Ontologien - Recap

Definition: Eine formale Darstellung von Wissen als eine Menge von Konzepten und die Beziehungen zwischen ihnen.

Eine Ontologie ist wie eine **Bauplan** für die Repräsentation von Wissen. Es besteht aus:

- Klassen
- Instanzen
- Eigenschaften

Knowledge Graphs

Definition: Ein Knowledge Graph (KG) ist eine Ontologie, die zusammen mit ihren Instanzen und Daten als Graph dargestellt wird.

KG = Ontologie (Bauplan) + Instanzen (Daten) + Graphenstruktur

Kgs sind auf dem Konzept der Triple basiert – wie Ontologien:

<Subjekt – Prädikat - Objekt>

KG – Datanbank Analogoie

In Datenbanken, wie z. B. SQL-Datenbanken, definiert das **Schema** die Struktur - wie ein Bauplan. Es legt fest, welche Tabellen es gibt, welche Spalten sie haben und welche Datentypen zulässig sind.

Eine **Datenzeile** (Record) sind die eigentlichen Daten - ein bestimmter Satz von Werten, die das Schema ausfüllen.

KG – Datanbank Analogoie

Schicht	**Datenbank**
Ontologie	Schema
Instanzen	Datenzeile
Knowledge Graph	Datenzeile + Schema

Graphen

KG sind dargestellt durch gerichteten, beschrifteten Graphen. Ein Graph ist eine Struktur, die ein Netzwerk beschreibt. Er ist mathematisch als Tupel definiert, mit einer Menge von Nodes (Knoten) und Edges (Verbindungen).

- Graph $G = (V, E)$
- Nodes $V = \{v_1, v_2, ...\}$
- Edges $E = \{e_1, e_2, ...\}$

- Für Nodes u und v in einem gerichteten Graphen definieren wir eine Edge von u nach v als $e = (u, v)$ für $u,v \in V$.

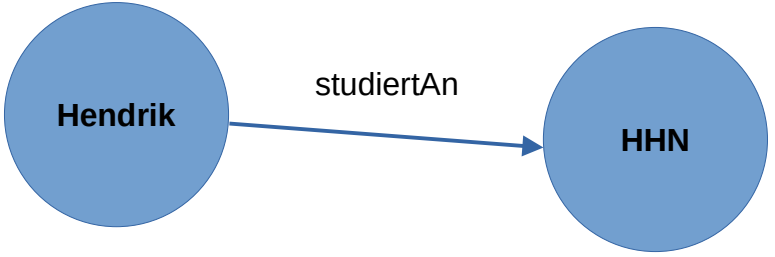
RDF Triples - KG Implementation

Ein **Triple-Store** ist eine Datenbank, die für die Speicherung von RDF-Triples konzipiert ist. Ein RDF-Triple wird als gerichtete Edge (Prädikat) zwischen einem Node-Pair (Subjekt und Objekt) ausgedruckt. Ein KG-Node steht für eine Klasse oder Instanz, während ein Edge eine Eigenschaft darstellt.

Beispiel:

```
:Hendrik rdfs:studiertAn :HHN
```

$e_{studiertAn} = (Hendrik, HHN)$



KG Beispiel – Autonomes Fahren

Als Beispiel nehmen wir das autonom fahrende Auto aus der letzten Teamübung. Wir erstellen zunächst einen Triple-Store mit der Ontologie (Klassen und Eigenschaften) und den Daten (Instanzen). Anschließend wird der Knowledge Graph unter Verwendung der RDF-Triples im Triple-Store erstellt.

Triple-Store Klassen

- **FZG** → `:Fahrzeug rdf:type rdf:Class`
- **FZGT** → `:Fahrzeugteil rdf:type rdf:Class`
- **SYSZ** → `:Systemzustand rdf:type rdf:Class`
- **SYSK** → `:SystemKomponent rdf:type rdf:Class`
- **SNSR** → `:Sensor rdfs:subClassOf :SystemKomponent`
- **AKTR** → `:Aktor rdfs:subClassOf :SystemKomponent`

