

**252-0027**

**Einführung in die Programmierung**  
**Übungen**

**Woche 13: Sets, Maps, Graphenaufgaben**

**Henrik Pätzold**  
**Departement Informatik**  
**ETH Zürich**

# **Auswertung – Umfrage 2**

**Korrekturen Eurer Abgaben**

**Weihnachtskahoot**

# Sets - Überblick

- **Intuition**
  - Eine Instanz von einem Set soll sich in Java genauso verhalten eine mathematische Menge
  - Keine doppelten Elemente, keine zwingende Sortierung, usw.
- **Interface**
  - Das Set Interface gibt vor, welche Methoden wir auf einer Set-Instanz aufrufen können **müssen**
- **Was haben Sets, was Listen nicht haben?**
  - **Listen erlauben Duplikate**, Sets nicht.
  - **Indizes**: Listen haben eine feste Reihenfolge mit zugreifbaren Indizes (z. B. Index 0). Sets nicht.

# HashSet

- **Intuition**
  - Eine Instanz von einem HashSet verhält sich wie eine **ungeordnete Menge ohne Duplikate**.
  - Nutzt **Hashing**, um Elemente effizient zu speichern und zu finden.
- **Besonderheiten von HashSet**
  - **Keine Reihenfolge**: Die Elemente haben keine feste Reihenfolge.
  - **Schnelle Operationen**: add, remove und contains haben in der Regel eine Zeitkomplexität von  **$O(1)$** .
  - **Einzigartigkeit**: Duplikate werden automatisch entfernt.
  - **Null erlaubt**: Ein null-Element ist zulässig.
  - **HashSet** ist für jeden Typ benutzbar, weil jedes Objekt einen **HashCode** hat

# TreeSet

- **Sortierung:**
  - Die Elemente im TreeSet sind immer **sortiert**.
  - Ein TreeSet sortiert basierend auf einem Binärbaum
- **Spezielle Methoden vom TreeSet:**
  - Methoden wie `subSet()`, `headSet()` und `tailSet()` erlauben es, Teilmengen basierend auf einem Bereich zu erstellen.

# Wie iterieren wir über ein Set

- **Iterator**
  - Sei s1: Ein Beispiel-HashSet.
  - s1.iterator() gibt ein Objekt zurück, das den Zustand der Datenstruktur bei Erstellung repräsentiert
  - Zugriff auf die Elemente über den **Iterator**:
    - hasNext() prüft, ob weitere Elemente vorhanden sind.
    - next() liefert das nächste Element.
    - remove() entfernt das zuletzt zurückgegebene Element (optional)
- **Ansonsten auch einfacher mit einer foreach loop**
  - **for(String s:s1)** nutzt hasNext() und next() des Iterators



# Comparator

- **Wie weiß Java für jede Klasse, wie es zu sortieren hat?**
  - Gar nicht!
  - Wir müssen für eigene Klassen, definieren wie man die Objekte untereinander vergleicht
- **Was ist ein Comparator?**
  - Ein **Comparator** ist ein Funktionsobjekt, das verwendet wird, um die Reihenfolge von Objekten zu definieren.
- **Warum Comparator verwenden?**
  - Wenn die Klasse nicht die **natürliche Ordnung (Comparable)** implementiert.
  - Um mehrere oder flexible Vergleichslogiken zu ermöglichen.

```
TreeSet<Integer> t1 = new TreeSet<>();  
Iterator<Integer> i = t1.iterator();  
while(i.hasNext()) {  
    Integer curr = i.next();  
    // mach etwas mit i  
}
```

```
TreeSet<Integer> t1 = new TreeSet<>();  
for(Integer i : t1) {  
    // mach etwas mit i  
}
```

```
TreeSet<Integer> t1 = new TreeSet<>();

t1.add(1);
t1.add(2);
t1.add(3);
t1.add(4);
Iterator<Integer> i = t1.iterator();
t1.add(5);

while(i.hasNext()) {
    Integer curr = i.next();
    System.out.println(curr);
}
```

# Maps

# Maps - Überblick

- **Intuition**
  - Eine Map verknüpft **Schlüssel** mit **Werten**.
  - Jeder Schlüssel ist eindeutig, aber Werte dürfen dupliziert werden.
- **Interface**
  - Das Map-Interface definiert Methoden für das Hinzufügen, Entfernen und Abfragen von Schlüssel-Wert-Paaren.
- **Was haben Maps, was Sets oder Listen nicht haben?**
  - **Schlüssel-Wert-Paare:** Listen und Sets speichern nur einzelne Elemente, Maps speichern Zuordnungen.
  - **Effiziente Schlüssel-basierte Abfragen:** Direkter Zugriff auf Werte über Schlüssel (z. B. `map.get(key)`).

