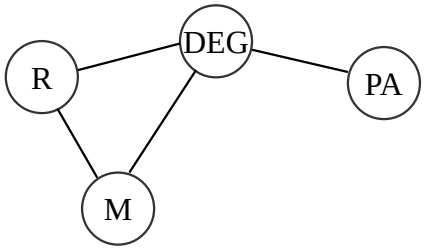
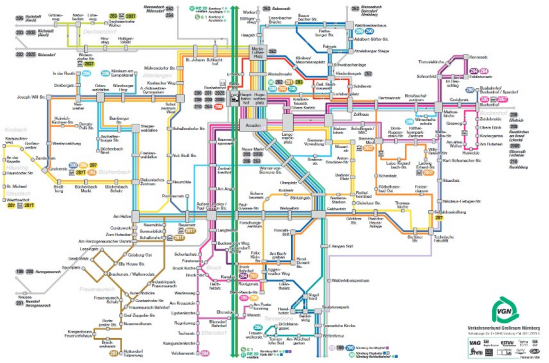
Datenstruktur GRAPH

Begriffsdefinition



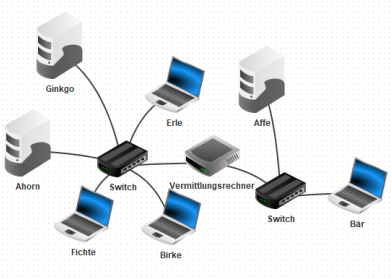
Ein ***Graph*** ist eine nicht-hierarchische Datenstruktur. Man kann sich einen Graph vorstellen, wie ein Spinnennetz. Die Fäden nennen wir ***Kanten*** des Graphen. Die Orte, an denen die Fäden beginnen oder enden nennen wir ***Knoten***. Dabei muss nicht jeder Knoten mit jedem anderen durch eine Kante direkt verbunden sein.



Anwendungsgebiete

Graphen finden immer dann Verwendung, wenn man eine Menge von Objekten miteinander „vernetzen“ möchte:

- Liniennetz Erlangen (Busnetz),
- Skipistenplan,
- Rechnernetze,
- Beziehungen in sozialen Netzwerken



Aufgabe 1:

a) Bestimme, was die Kanten und Knoten in den obigen vier Beispielen darstellen.

b) Überlege dir drei weitere Beispiele für Graphen und bestimme dort die Kanten und Knoten.

Eigenschaften von Graphen

1. Eine ***gerichtete Kante*** darf nur in eine Richtung betrachtet werden.

   Beispiele: *Einbahnstraßen im Verkehr,*
   Dioden in der Elektronik,
   „Follower-Beziehung“

2. Eine ***gewichtete Kante*** ist mit einem oder mehreren Werten im Sachzusammenhang versehen.

   Beispiele: *Länge einer Strecke, Dauer einer Fahrt, Kosten einer Fahrt*
   Maximal zulässige Stromstärke bei elektrischen Leiterbahnen
   Art einer sozialen Beziehung (Bekannter, Freund, Familie, Kollege, ...)

3. Einen ***Weg*** in einem Graphen gibt man als Folge der dabei durchlaufenen Knoten an. Endet ein Weg an demselben Knoten, an dem er begonnen hat, so spricht man von einem ***Rundweg (Zyklus)***. Besitzt ein Graph mindestens einen Zyklus, so spricht man von einem ***zyklischen Graphen***.

4. Kann man von jedem Knoten aus jeden anderen Knoten über einen Weg erreichen, so nennt man diesen ***ungerichteten Graph*** einen ***zusammenhängenden Graphen***.

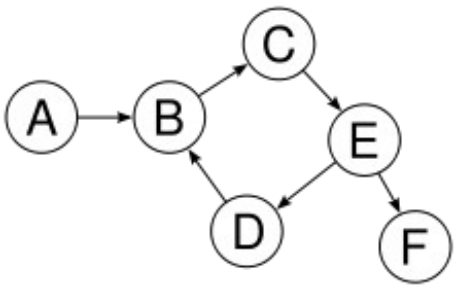
   4.1. Falls man bei einem ***gerichteten Graphen*** von jedem Knoten zu jedem anderen Knoten über einen Weg gelangen kann, ist dies ein ***stark zusammenhängender*** Graph.