Triple-Store Eigenschaften

- **E1** → `löstAus rdf:type rdf:Property ;`

  `rdfs:domain :Sensor`

  `rdfs:range :Aktor`

- **E2** → `veranlasst rdf:type rdf:Property ;`

  `rdfs:domain :Aktor`

  `rdfs:range :Systemzustand`

- **E3** → `istTeilVon rdf:type rdf:Property ;`

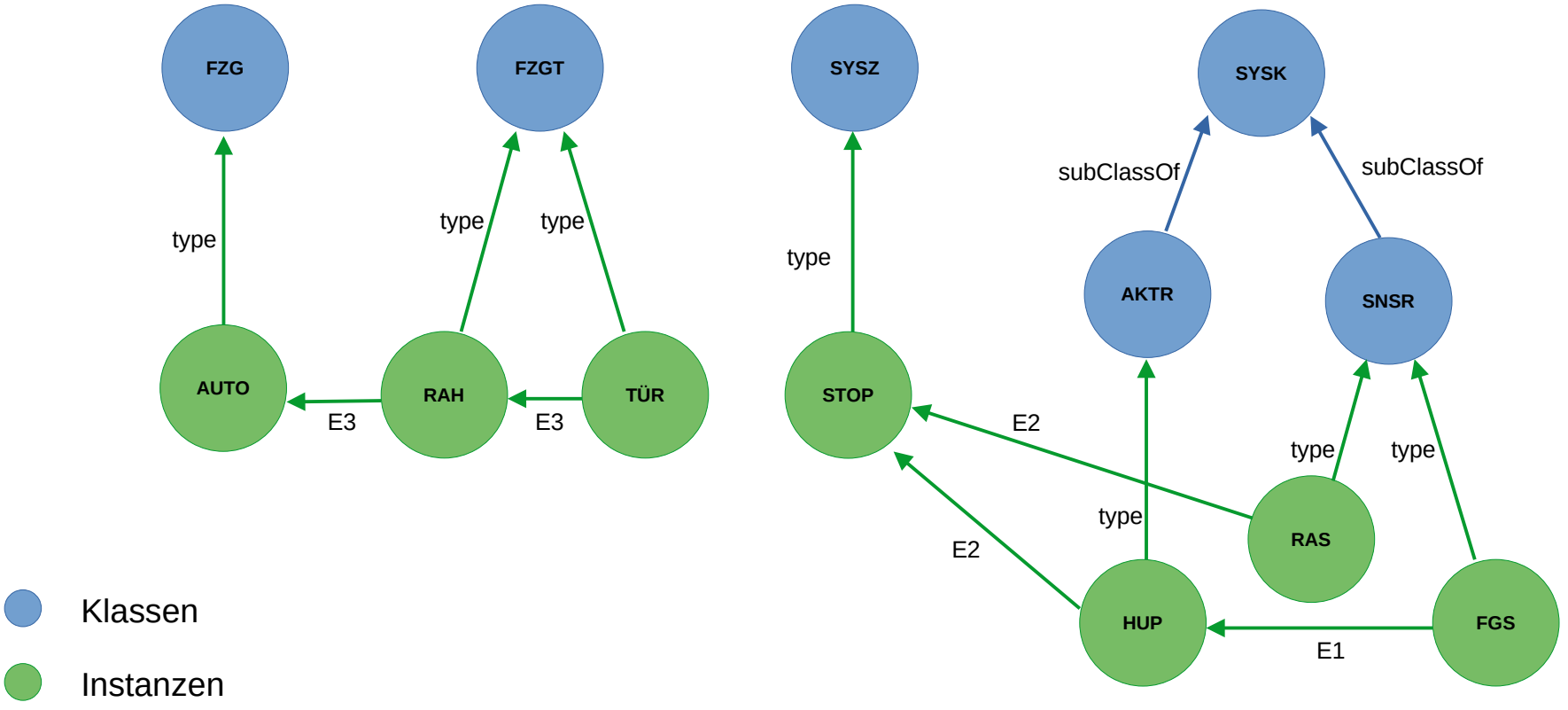
  `rdfs:domain :Fahrzeugteil`

  `rdfs:range` **`[ a owl:Class owl:unionOf ( :FahrzeugTeil :Fahrzeug ) ] .`**

Triple-Store Instanzen

- **AUTO** → `:AutonomesAuto rdf:type :Fahrzeug`
- **RAH** → `:Rahmen rdf:type :Fahrzeugteil`
- **TÜR** → `:Tür rdf:type :Fahrzeugteil`
- **FGS** → `:FussgaengerSensor rdf:type :Sensor`
- **RAS** → `:RoterAmpelSensor rdf:type :Sensor`
- **HUP** → `:Hupe rdf:type :Aktor`
- **STOP** → `:Stop rdf:type :Systemzustand`

KG – Autonomes Fahren



KG – Inferenz

Inferenz in einem Knowledge Graph führt zu neuem **Wissen** in Form von *neuen Edges* im Graphen. In einem KG erfolgt die Inferenz durch Axiome und Regeln. In diesem Kurs behandeln wir drei Arten:

1. Klassen-Unterordnung (Class-Subsumption): Wenn die Klasse A eine Unterklasse der Klasse B ist $(A \subseteq B)$, dann ist jede Instanz von A auch eine Instanz von B.

2. Eigenschaftskette (Property Chains): Wenn A mit B und B mit C verwandt ist, dann ist A mit C verwandt (durch eine neue oder bestehende Eigenschaft).

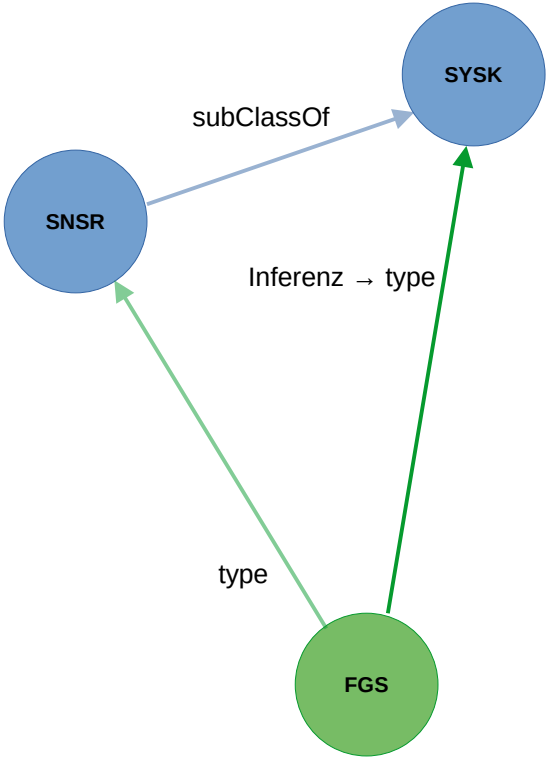
3. Transitivität (Transitivity): Wenn A mit B verbunden ist und B mit C verbunden ist, dann ist A mit C verbunden - mit der gleichen Eigenschaft.

Beispiel – Klassen-Unterordnung

- `:Sensor rdfs:subClassOf :SystemKomponent`
- `:FussgaengerSensor rdf:type :Sensor`

Inferenz

→ `:FussgaengerSensor rdf:type :SystemKomponent`



Beispiel – Eigenschaftskette

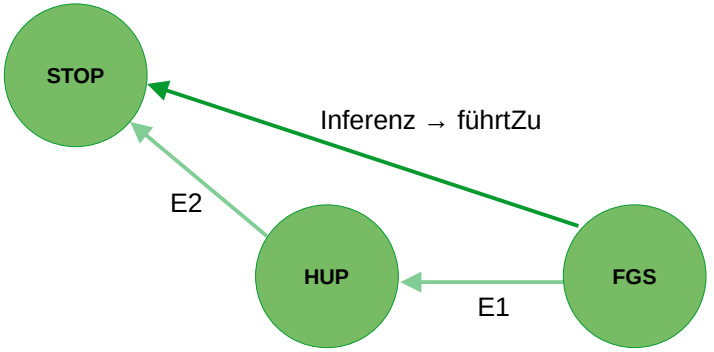
Axiom

- `:führtZu owl:propertyChainAxiom ( :löstAus :veranlasst ) .`

- `:FussgaengerSensor :löstAus :Hupe`
- `:Hupe :veranlasst :Stop`

Inferenz

→ `:FussgaengerSensor :führtZu :Stop`



Beispiel – Transivität

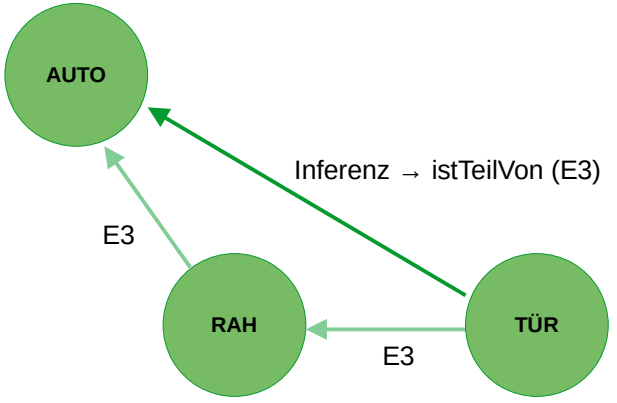
Axiom

- `:istTeilVon rdf:type owl:TransitiveProperty .`

- `:Tür :istTeilVon :Rahmen`
- `:Rahmen :istTeilVon :AutonomesAuto`

Inferenz

→ `:Tür :istTeilVon :AutonomesAuto`



KG – Abfragen (Querying)

Während die Inferenz dazu dient, neues Wissen in einer KG zu schaffen, wird die **Abfrage** dazu verwendet, *nützliche Fakten aus dem Graphen zu extrahieren*. Abfragen beantworten Fragen wie:

- Wie viele der Komponenten des autonomen Fahrzeugsteuerungssystems sind Sensoren?
- Führt der Fußgängersensor in einem autonomen Auto dazu, dass das Auto anhält?

