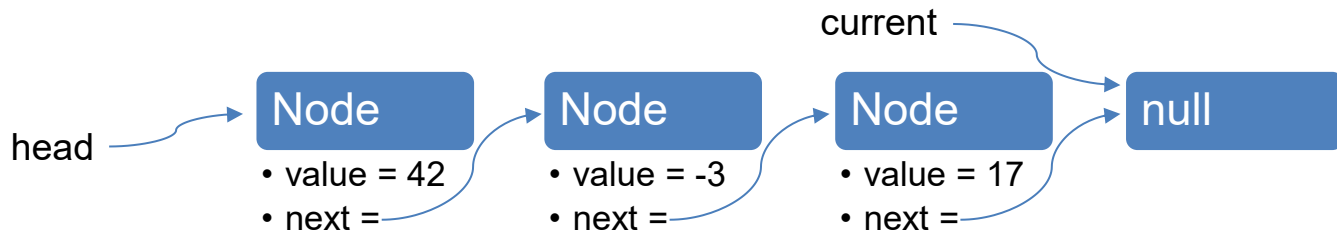
Output: 42 -3 17

LinkedLists durchlaufen

```
class Node {  
    int value;  
    Node next;  
  
    Node(int value) {  
        this.value = value;  
        this.next = null;  
    }  
}
```

```
Node head = new Node(42);  
head.next = new Node(-3);  
head.next.next = new Node(17);
```

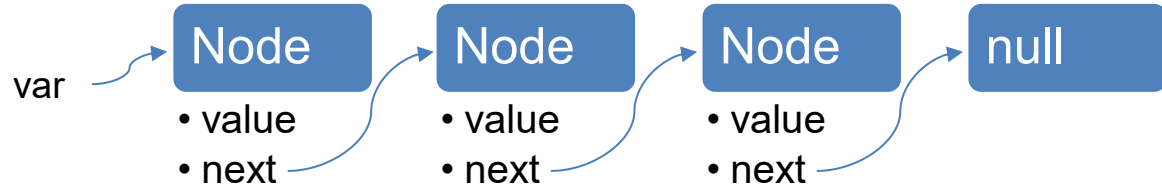
```
Node current = head;  
while (current != null) {  
    System.out.println(current.value);  
    current = current.next;  
}
```