# Maps – Konkrete Implementationen

- **HashMap**

- **Schlüssel-Wert-Paare:**

- Speichert Schlüssel-Wert-Paare basierend auf dem Hashcode des Schlüssels.

- Analog zu HashSet, nur das wir mit den Keys auf die Buckets zugreifen können.

- **TreeMap**

- **Schlüssel-Wert-Paare:**

- Speichert Schlüssel-Wert-Paare in **natürlicher Ordnung** oder nach einem **benutzerdefinierten Comparator**.

- **Sortiert:**

- Die Einträge sind sortiert basierend auf der Ordnung der Schlüssel.

# Maps – Nachtrag

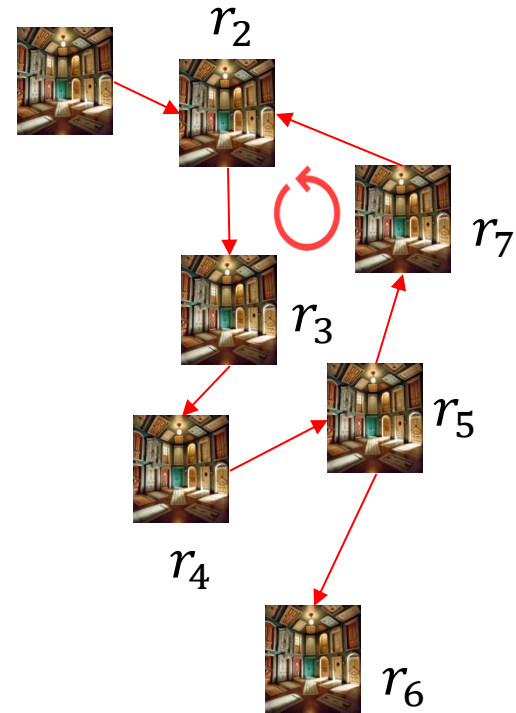
- **Option 1:** HashMap
  - Die Methoden equals und hashCode der Objekt-Klasse überschreiben. (Hier gibt es einen nützlichen Eclipse-Trick).
- **Option 2:** TreeMap
  - Comparable-Interface implementieren und compareTo-Methode überschreiben.

# **Zyklen finden mit Sets**



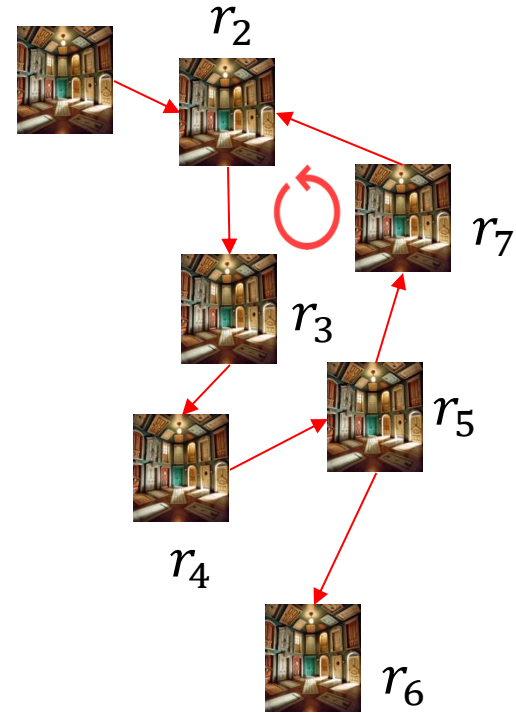
# Zyklen finden

- **Oft:** Annahme, dass keine Zyklen vorkommen.
- **Problem:** Was wenn doch Zyklen vorkommen dürfen?