Obwohl es Methoden für die Abfrage in KGs gibt (SPARQL, Cypher usw.), werden wir uns mit der eigentlichen Theorie der Suche in einem Graphen befassen.

Abfrage als Suchproblem

Bei Graphen gibt es zwei wichtige Suchalgorithmen: die Breitensuche (breadth-first-search, BFS) und die Tiefensuche (depth-first-search, DFS).

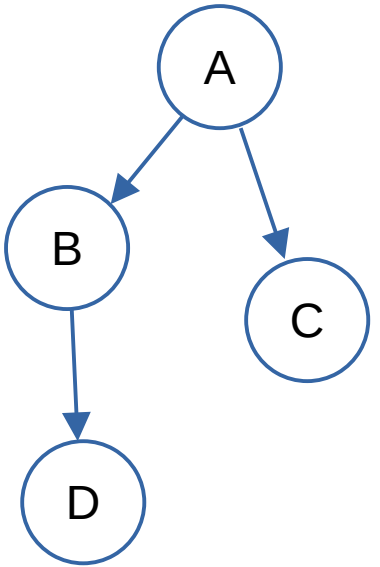
- **BFS** erkundet einen Graphen Ebene für Ebene - ausgehend von einem Root-Node werden zuerst alle Nachbarn besucht, dann deren Nachbarn und so weiter. Hierfür verwendet BFS eine **Queue** (FIFO).
- **DFS** erkundet einen Graphen, indem es so tief wie möglich in einen Pfad eindringt, bevor es wieder zurückgeht. Um die Nodes entlang des Pfades zu verfolgen, wird ein **Stack** (LIFO) verwendet.
- Normalerweise erforscht BFS/DFS auf einem gerichteten Graphen von einem Node zu seinen Nachfolgern („ausgehende Edges“). Es ist jedoch auch eine *rückwärts gerichtete Durchquerung des Graphen möglich* - dies ist bei der Auflösung von Abhängigkeitsbeziehungen nützlich.

BFS/DFS Query Ansatz

- Bestimmen Sie einen Root Node
- Definieren Sie die Suchparameter
- Definieren Sie Stopp-Bedingungen

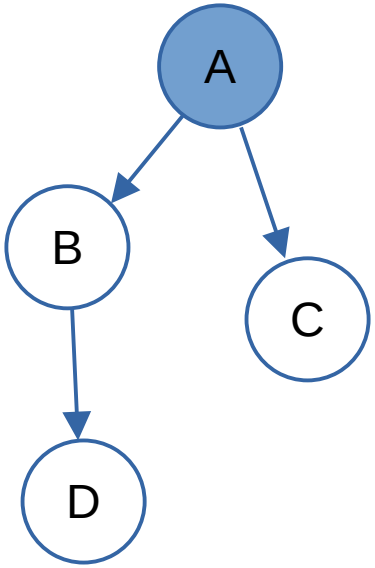
Beispiel – BFS

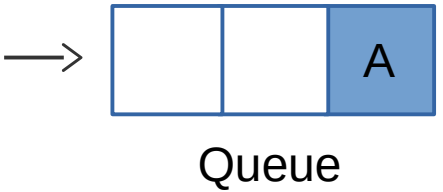
- Root Node: A
- Suchparameter: Anzahl der Nachbarn
- Stopp-Bedingungen: Keine weiteren Nachbarn



BFS – Schritt 1

Der Root-Node A wird in die FIFO-Queue gestellt und blau markiert, während die Suchergebnisse ermittelt werden.

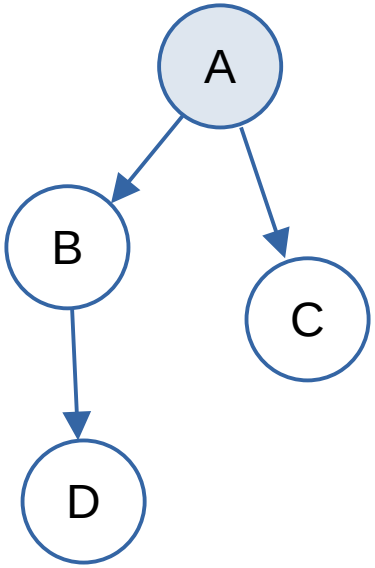


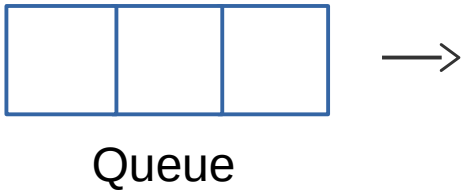


Suchergebnisse:

BFS – Schritt 2

A wird aus der Queue entnommen, seine Suchergebnisse in eine Liste.



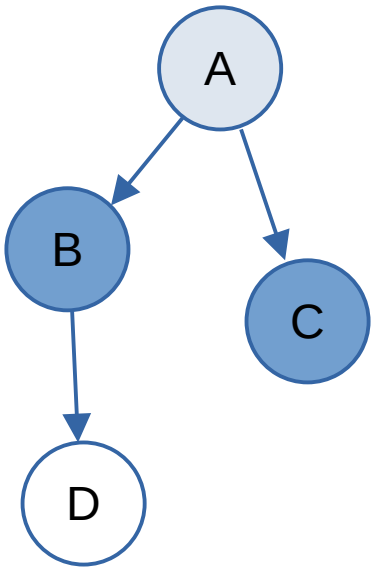


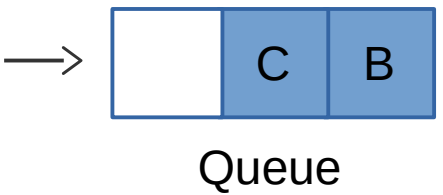
Suchergebnisse:

- **A** hat 2 Nachbarn

BFS – Schritt 3

Die Nachbarn von Node A (B und C) werden in die Queue aufgenommen.



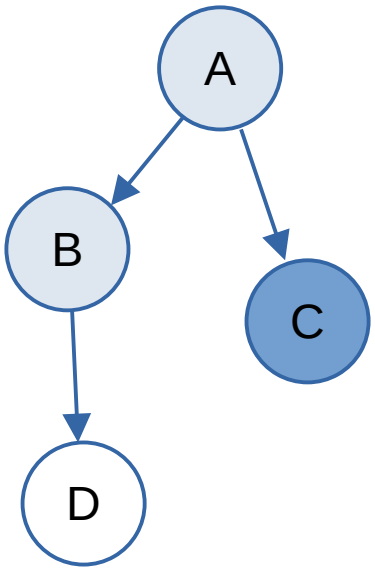


Suchergebnisse:

- **A** hat 2 Nachbarn

BFS – Schritt 4

B wird aus der Queue entnommen, seine Suchergebnisse in eine Liste.