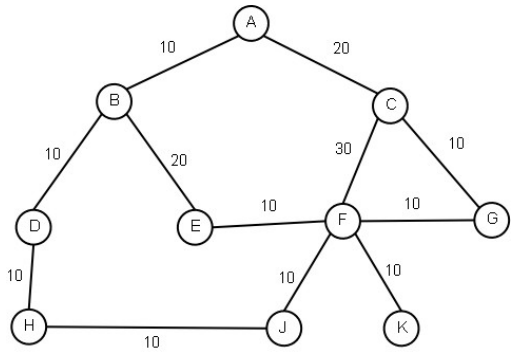
   4.2. Man nennt einen ***gerichteten Graph schwach zusammenhängend***, wenn dieser Graph nach vorheriger Definition nicht zusammenhängend ist, aber der dazugehörige ungerichtete Graph es wäre (= also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt).

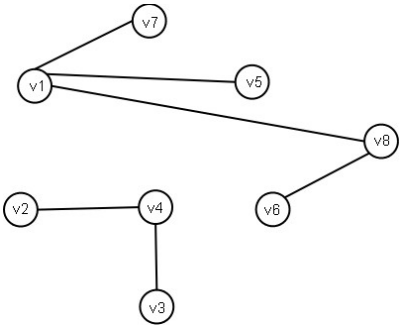
Aufgabe 2:

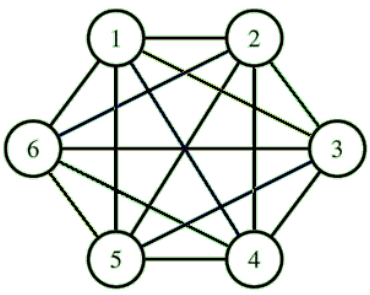
a) Gib für deine drei eigenen Beispiele aus der Aufgabe **1b)** je eine Möglichkeit für eine gerichtete und gewichtete Kante an.

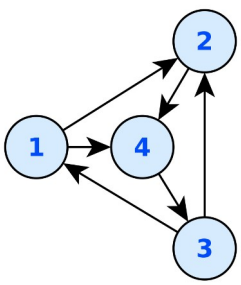
b) Betrachte die folgenden fünf Graphen und untersuche jeden darauf in welcher Form die vier Eigenschaften „(un-)gerichtet“, „(un-)gewichtet“, „(nicht) zyklisch“, „(nicht/stark/schwach) zusammenhängend“ auf ihn zutrifft.





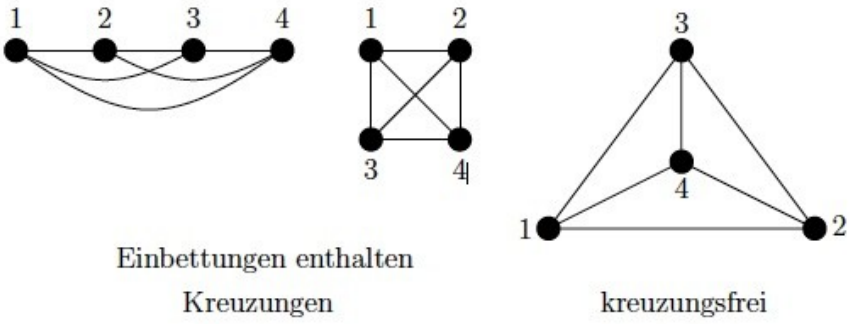






Bemerkung:

Man sieht zwei Graphen selten an, ob sie identisch sind, denn man kann einen Graphen zeichnerisch sehr unterschiedlich anordnen. Folgende drei Graphen stellen denselben Sachverhalt dar und gehen nur durch räumlich unterschiedliche Anordnung auseinander hervor.



Einbettungen enthalten Kreuzungen

kreuzungsfrei

Informatische Repräsentation eines Graphen

Bei einer Datenstruktur Graph müssen die Bestandteile eines Graphen, also die Kanten und Knoten, informatisch umgesetzt werden.

Knoten im Graph

Knoten werden als eigene Klasse implementiert, die eine Referenz auf ein Inhaltsobjekt besitzt. Zum Beispiel hat der Knoten eine Referenz auf eine Stadt oder eine Haltestelle.

Einschub: Warum referenziert der Graph nicht gleich Städte oder Haltestellen?