# Zyklen finden

- **Option 1:** Modifizieren der Datenstruktur mit einem visited Attribut. (u08) ✓
- **Option 2:** Nutzen von Sets um besuchte Nodes zu speichern.



# Zyklen finden

- **Wie unterscheiden wir zwei Objekte?**
  - equals können wir nicht immer nutzen!

equals(  ,  )  
          age = 2                      age = 2  
          Room@29ca901e          Room@5025a98f



true



```
public boolean equals(Object o) {  
    if(o instanceof Room) {  
        o = (Room)o;  
        if(this.age == o.age) {  
            return true;  
        }  
    }  
    return false;  
}
```

# Zyklen finden

- **Wie unterscheiden wir zwei Objekte?**
  - Hier können wir Referenzen vergleichen!

`equals(`  `,`  `)`  
                    age = 2                      age = 2  
                    Room@29ca901e          Room@5025a98f

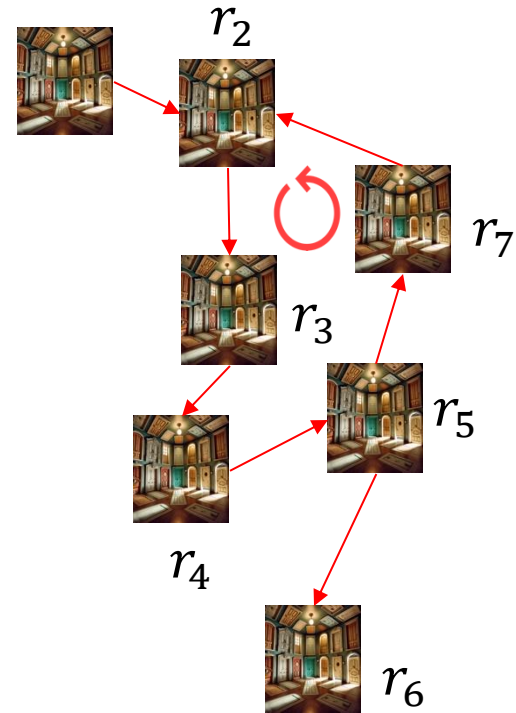
false



```
public boolean equals(Object o) {  
    if(o instanceof Room) {  
        if(this == o) {  
            return true;  
        }  
    }  
    return false;  
}
```

# Zyklen finden

- **Mit Sets:** Nutzen von `Set<Room>` um alle besuchten Nodes zu speichern.
- **Option 1:** `HashSet<Room>`
- **Option 2:** `TreeSet<Room>`

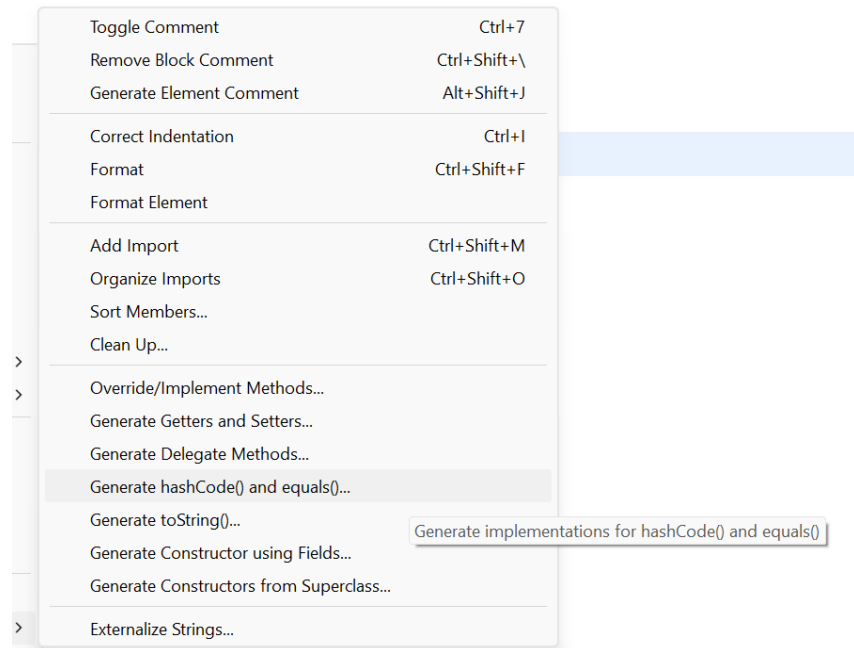


# Zyklen finden

- **Option 1:** `HashSet<Room>`
  - Die Methoden `equals` und `hashCode` der Objekt-Klasse überschreiben.
- **Option 2:** `TreeSet<Room>`
  - `Comparable`-Interface implementieren und `compareTo`-Methode überschreiben.