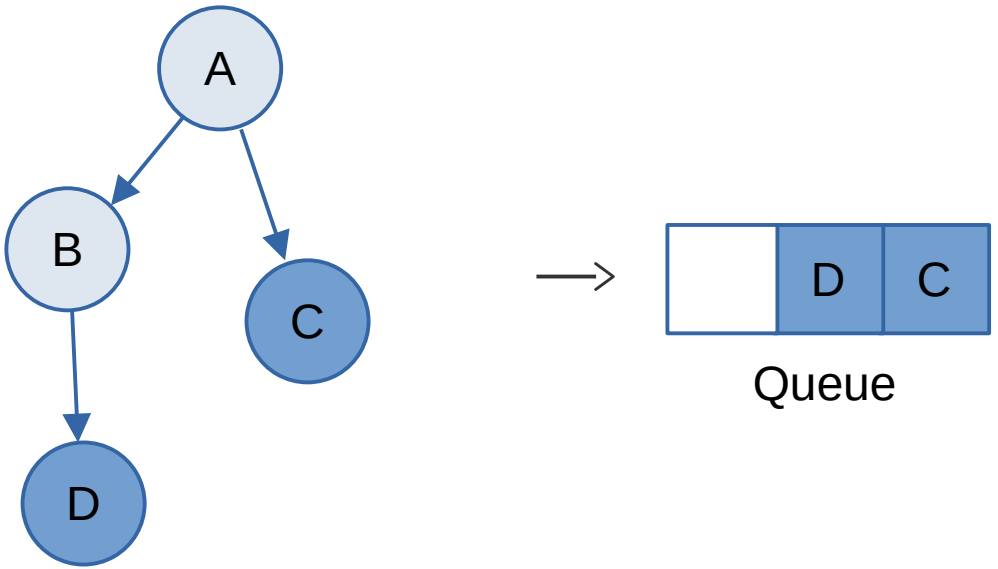
Queue

Suchergebnisse:

- **A** hat 2 Nachbarn
- **B** hat 1 Nachbar

BFS – Schritt 5

Die Nachbarn von Node B werden in die Queue aufgenommen.

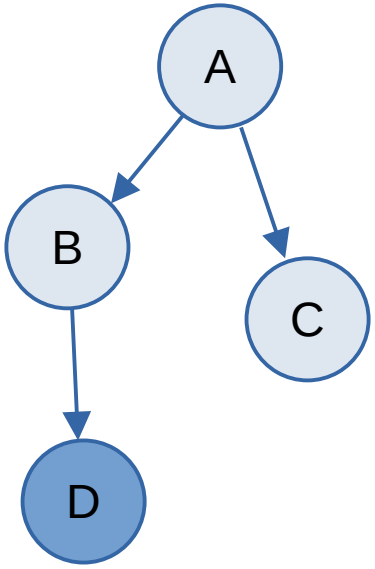


Suchergebnisse:

- **A** hat 2 Nachbarn
- **B** hat 1 Nachbar

BFS – Schritt 6

C wird aus der Queue entnommen, seine Suchergebnisse in eine Liste.



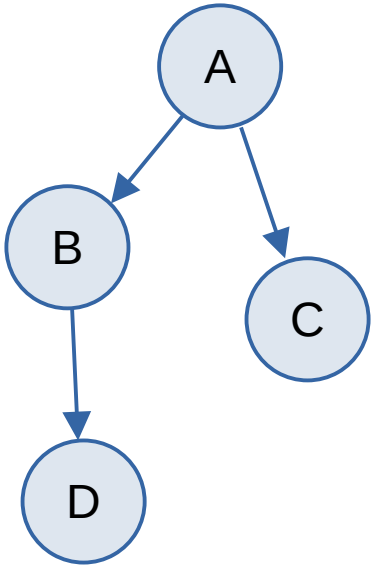
Queue

Suchergebnisse:

- **A** hat 2 Nachbarn
- **B** hat 1 Nachbarn
- **C** hat 0 Nachbarn

BFS – Schritt 6

D wird aus der Queue entnommen, seine Suchergebnisse in eine Liste. Da die Stopp-Bedingung erfüllt ist, beenden wir die Suche.



Queue

Suchergebnisse:

- **A** hat 2 Nachbarn
- **B** hat 1 Nachbarn
- **C** hat 0 Nachbarn
- **D** hat 0 Nachbarn

Beispiel – BFS Abfrage

- Zur Beantwortung der Frage: „Wie viele der Komponenten des autonomen Fahrzeugsteuerungssystems sind Sensoren?“, führen wir ein **BFS mit Rückwärts-Traversal** durch.
- Root Node: **SYSK** (Klasse SystemKomponent)
- Suchparameter:

  - Anzahl Nachbarn

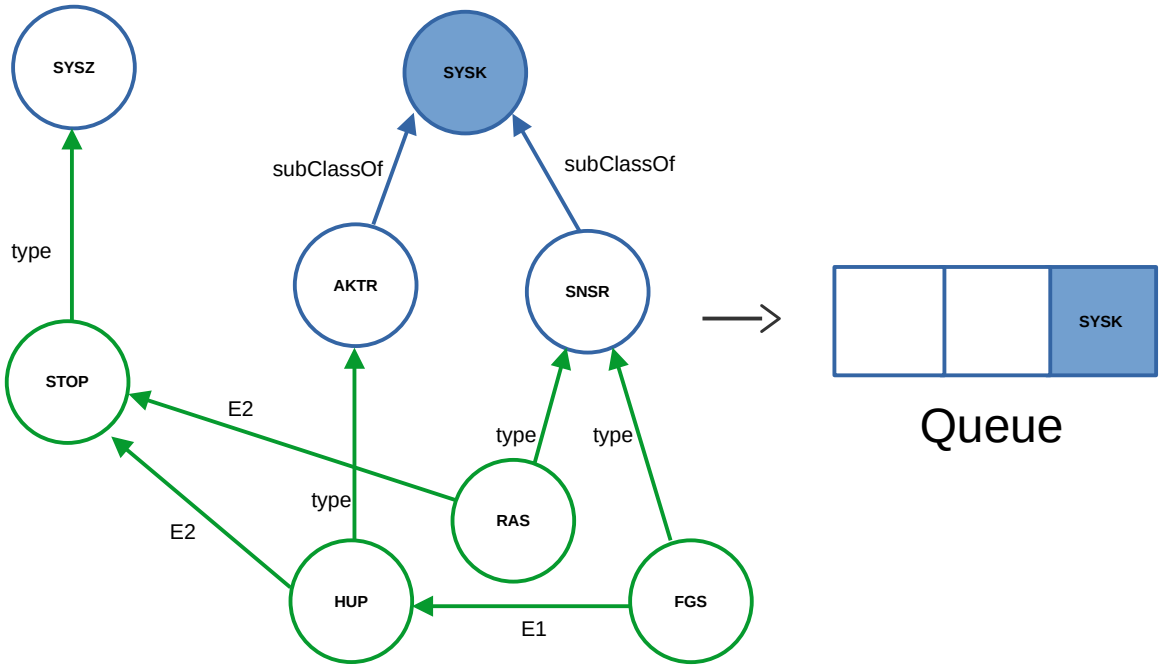
  - Wenn Node keine Sensorklasse ist, übergehen