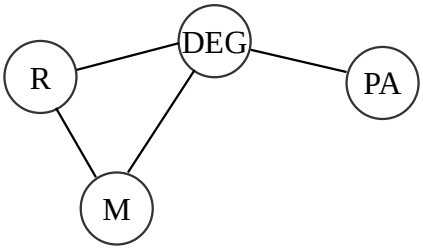
Sinnvoll entworfene Software folgt dem Prinzip der Trennung von Struktur (Knoten) und Inhalt (Stadt, Haltestelle,…). Das bedeutet, dass die Attribute und Methoden, die man benötigt um die Struktur Graph/Knoten zu implementieren mit der Klasse Stadt nichts zu tun hat. Umgekehrt könnte man durch Schritte der Verallgemeinerung des Knoten durch „allgemeinere Referenzen“ und/oder generische Datentypen den Knoten von der Inhaltsklasse entkoppeln und somit einen allgemein nutzbaren Graphen entwickeln. → Daher ist es sinnvoll, den Knoten als eigene Klasse einzuführen.

Eine Möglichkeit ist es, die Kanten als Attribut in der Klasse Graph zu repräsentieren. Hierfür wird dann eine Matrix genutzt.

Kanten im Graph: Adjazenzmatrix

Bei dieser Variante wird die maximale Anzahl von Knoten bereits im Konstruktor festgelegt, weil die Verwaltung hier über ein Array erfolgt. Es wird auch Speicherplatz für

alle möglichen Kanten (vollständiger Graph) reserviert. Die Knoten werden in einem Array gespeichert und die Kanten als ***Matrix*** (zweidimensionales Array) des Typs **int** (gewichtete Kanten).



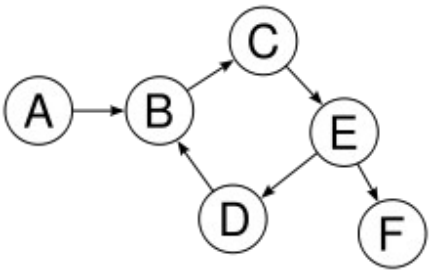
	M	R	PA	DEG
M	0	1	-1	1
R	1	0	-1	1
PA	-1	-1	0	1
DEG	1	1	1	0

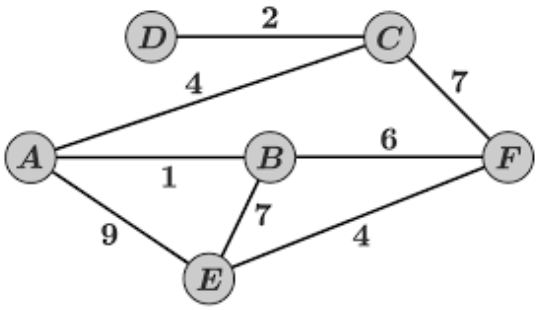
Da in der Matrix sowohl ein Feld für z.B. M→PA als auch ein Feld für PA→M zur Verfügung steht, kann man sehr leicht auch gerichtete Kanten realisieren.

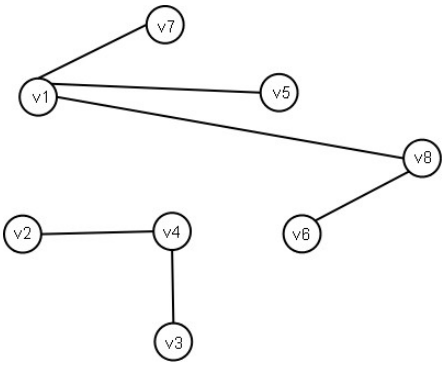
Hinweis: In der Regel gibt es keine Kanten von einem Knoten zu sich selbst, weswegen dieses Feld in der Matrix mit 0 belegt wird, es entsteht die sog. ***Nulldiagonale***. Existiert keine Kante wird das in einer Integer-Matrix durch den Wert -1 gekennzeichnet.

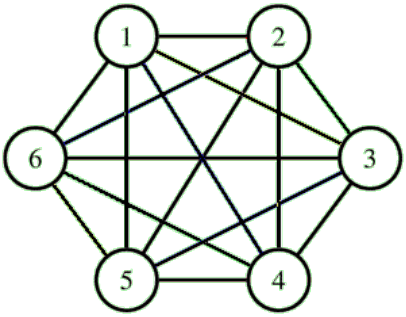
Aufgabe 3:

a) Gib für jeden der folgenden Graphen die Adjazenzmatrix an.









b) Welche der vier Eigenschaften von Graphen lassen sich anhand der Adjazenzmatrix leicht bestimmen?