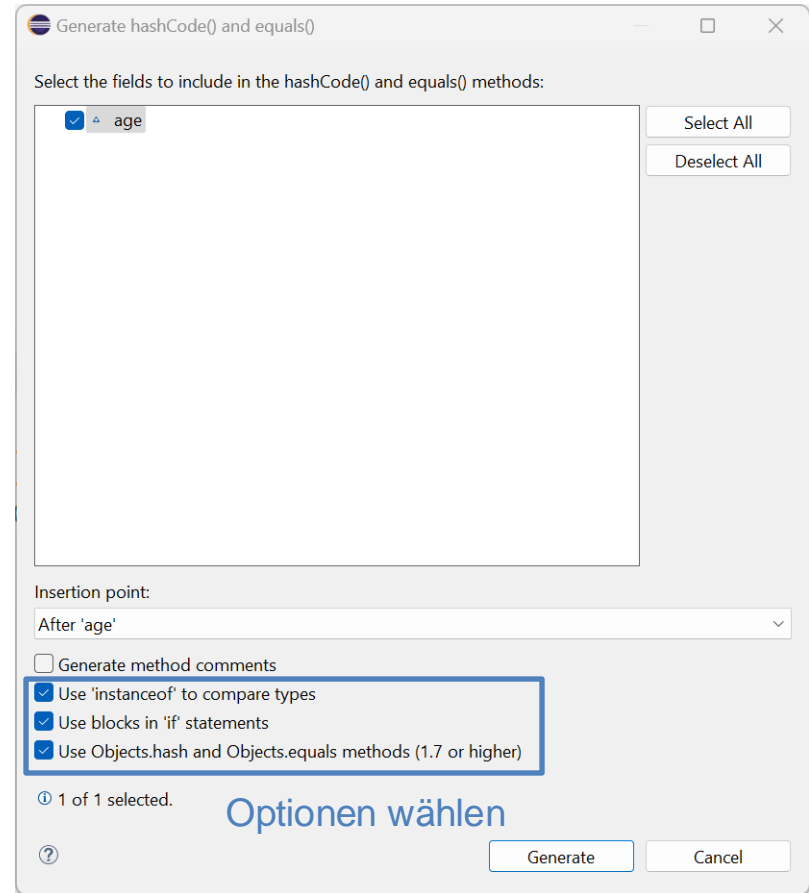
# Zyklen finden: HashSet

- **Option 1: HashSet<Room>**
  - **In Eclipse:** Rechtsclick -> Source -> Generate hashCode() und equals()...



# Zyklen finden: HashSet

- **Option 1:** HashSet<Room>
  - **In Eclipse:** Rechtsclick -> Source -> Generate hashCode() und equals()...






# Zyklen finden: HashSet

- Option 1: HashSet<Room>

```
@Override
public int hashCode() {
    return Objects.hash(age);
}
```

Wir benutzen Operationen und Methoden von Superklassen um hashCode zu implementieren.



```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

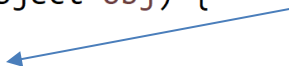
# Zyklen finden: HashSet

- Option 1: HashSet<Room>

```
@Override  
public int hashCode() {  
    return Objects.hash(age);  
}
```

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (!(obj instanceof Room)) {  
        return false;  
    }  
    Room other = (Room) obj;  
    return age == other.age;  
}
```

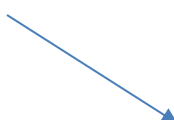
Genau gleiche  
Objekte sollten auch  
immer equals sein.