  - Wenn der Node eine Sensorklasse ist, seine Instanzen zählen
- Stopp-Bedingungen:

  - Alle Instanzen von Sensorklassen wurden gezählt

BFS Abfrage – Schritt 1

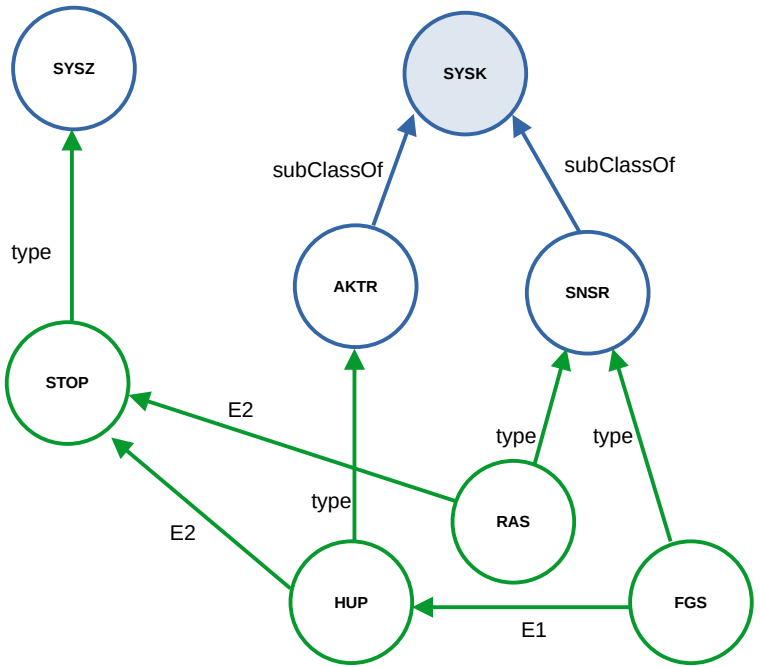
Der Root-Node **SYSK** wird in die FIFO-Queue gestellt und blau markiert, während die Suchergebnisse ermittelt werden.



Suchergebnisse:

BFS Abfrage – Schritt 2

SYSK wird aus der Queue entnommen, seine Suchergebnisse in eine Liste.



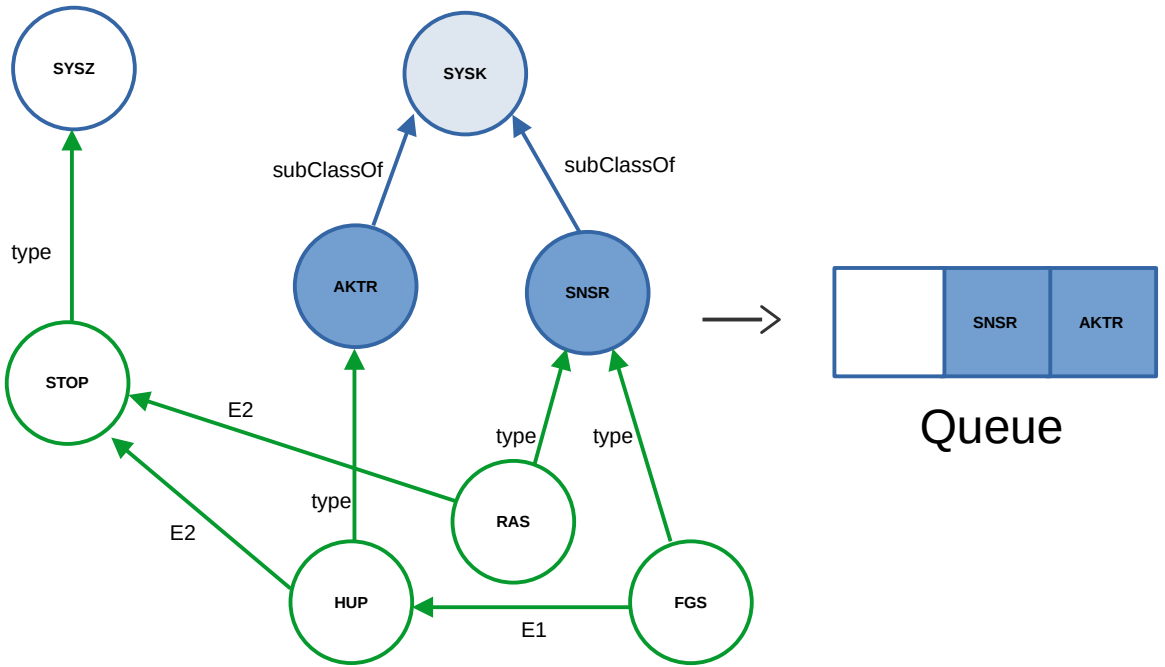
Queue

Suchergebnisse:

- **SYSK** hat 2 Nachbarn

BFS Abfrage – Schritt 3

Die Nachbarn von **SYSK** werden in die Queue aufgenommen.

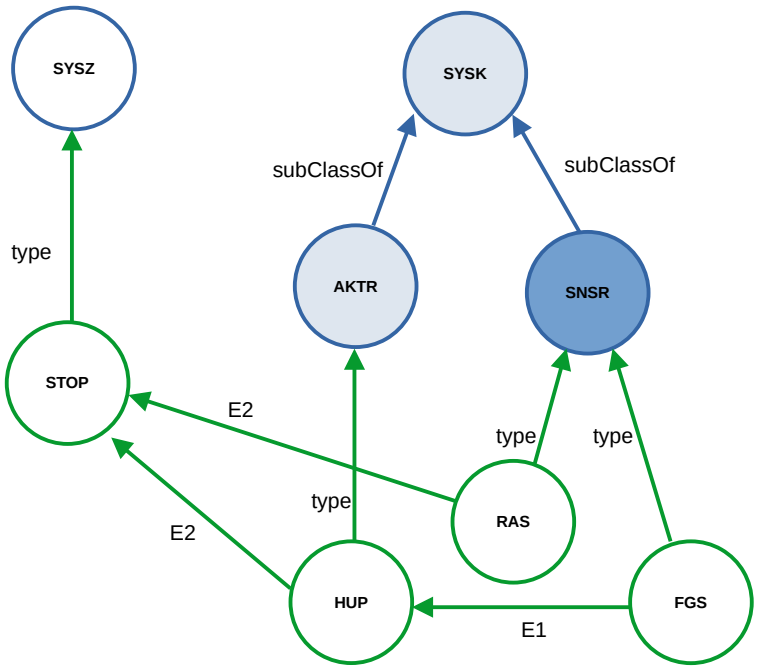


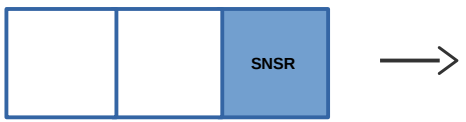
Suchergebnisse:

- **SYSK** hat 2 Nachbarn

BFS Abfrage – Schritt 4

AKTR wird aus der Queue entnommen, seine Suchergebnisse in eine Liste.