Implementierung einer Matrix

Zweidimensionale Arrays werden in Java durch zwei Paar eckige Klammern realisiert:

```
int[][] kanten;

kanten = new int[25][25];
```

Dabei steht das erste Klammernpaar für r(ow) (Länge der Zeile, sprich: waagrecht) und das zweite für c(olumn) (Länge der Spalte, sprich: senkrecht).

Ein **Spaltendurchlauf** wird wie folgt implementiert:

```
for (int r = 0 ; r < knoten.length ; r++) {… kanten[r][c] …}
```

Ein **Zeilendurchlauf** analog mit:

```
for (int c = 0 ; c < knoten.length ; c++) {… kanten[r][c] …}
```

Für die **gesamte Matrix** werden zwei Schleifen geschachtelt:

```
for (int r = 0 ; r < knoten.length ; r++) {
     for (int c = 0 ; c < knoten.length ; c++) {
          … kanten[r][c] …
     }
}
```

Aufgabe 4:

Beschreibe genau die Reihenfolge der Zellen bei obiger Doppelschleife.

Es ergibt sich dadurch folgende Feldernummierung:

[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]
[3][0]	[3][1]	[3][2]	[3][3]

Zuordnung über den Index

Die Datenstruktur Graph wird durch zwei Felder realisiert:

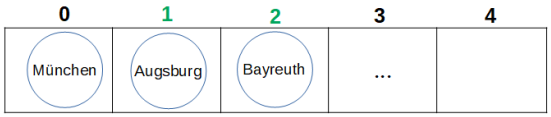
- ***knoten***: speichert Knoten mit z. B. Städten
- ***kanten***: speichert nur die Information, ob eine Kante (mit Gewicht) existiert

→ Die einzige Verbindung zwischen den Feldern besteht lediglich über die Indizies.

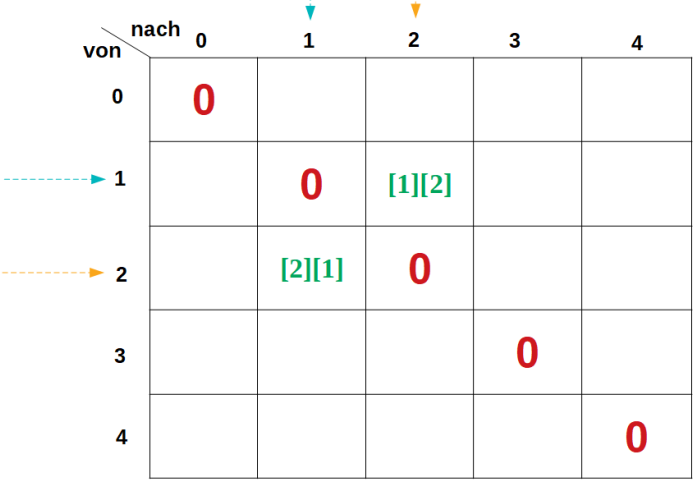
Beispiel:

Soll eine neue (ungerichtete) Kante von Augsburg nach Bayreuth eingefügt werden, müssen zunächst die beiden Indizies 1 und 2 des Knoten-Arrays herausgefunden werden. Über diese Indizies wird die richtige Position für die Kante in der Kanten-Matrix bestimmt: [1][2] und [2][1].

Knoten-Array:



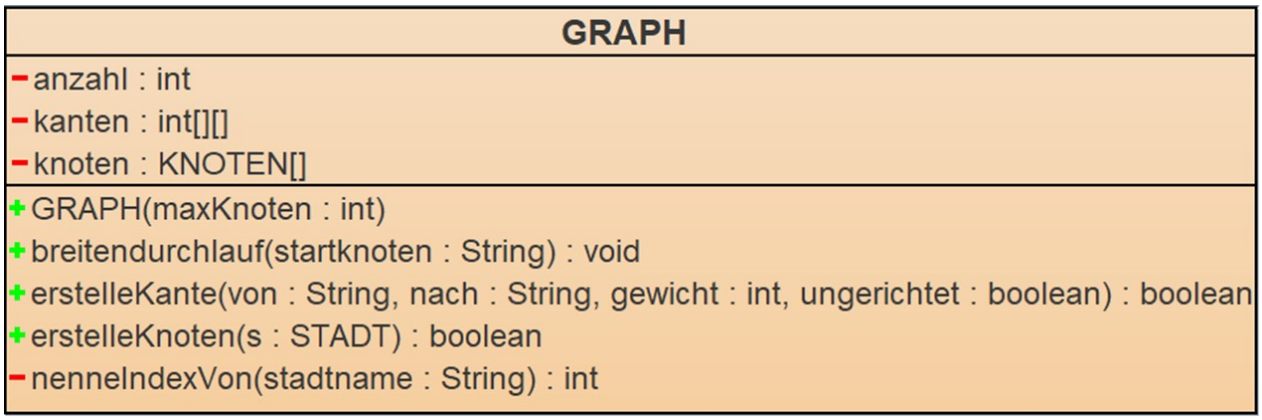
Kanten-Matrix:



Implementierung des Graphen

Aufgabe 5:

a) Öffne das Projekt Graph:



Programmierreihenfolge:

- Attribute des Graphen deklarieren und im Konstruktor initialisieren (Wie sieht die Adjazenzmatrix bei einem leeren Graph aus?)

- Methode **`erstelleKnoten(…)`** gibt **`true`** zurück, falls noch ein Knoten in das Knoten-Array eingefügt werden kann; ansonsten **`false`**.

- Methode **`nenneIndexVon(…)`** sucht im Knoten-Array nach dem Knoten mit dem passenden Stadtnamen.
Stringvergleich: `s1.compareToIgnoreCase(s2) == 0`, falls die Strings identisch sind

- Methode **`erstelleKante(…)`** sucht sich die Indizies von den Städten `von` und `nach` heraus und falls diese Indizies nicht -1 und nicht identisch sind, dann wird die Kante (bei ungerichtet in beide Richtungen) in die Matrix eingefügt.

Graphendurchläufe

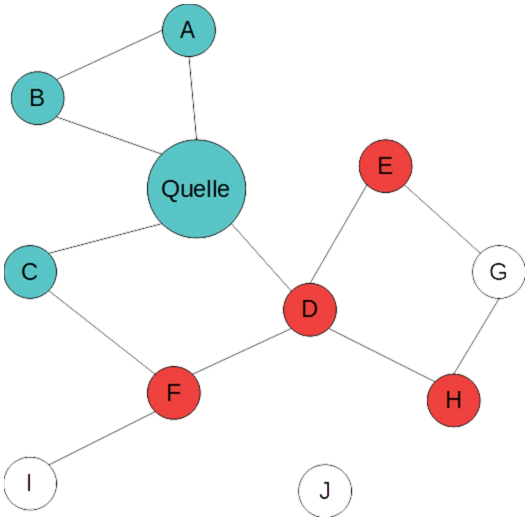
Es gibt viele Anlässe, die es nötig machen, einen Graphen systematisch ganz oder teilweise zu durchlaufen. So sucht z.B. ein Handlungsreisender eine Route, bei der er alle Städte einmal anfahren kann und dabei seine zu fahrende Strecke minimiert. Eine Kehrmaschine versucht, alle Wege in jede Richtung genau einmal abzufahren, ohne dabei unnötig viel Leerfahrten zu haben. Ein Navigations-System versucht, den kürzesten Weg von A nach B zu finden.

Breitendurchlauf

Die Idee des ***Breitendurchlaufs*** alle Nachbarknoten besucht, die von dem Ausgangsknoten direkt erreichbar sind, bevor man in die „Tiefe“ des Graphen vordringt.

→ **Präsentation Breitendurchlauf**

- **unbesucht**: Ausgangszustand
- **besucht**: Gefundene Knoten, die noch nicht Ausgangsknoten waren und damit noch nicht alle Nachbarknoten vollständig durchsucht wurden
- **fertig**: Knoten war mal Ausgangsknoten und somit sind alle Nachbarknoten besucht gesetzt



Breitendurchlaufalgorithmus (händischer Durchlauf)

Initialisierung:

- Der Startknoten ist besucht und erster Ausgangsknoten.