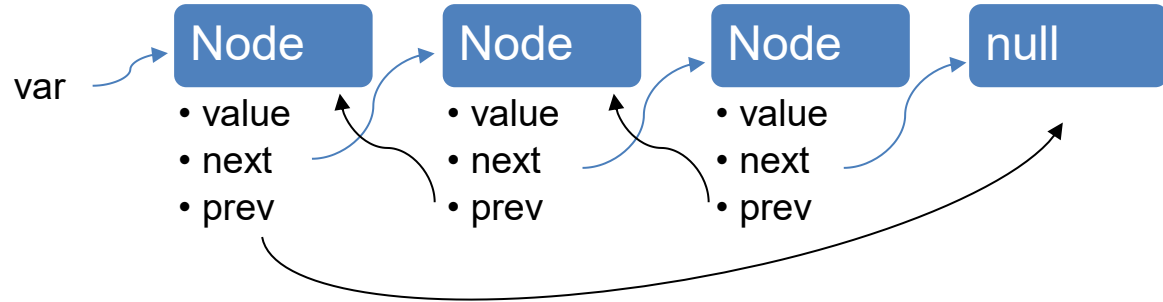
Output: 42 -3 17

Verschiede Typen von LinkedLists

single LinkedList

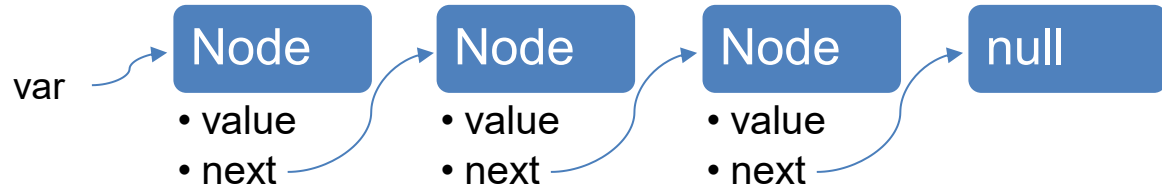


double LinkedList

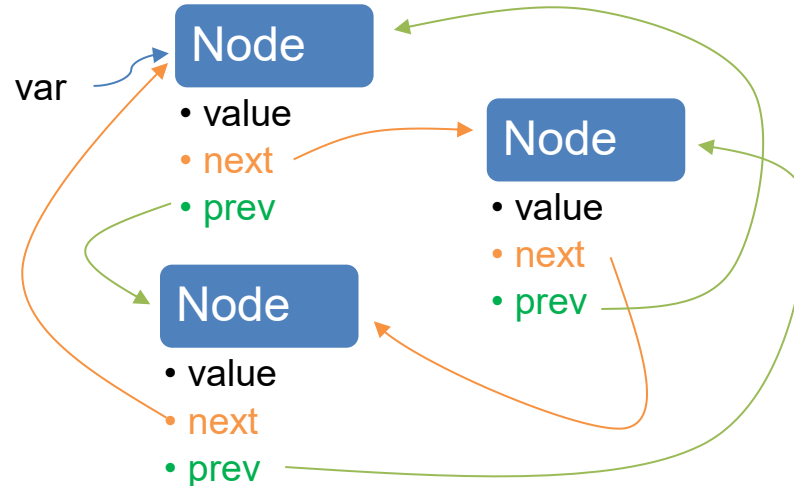


Verschiede Typen von LinkedLists

single LinkedList



cycle LinkedList



LinkedLists – bereits implementiert

- Implementation ist eine double linked list
- Import mit

```
import java.util.LinkedList;
```

- Initialisierung mit

```
LinkedList<Type> name = new LinkedList<Type>();
```



„Type“ muss ein **Referenztyp** sein

LinkedLists – bereits implementiert


- Implementation ist eine double linked list
- Import mit

```
import java.util.LinkedList;
```

- Initialisierung mit

```
LinkedList<Type> name = new LinkedList<Type>();
```

Diesen Typ
kann man auch
weglassen



„Type“ muss ein **Referenztyp** sein

LinkedLists – Methoden

Methode (LinkedList<String>)	Bedeutung
<code>list.addFirst("C");</code> <code>list.add(1, "B");</code> <code>list.add("A");</code>	Hinzufügen eines Elements <ul style="list-style-type: none">- am Anfang- an einer bestimmten Position- am Ende
<code>list.removeFirst();</code> <code>list.remove(1);</code> <code>list.removeLast();</code>	Entfernen eines Elementes <ul style="list-style-type: none">- am Anfang- an einer bestimmten Position- am Ende
<code>String element = list.get(0);</code>	Lesen eines Elementes an einer bestimmten Position
<code>int size = list.size();</code>	Länge der Liste
<code>list.contains("A");</code>	Überprüfen, ob ein Element enthalten ist
<code>list.clear();</code>	Liste leeren

alle Methoden siehe <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html>

ArrayLists

ArrayLists


- Implementation via Arrays
- Import mit

```
import java.util.ArrayList;
```

- Initialisierung mit

```
ArrayList<Type> name = new ArrayList<Type>();
```

Diesen Typ
kann man auch
weglassen



„Type“ muss ein **Referenztyp** sein

ArrayLists – Methoden

Die meisten Methoden sind absolut gleich!

alle Methoden siehe <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html>

ArrayLists – Methoden

	Array	ArrayList
Grösse	fixe Grösse	dynamische Grösse
speichert...	primitive Datentypen oder Objekte	nur Objekte (unterstützt Wrapper-Klassen)
Deklaration	<code>int[] myArray = new int[3];</code>	<code>ArrayList<Integer> myArrayList = new ArrayList<>();</code>
Lesen	<code>int x = myArray[0];</code>	<code>myArrayList.get(0);</code>
Schreiben	<code>myArray[0] = 3;</code>	<code>myArrayList.set(0, 3);</code> // 3 at index 0
Länge	<code>myArray.length;</code> // field	<code>myArrayList.size();</code> // method
Element hinzufügen	---	<code>myArrayList.add(35);</code> // 35 at the end <code>myArrayList.add(1, 35);</code> // 35 at index 1
Element entfernen	---	<code>myArrayList.remove(1);</code> // per index or <code>myArrayList.remove(e);</code> // per object e
print	<code>Arrays.toString(myArray);</code>	<code>myArrayList.toString();</code>