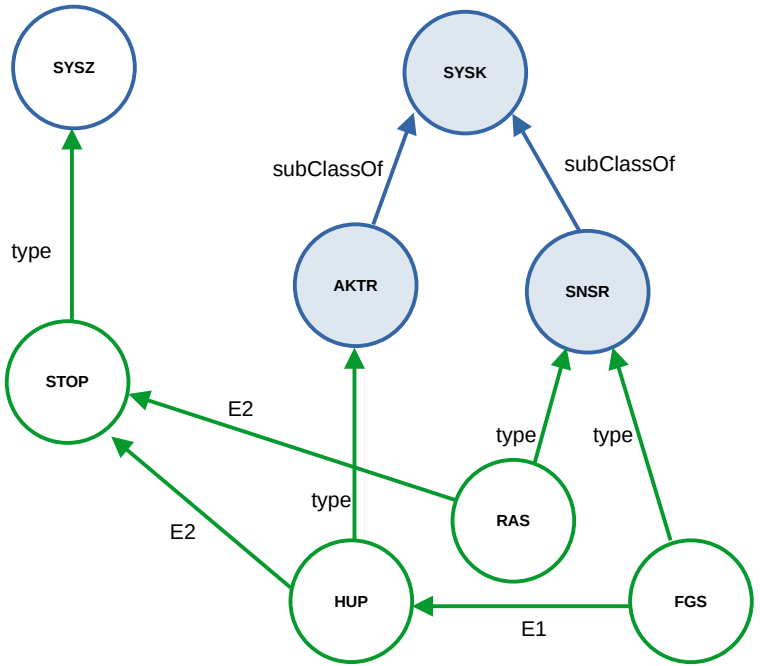
Queue

Suchergebnisse:

- **SYSK** hat 2 Nachbarn
- **AKTR** ist keine Sensor-Klasse

BFS Abfrage – Schritt 5

SNSR wird aus der Queue entnommen, seine Suchergebnisse in eine Liste. Da unsere Suche die Anzahl der Sensoren (Instanzen von Sensor) ergeben hat, brechen wir die Suche ab.



Queue

Suchergebnisse:

- **SYSK** hat 2 Nachbarn
- **AKTR** ist keine Sensor-Klasse
- **SNSR** hat 2 Instanzen

Beispiel – DFS Abfrage

- Zur Beantwortung der Frage: „Führt der Fußgängersensor in einem autonomen Auto dazu, dass das Auto anhält?“, führen wir ein **DFS** durch.
- Root Node: **FSG** (Instanz FussgaengerSensor)
- Suchparameter:

  - Anzahl Nachbarn

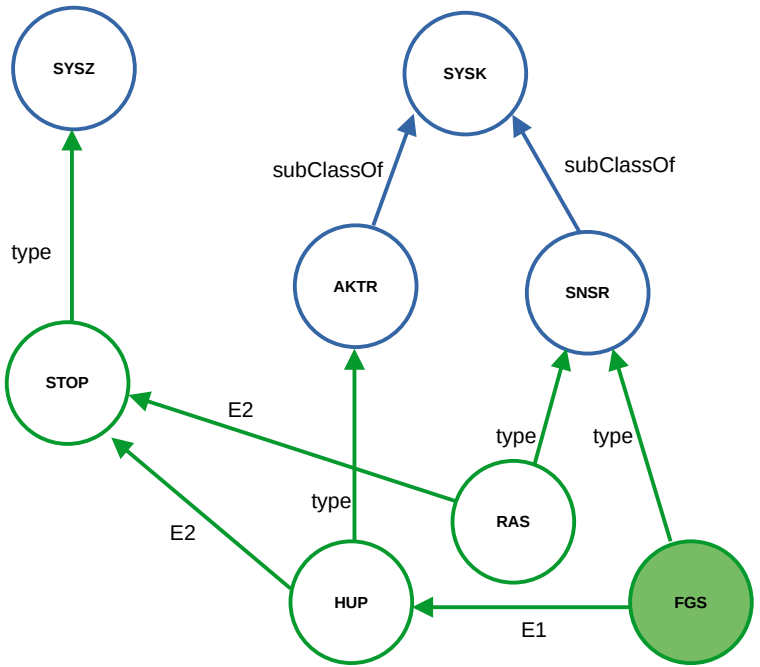
  - Wenn Node keine Instanz ist, übergehen

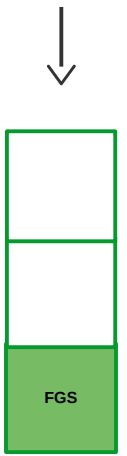
  - Wenn der Node eine Instanz Stop ist, Antwort auf „Ja“ setzen
- Stopp-Bedingungen:

  - Wenn der Node eine Instanz Stop ist

DFS Abfrage – Schritt 1

Der Root-Node **FSG** wird in die LIFO-Stack gestellt und grün markiert, während die Suchergebnisse ermittelt werden.



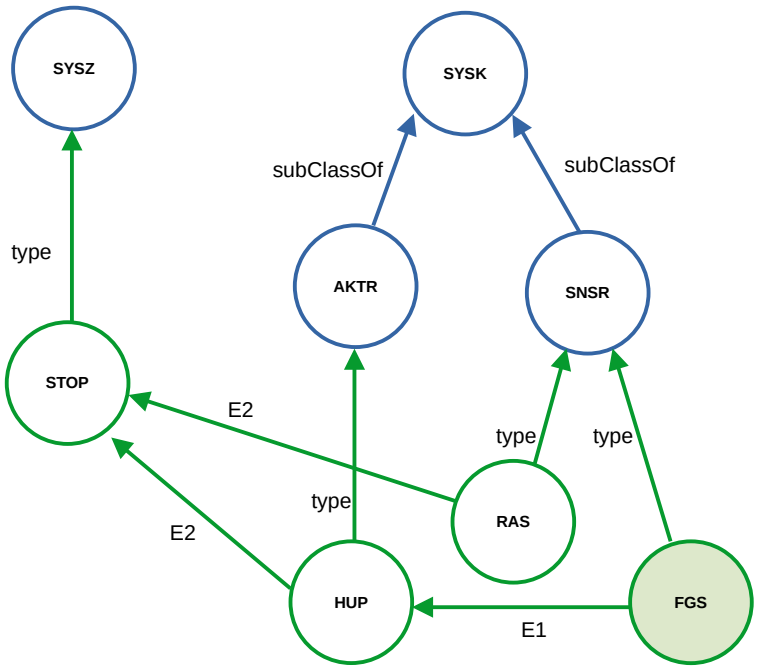


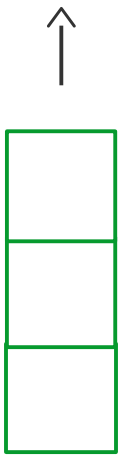
Stack

Suchergebnisse:

DFS Abfrage – Schritt 2

FSG wird aus der Stack entnommen, seine Suchergebnisse in eine Liste.