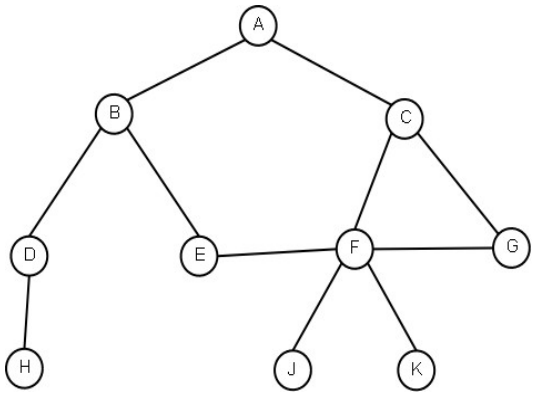
Algorithmus:

- Wiederhole für jeden gefundenen Knoten die folgenden Schritte:
  - Wähle den nächsten Ausgangsknoten.
  - Besuche in alphabetischer Reihenfolge die noch unbesuchten Nachbarknoten des Ausgangsknotens und merke dir die Reihenfolge.
  - Sobald es keine unbesuchten Nachbarknoten mehr gibt, ist der Ausgangsknoten fertig abgeschlossen.

Aufgabe 6:

Gib den Breitendurchlauf für den abgebildeten Graphen an, indem du nacheinander die besuchten Knoten notierst.

Starte bei Knoten E.



Implementierung des Breitendurchlaufs

Vorarbeit in der Klasse Knoten:

- Die Knoten müssen ein Attribut besucht mit Getter/Setter-Methode (setzeBesucht(…), nenneBesucht()) erhalten.

Vorbereitung in der Klasse Graph → Methode breitendurchlauf(String startknoten):

- Existiert der angegebene Startknoten im Graphen? Falls nicht -> Methode abbrechen!
  Tipp: man kann auch in Methoden mit Rückgabetyp void ein **return;** nutzen, um die Methode zu beenden. Dabei gibt man aber keinen Rückgabewert an.
- Alle Knoten müssen auf unbesucht gesetzt werden.
- Schlange (mit passender Größe) erstellen
- Startknoten in die Schlange einreihen und auf besucht setzen

Algorithmus in der Methode breitendurchlauf(String startknoten) nach der Vorbereitung:

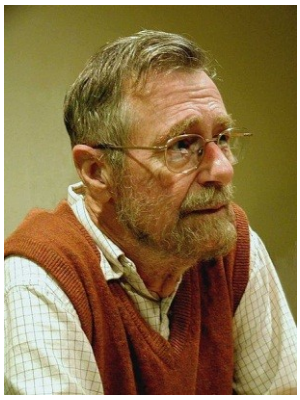
- Wiederhole solange bis die Schlange leer ist:
  - Nimm den vordersten Knoten aus der Schlange, bestimme den Index und gib per System.out.println() aus, welche Stadt der neue Ausgangsknoten ist.
  - Iteriere über das Kanten-Array in der passenden Zeile und prüfe, ob es unbesuchte Nachbarknoten gibt. Falls ja:
    - Setze diesen auf besucht
    - Gib per System.out.println() aus, dass dieser Knoten jetzt besucht ist
    - Füge diesen Knoten hinten in der Schlange ein
  - Gib per System.out.println() aus, dass der aktuelle Ausgangsknoten fertig ist.

Dijkstra-Algorithmus / Kürzeste Wege in Graphen

→ **Präsentation Dijkstra-Algorithmus**

Der Dijkstra-Algorithmus ist ein Breitendurchlauf, der die Knoten aber nach der kleinsten Gesamtentfernung auswählt (gieriger Algorithmus) und nicht in der gefundenen Reihenfolge.

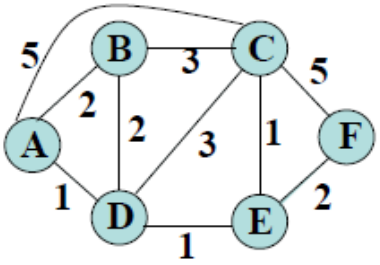
Algorithmus:



- Der Startknoten hat die Distanz 0 und ist der erste aktuelle Knoten k:
- Untersuche alle unfertigen Nachbarknoten n von k:
  - Berechne die Gesamtdistanz von allen nicht fertigen Nachbarknoten n über Knoten k.
  - Sollte n so durch eine kleinere Distanz als bisher erreicht werden können, notiere die Distanz und den Vorgängerknoten. Bei einer größeren Gesamtdistanz bleibt die vorher kleinere Distanz bestehen.
  - Setze Knoten k fertig/erreicht und wähle unter allen noch nicht fertigen Knoten den mit dem geringsten Gesamtaufwand und verwende ihn als neuen aktuellen Knoten k.