# Zyklen finden: HashSet

- Option 1: HashSet<Room>

Objekte von  
unterschiedlichem  
Typ sind nie equals.



```
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

# Zyklen finden: HashSet

- Option 1: HashSet<Room>

```
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Casten damit der  
Compiler Zugriff auf  
die Attribute erlaubt.



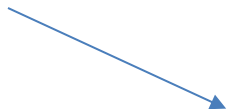
# Zyklen finden: HashSet

- Option 1: HashSet<Room>

```
@Override
public int hashCode() {
    return Objects.hash(age);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Room)) {
        return false;
    }
    Room other = (Room) obj;
    return age == other.age;
}
```

Attribute vergleichen.



# Zyklen finden: TreeSet

- **Option 2: TreeSet<Room>** Damit TreeSet funktioniert muss Room das Comparable-Interface implementieren.

```
public class Room implements Comparable<Room>{  
  
    int age;  
  
    @Override  
    public int compareTo(Room o) {  
        // TODO Auto-generated method stub  
        return 0;  
    }  
}
```

# Zyklen finden: TreeSet

- Option 2: TreeSet<Room>

```
public class Room implements Comparable<Room>{  
  
    int age;  
  
    @Override  
    public int compareTo(Room o) {  
        // TODO Auto-generated method stub  
        return 0;  
    }  
}
```

Jetzt müssen wir nur noch die compareTo-Methode sinnvoll implementieren.

# Zyklen finden: TreeSet

- **Option 2: TreeSet<Room>**
  - compareTo gibt 0 zurück, falls die Objekte equals sind.
  - compareTo gibt 1 zurück, falls this-Objekt „grösser“ als das Parameter-Objekt ist.
  - compareTo gibt -1 zurück, falls this-Objekt „kleiner“ als das Parameter-Objekt ist.



# Zyklen finden: TreeSet

- Option 2: TreeSet<Room>

```
@Override
public int compareTo(Room o) {
    if(this.equals(o)) {
        return 0;
    } else if(this.age > o.age) {
        return 1;
    } else {
        return -1;
    }
}
```

# Zyklen finden: TreeSet

- **Option 2: TreeSet<Room>**
  - Wenn Comparable implementiert ist, dann können wir mit `Collections.sort` sortieren.
  - Wenn Comparable implementiert ist, dann können wir mittels `PriorityQueue<Room>` einen Heap erstellen.
  - Mit `Collections.reverseOrder()` können wir von aufsteigender Ordnung zu absteigender Ordnung wechseln.

# Zyklen finden: TreeSet

- **Option 2: TreeSet<Room>**
  - Wenn Comparable implementiert ist, dann können wir mit `Collections.sort` sortieren.

```
public void sortList(List<Room> rooms) {  
    Collections.sort(rooms);  
}
```

# Zyklen finden: TreeSet

- **Option 2: TreeSet<Room>**
  - Wenn Comparable implementiert ist, dann können wir mittels PriorityQueue<Room> einen Heap erstellen.

```
public PriorityQueue<Room> priorityQueueFromList(List<Room> rooms) {  
    PriorityQueue<Room> pQueue = new PriorityQueue<Room>(rooms);  
    return pQueue;  
}
```

# Zyklen finden: TreeSet

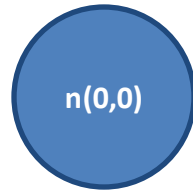
- **Option 2:** TreeSet<Room>
  - Mit `Collections.reverseOrder()` können wir von aufsteigender Ordnung zu absteigender Ordnung wechseln.

```
public void sortListReversed(List<Room> rooms) {  
    Collections.sort(rooms, Collections.reverseOrder());  
}
```