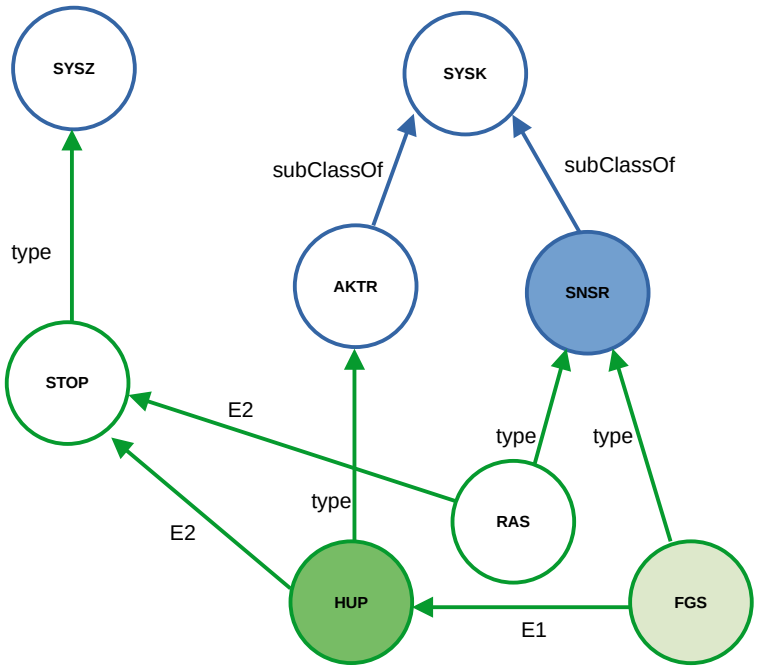
Stack

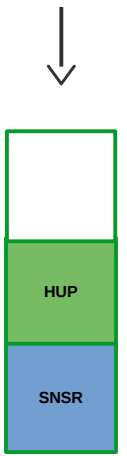
Suchergebnisse:

- **FSG** hat 2 Nachbarn

DFS Abfrage – Schritt 3

Die Nachbarn von **FSG** werden in die Stack aufgenommen.





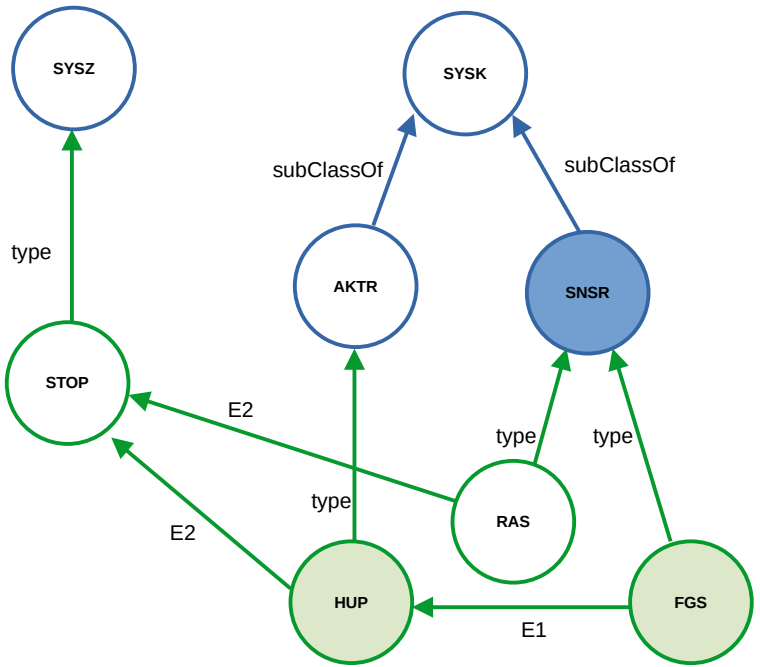
Stack

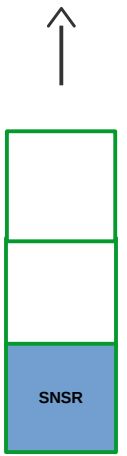
Suchergebnisse:

- **FSG** hat 2 Nachbarn

DFS Abfrage – Schritt 4

HUP wird aus der Stack entnommen, seine Suchergebnisse in eine Liste.





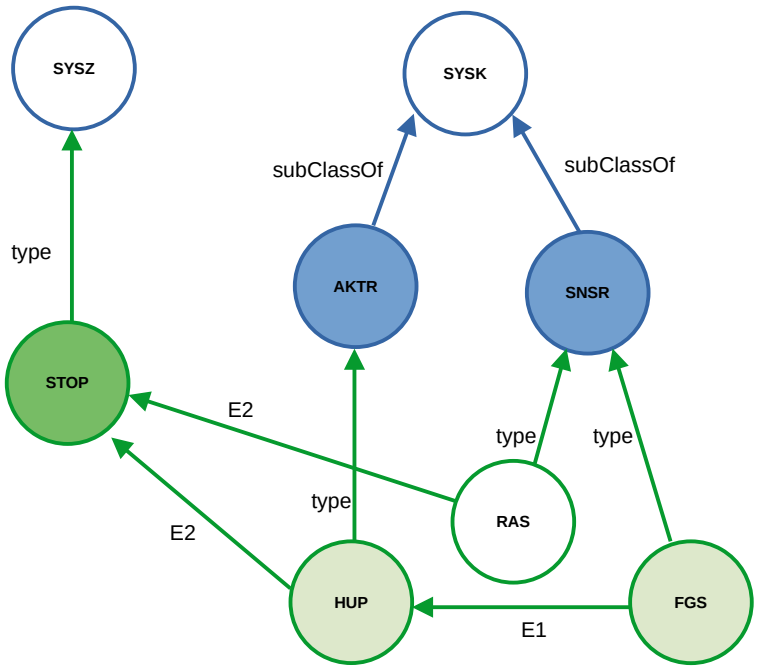
Stack

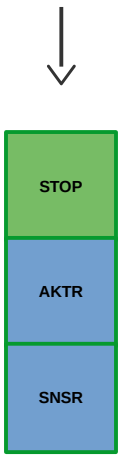
Suchergebnisse:

- **FSG** hat 2 Nachbarn
- **HUP** hat 2 Nachbarn

DFS Abfrage – Schritt 5

Die Nachbarn von **HUP** werden in die Stack aufgenommen.