alle Methoden siehe <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html>

LinkedList vs ArrayList

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher		
Hinzufügen eines Elements		
Zugriffszeit		
Einfügen an einer beliebigen Position		
Einfügen am Anfang		
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements		
Zugriffszeit		
Einfügen an einer beliebigen Position		
Einfügen am Anfang		
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit		
Einfügen an einer beliebigen Position		
Einfügen am Anfang		
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position		
Einfügen am Anfang		
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position	$O(n)$	$O(n)$ wegen Verschiebung der Elemente
Einfügen am Anfang		
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position	$O(n)$	$O(n)$ wegen Verschiebung der Elemente
Einfügen am Anfang	$O(1)$	$O(n)$
Löschen am Anfang		
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position	$O(n)$	$O(n)$ wegen Verschiebung der Elemente
Einfügen am Anfang	$O(1)$	$O(n)$
Löschen am Anfang	$O(1)$	$O(n)$
Einfügen am Ende		
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position	$O(n)$	$O(n)$ wegen Verschiebung der Elemente
Einfügen am Anfang	$O(1)$	$O(n)$
Löschen am Anfang	$O(1)$	$O(n)$
Einfügen am Ende	$O(1)$	$O(1)$, falls Array gross genug
Löschen am Ende		

LinkedList vs ArrayList

Eigenschaft	LinkedList	ArrayList
Im Speicher	Doppelt verkettete Liste	Array
Hinzufügen eines Elements	$O(n)$, aber nie Kopieren aller Daten notwendig	$O(1)$, manchmal $O(n)$ durch Kopieren aller Daten
Zugriffszeit	$O(n)$	$O(1)$
Einfügen an einer beliebigen Position	$O(n)$	$O(n)$ wegen Verschiebung der Elemente
Einfügen am Anfang	$O(1)$	$O(n)$
Löschen am Anfang	$O(1)$	$O(n)$
Einfügen am Ende	$O(1)$	$O(1)$, falls Array gross genug
Löschen am Ende	$O(1)$	$O(1)$

LinkedList vs ArrayList

Für die Implementierung von **Stacks** eignet sich eine _____, da Einfügen und Entfernen am Anfang/Ende $O(1)$ sind.

Für **Queues** ist eine _____ besser geeignet, da sie effizient Einfügen am Ende und Entfernen am Anfang unterstützt, ohne Elemente zu verschieben.

LinkedList vs ArrayList

Für die Implementierung von **Stacks** eignet sich eine **LinkedList**, da Einfügen und Entfernen am Anfang/Ende $O(1)$ sind.

Für **Queues** ist eine _____ besser geeignet, da sie effizient Einfügen am Ende und Entfernen am Anfang unterstützt, ohne Elemente zu verschieben.

LinkedList vs ArrayList

Für die Implementierung von **Stacks** eignet sich eine **LinkedList**, da Einfügen und Entfernen am Anfang/Ende $O(1)$ sind.

Für **Queues** ist eine **LinkedList** besser geeignet, da sie effizient Einfügen am Ende und Entfernen am Anfang unterstützt, ohne Elemente zu verschieben.

Operation	LinkedList list	LinkedList list
Liste als Stack (Element entfernen)	<code>list.pop()</code>	<code>list.removeFirst()</code>
Liste als Stack (Element e hinzufügen)	<code>list.push(e)</code>	<code>list.addFirst(e)</code>
Liste als Queue (Element entfernen)	<code>list.poll()</code>	<code>list.removeFirst()</code>
Liste als Queue (Element e hinzufügen)	<code>list.add(e)</code>	<code>list.add(e)</code>

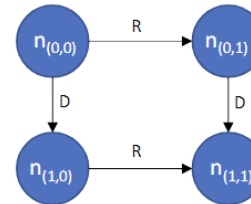
Vorbesprechung

Aufgabe 1: Square Grid

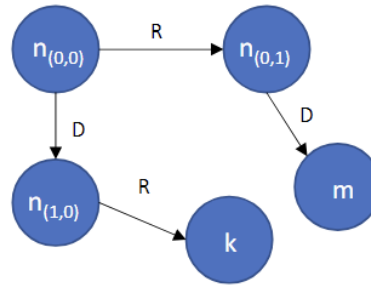
In dieser Aufgabe betrachten wir gerichtete Graphen, wobei es für jeden Knoten g höchstens zwei gerichtete Kanten von g zu anderen Knoten f, h geben kann (f, g, h können gleich sein). Wir unterscheiden dabei zwischen der rechten und der unteren Kante (und damit dem rechten und dem unteren Knoten).