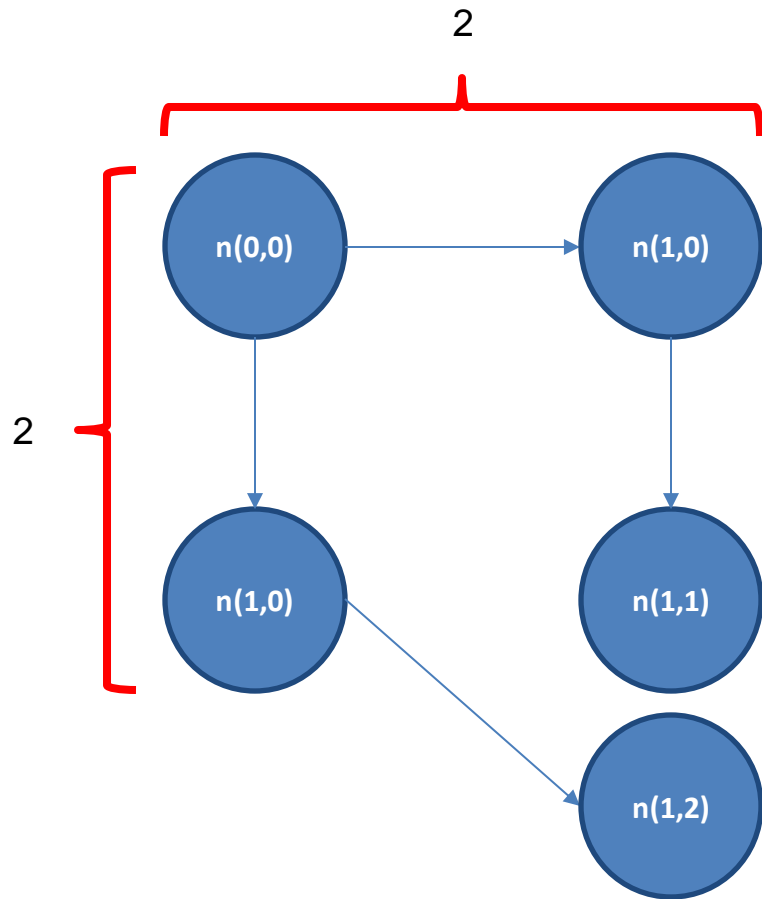
# Graphenaufgaben

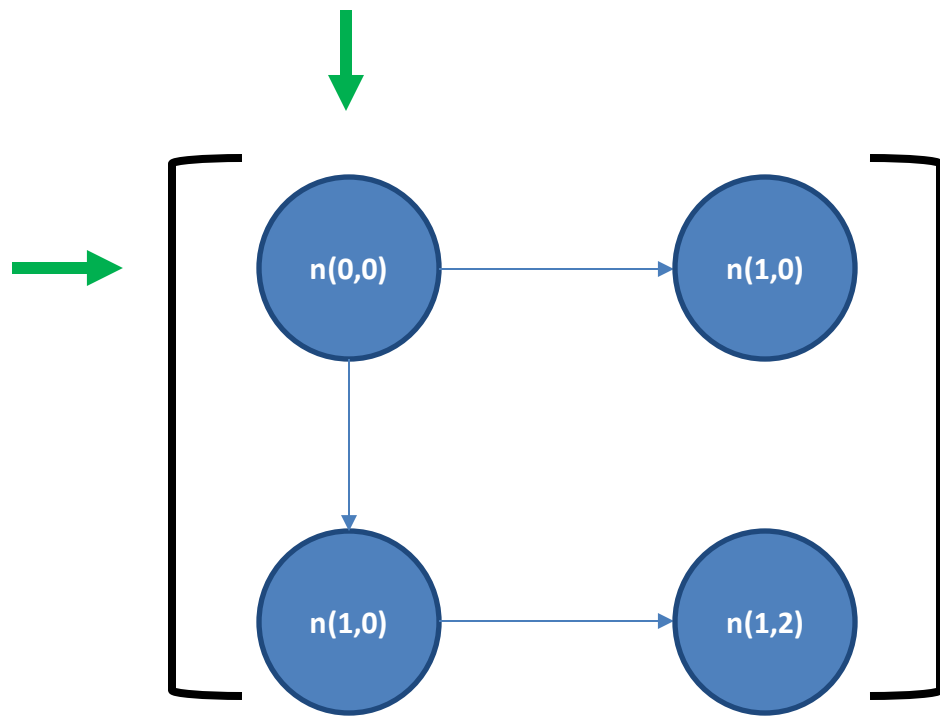
# Trick zum lösen *vieler* Graphenaufgaben