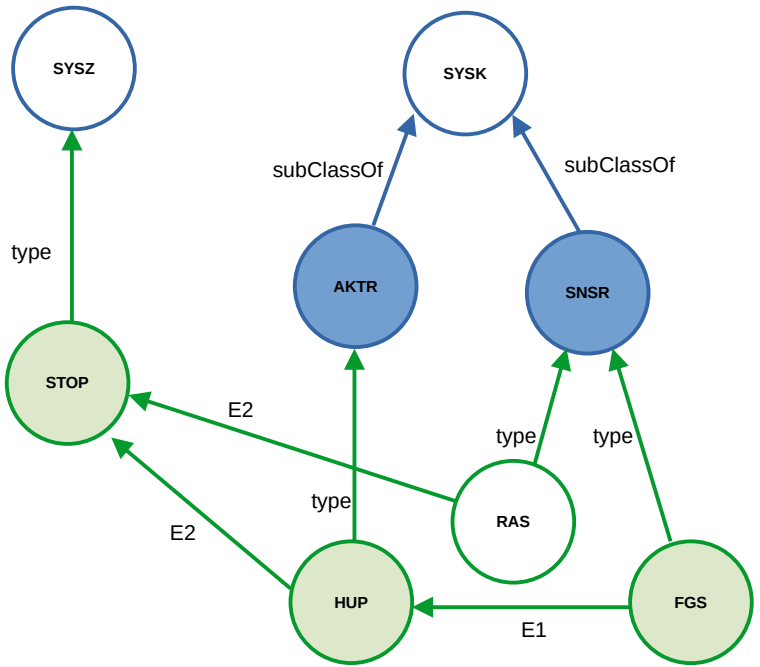
Stack

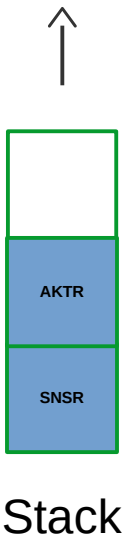
Suchergebnisse:

- **FSG** hat 2 Nachbarn
- **HUP** hat 2 Nachbarn

DFS Abfrage – Schritt 6

- **STOP** wird aus der Stack entnommen, seine Suchergebnisse in eine Liste. Da die Instanz von SystemZustand 'Stop' gefunden wird, beenden wir die Suche.





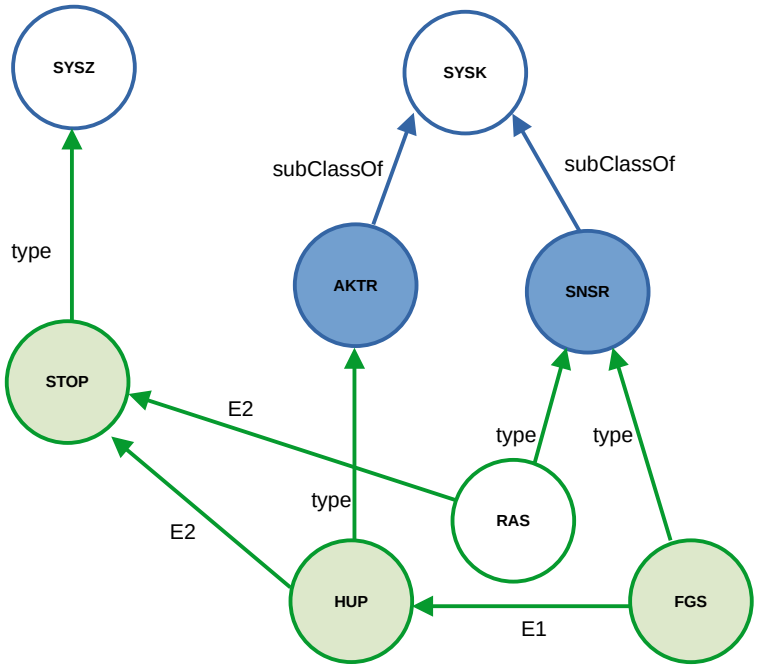
Stack

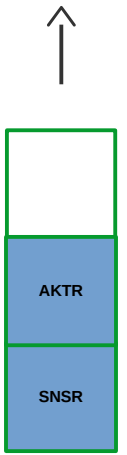
Suchergebnisse:

- **FSG** hat 2 Nachbarn
- **HUP** hat 2 Nachbarn
- **STOP** Instanz gefunden

DFS Abfrage – Ergebnisse

In den Suchergebnissen sehen wir, dass der **STOP** SystemZustand vom **FSG** Sensor über den **HUP** Aktor erreicht wird – Antwort: “JA”. Man beachte, dass die anderen Nodes im Stack ohnehin übersprungen worden wären, wenn die Stopp-Bedingung nicht erfüllt gewesen wäre, da sie keine Instanz-Nodes sind.





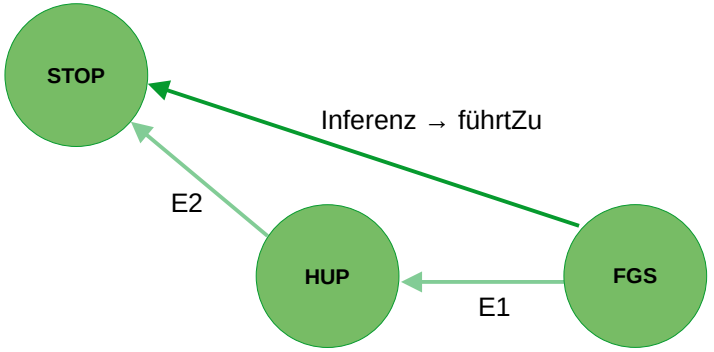
Stack

Suchergebnisse:

- **FSG** hat 2 Nachbarn
- **HUP** hat 2 Nachbarn
- **STOP** Instanz gefunden

KG – Inferenz und Pfadlänge

Nach der DFS könnten wir die extrahierte Tatsache verwenden, um ein Axiom für die Inferenz „führtZu“ zu erstellen. Die Inferenz in einem KG erzeugt neues Wissen als gerichtete Edges im Graph. Diese Edges können die Wege zwischen zwei Nodes verkürzen. In der gezeigten Eigenschaftskette-Inferenz wird der Pfad zwischen **FGS** und **STOP** *um 1 Hop verkürzt*. Je höher die Anzahl der Inferenzen, desto kürzer ist die durchschnittliche Pfadlänge zwischen den Nodes im KG.



Graphen – APL und Effizienz

Average Path Length (**APL**): Der Durchschnitt der kürzesten Wege zwischen allen Node-Pairs i und j, für n Nodes. Für gerichtete Graphen ist dies definiert als:

$$APL = \frac{1}{n(n-1)} \sum_{\substack{i,j \in V \\ i \neq j}} d(i,j) \; .$$

Global Effizienz (**E_{glob}**): Durchschnitt der Umkehrung der kürzesten Wege zwischen allen Node-Pairs. Für gerichtete Graphen ist dies definiert als:

$$E_{\mathrm{glob}} = \frac{1}{n(n-1)} \sum_{\substack{i,j \in V \\ i \neq j}} \frac{1}{d(i,j)} \; .$$

KG – Abfrage Effizienz

Da der Gesamtwirkungsgrad umgekehrt proportional zur APL ist, können wir die folgende Näherung verwenden:

$$E_{\mathrm{glob}} \sim \frac{1}{APL} \, .$$

Wichtig: *Dies gilt jedoch nur für Abfragen*. Wir müssen bedenken, dass die Speichereffizienz sinkt, denn je mehr Axiome und Inferenzen in einer KG enthalten sind, desto größer ist der Triple-Store oder die Datenbank.