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.

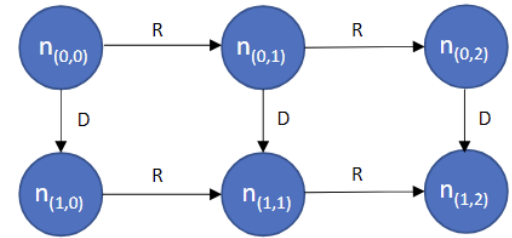
Das Ziel der Aufgabe ist, einen von einem `Node`-Objekt definierten Graphen zu analysieren. Konkret geht es darum, die Grösse des grössten quadratischen Gitters in dem Graphen zu bestimmen, der mit dem übergebenen `Node`-Objekt beschrieben wird, welches den gleichen Ursprungsknoten wie der Graph hat.



Aufgabe 1: Square Grid



(a)



(b)

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 `LinkedList` 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 `LinkedList`. 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 selbst¹.

Aufgabe 4: Klassenrätzel

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 `LinkedList`, 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 `LinkedList`-Objekt schliesslich enthält eine Referenz zum ersten und zum letzten Knoten und hat ausserdem ein Feld für die Länge der Liste.

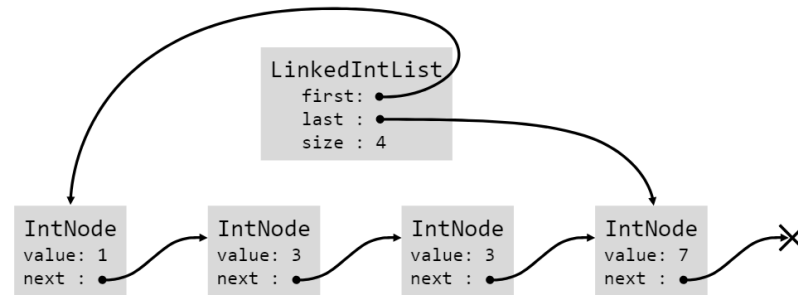


Abbildung 1: Verkettete Liste mit Werten 1, 3, 3, 7.

Aufgabe 2: Linked List

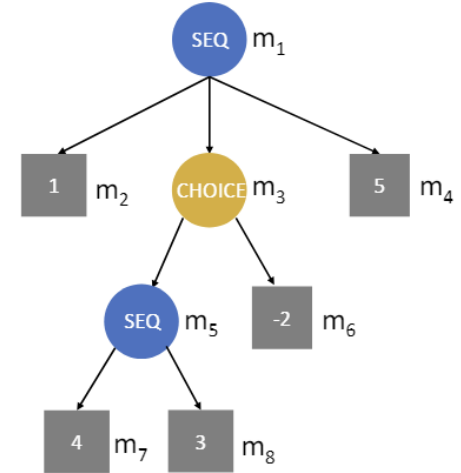
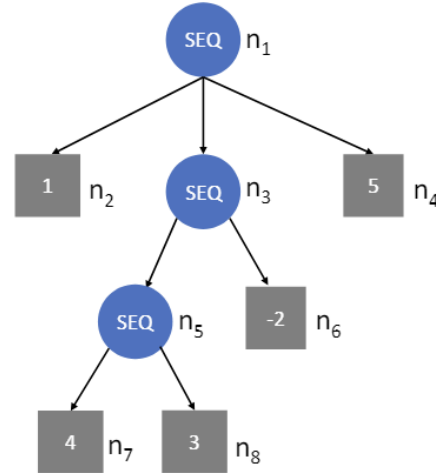
Name	Parameter	Rückg.-Typ	Beschreibung
<code>addLast</code>	<code>int value</code>	<code>void</code>	fügt einen Wert am Ende der Liste ein
<code>addFirst</code>	<code>int value</code>	<code>void</code>	fügt einen Wert am Anfang der Liste ein
<code>removeFirst</code>		<code>int</code>	entfernt den ersten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