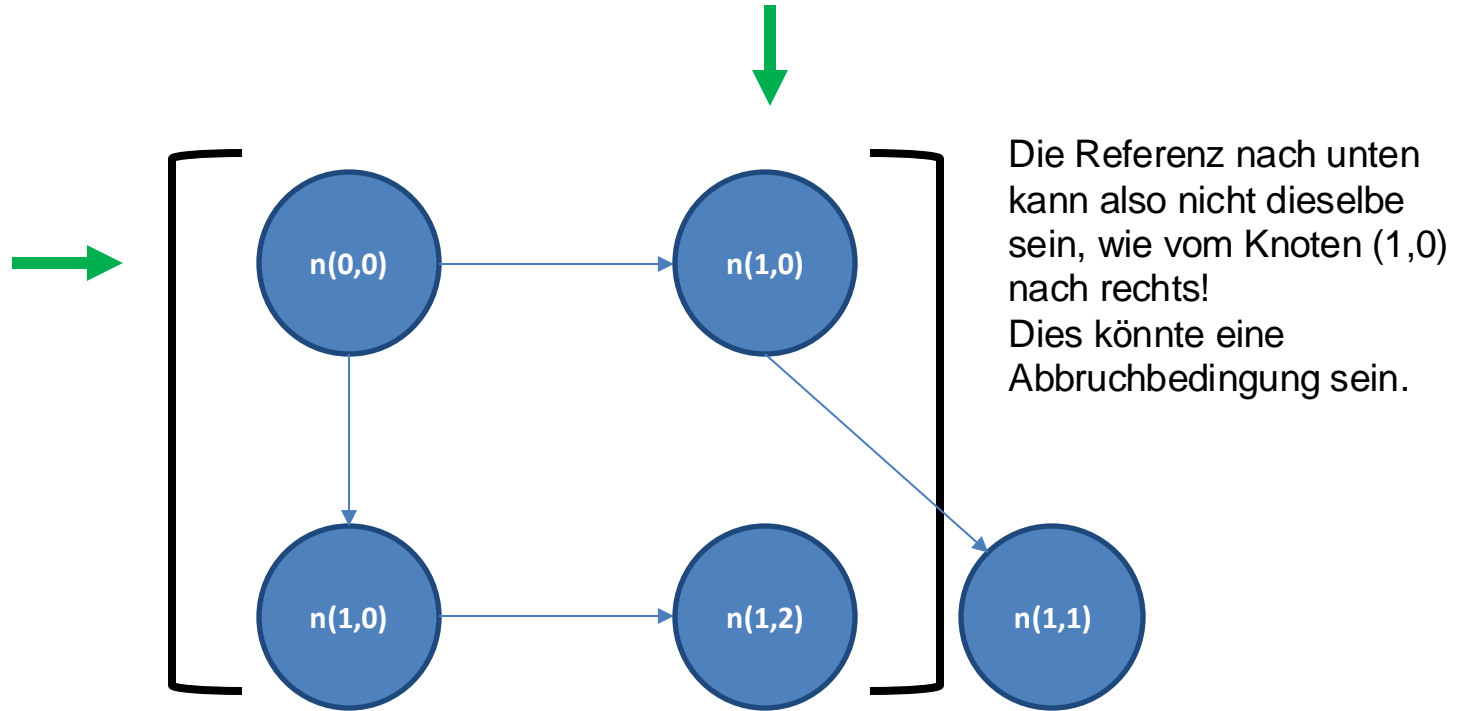
- **Rekursion**
  - Gegebene Daten (oft einzelner Knoten) rekursiv abarbeiten und wichtige Merkmale speichern. **Wieso rekursiv?**
- **Daten in richtige Form bringen**
  - Graph in einer Matrix o.Ä. speichern
- **Bedingungen iterativ überprüfen**
  - Die zu überprüfenden Bedingungen stehen in der Aufgabenbeschreibung
  - Iterativ kann ist das jetzt einfacher zu lösen durch konsequentes Überprüfen der Bedingungen