Aufgabe 7:

Führe den Dijkstra-Algorithmus ausgehend vom Knoten A durch.



Ausgangs-knoten	A	B	C	D	E	F

Vertiefungsbereich zu Graphen

Optionale Implementierungen für ganz Schnelle

Breitensuche

Die Breitensuche ist eine Anwendung des Breitendurchlaufalgorithmus. Es soll überprüft werden, ob von einem Startknoten ein bestimmter Zielknoten erreicht werden kann. Somit ist die Breitensuche vom Algorithmus her identisch zum Breitendurchlauf

Kopiere die Methode breitendurchlauf(…) und ändere folgende Punkte:

- Es wird zum Startknoten auch ein Zielknoten als Parameter übergeben und der Index berechnet werden. Falls einer der beiden Knoten nicht existiert, soll die Methode abgebrochen werden.
- Füge am Ende der bedingten Anweisung, ob einer unbesuchter Nachbarknoten existiert folgendes ein:
  Falls die Knotennummer des unbesuchten Nachbarknotens gleich der Zielnummer ist, gib per System.out.println() aus, dass der Zielknoten gefunden wurde und beende mit return; die Methode.

Route berechnen

Durch die Methode Breitensuche wird überprüft, ob ein Zielknoten vom Startknoten erreichbar ist. Die nächste Frage ist logischerweise welche Route man nehmen muss, um vom Startknoten zum Zielknoten zu gelangen.

Hierfür benötigt die Klasse Knoten ein Attribut **vorgänger** (am Besten mit dem Datentyp **int**, der den Index des Vorgängerknoten speichert) mit Getter/Setter-Methode (setzeBesucht(…), nenneBesucht()).

Füge folgendes in der Methode breitensuche(…) ein:

- Wenn alle Knoten auf unbesucht gesetzt werden, sollen auch alle Vorgänger auf -1 gesetzt werden → -1 bedeutet kein Vorgänger, denn -1 ist ein ungültiger Index
- Setze immer, wenn ein unbesuchter Nachbarknoten gefunden wurde, den Vorgänger auf den Index des Ausgangsknotens.
- Nachdem der Zielknoten gefunden wurde, aber bevor man die Methode mit return; abbricht, rufe die Methode nenneRoute(String Zielknoten) auf, die du noch implementieren musst.

Methode nenneRoute(String zielknoten): (ohne konkrete Anleitung)

Bastel dir die Route rückwärts vom Zielknoten zum Startknoten zusammen und gib sie dann in passender Reihenfolge aus.

Tipps: Den Startknoten erkennt man daran, dass es den Vorgängerwert -1 hat und du benötigst eine Datenstruktur, in welcher man die gefundenen Vorgänger zwischenspeichert, um sie umgekehrt auszugeben.

Implementierung des Dijkstra-Algorithmus und Route ausgeben

Beim Dijkstra-Algorithmus bietet es sich an eine Breitensuche zu einer Dijkstra-Suche umzubauen. Kopiere daher die Methode breitensuche(…) und benenne sie um.

Die Klasse Knoten benötigt zwei neue Attribute **fertig** (boolean) und **entfernung** (int) mit jeweils Getter/Setter! Fertig bedeutet, dass ein Knoten Ausgangsknoten war.

Das Attribut besucht wird beim Dijkstra nicht mehr gebraucht, dafür wird ein Knoten auf fertig gesetzt, wenn er (wie auch bei dem Breitendurchlauf fertig ist). Anschließend soll seine Entfernung mit seinem Stadtnamen ausgegeben werden. Zudem wird jetzt auch auf unfertige (und nicht mehr auf unbesuchte) Nachbarknoten geprüft.

Methode dijkstra(String startknoten, String zielknoten): (ohne konkrete Anleitung)

Passe die Methode an den entsprechenden Stellen an, sodass der Dijkstra-Algorithmus funktioniert.

Tipps: Um den Knoten mit der kleinsten Entfernung in der Schlange zu finden und zu entfernen, kann es nötig sein bei der Schlange weitere Methoden zu implementieren!

Alternative: Man nutzt die ArrayList<E> von Java, und lässt sich die Datenstruktur von Java mittels Collections.sort(…) sortieren und entnimmt wieder den ersten Knoten. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html