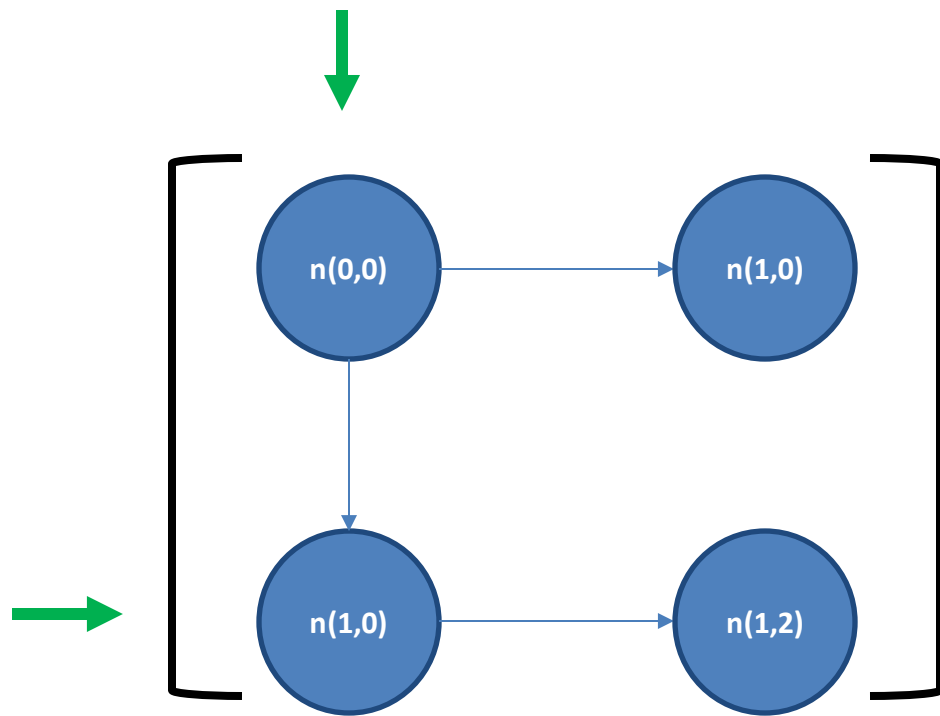


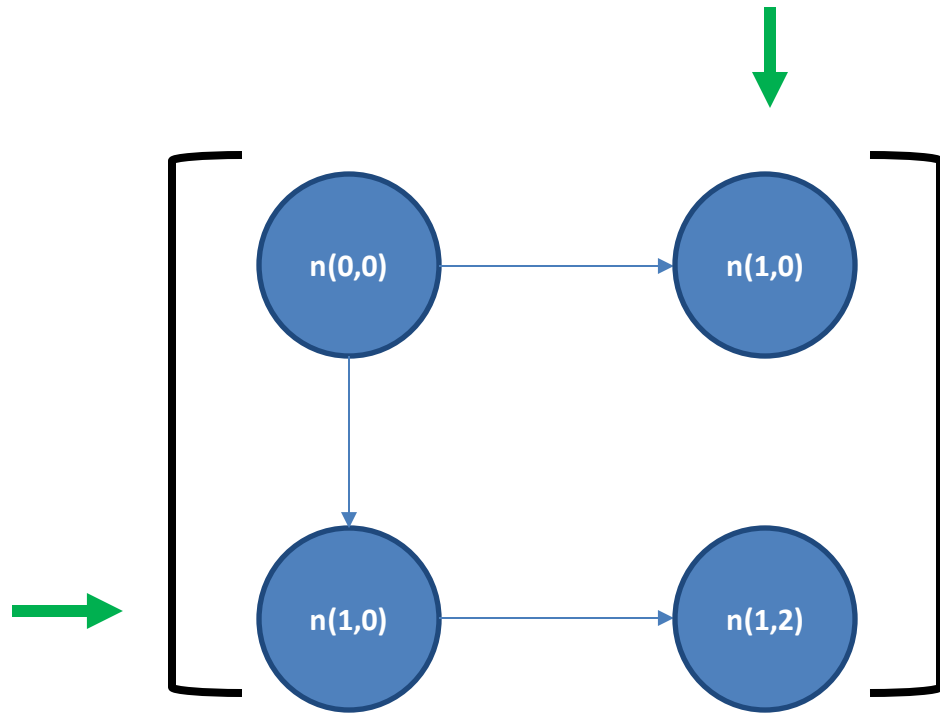












**Pyramide (U10)**