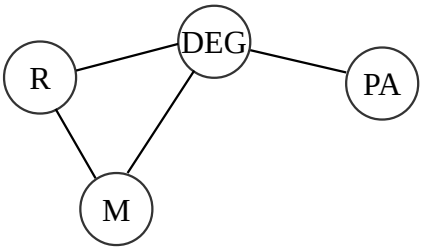
Datenstruktur GRAPH

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:

- Verkehrsnetze,
- Skipistenplan,
- Rechnernetze,
- Beziehungen in sozialen Netzwerken

Ergänzende Fachbegriffe

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

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

- 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, ...)

- 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***.

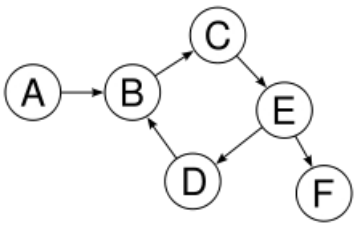
- Kann man von jedem Knoten aus jeden anderen Knoten über einen Weg erreichen, so nennt man diesen Graph einen ***zusammenhängenden Graphen***.
  Vorsicht! *Auch bei gerichteten Graphen muss jeder Knoten von jedem anderen aus erreichbar sein!*

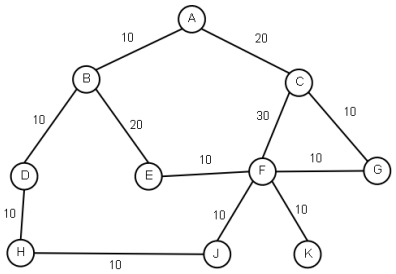
- Ein ***vollständiger Graph*** hat von jedem Knoten zu jedem anderen Knoten eine direkte Kante. Ist ein Graph fast vollständig, so bezeichnet man ihn als ***dichten Graph***. Ist ein Graph weit davon entfernt, vollständig zu sein, so bezeichnet man ihn als ***dünnen Graph***.

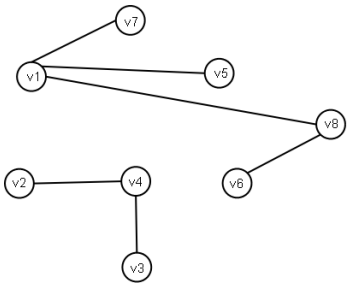
Aufgabe 1:

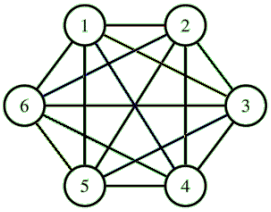
a) *Betrachte die vier Anwendungsbeispiele auf der letzten Seite.*
Welche Objekte stellen bei jedem Beispiel die Knoten und welche die Kanten dar?

b) *Betrachte die folgenden vier Graphen und untersuche jeden darauf, ob einer oder auch mehrere der Begriffe „(un-)gerichtet“, „(un-)gewichtet“, „(nicht) zyklisch“, „(nicht) zusammenhängend“, „dicht“, „dünn“, „vollständig“ auf ihn zutrifft.*



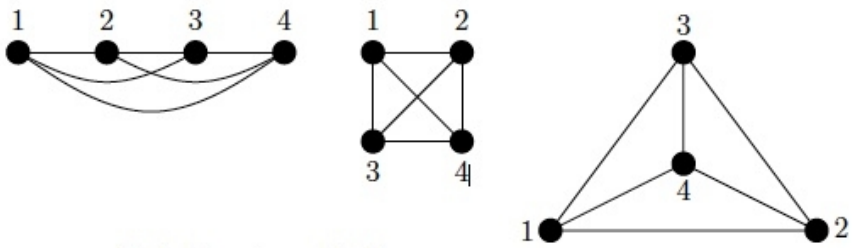






Bemerkung:

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

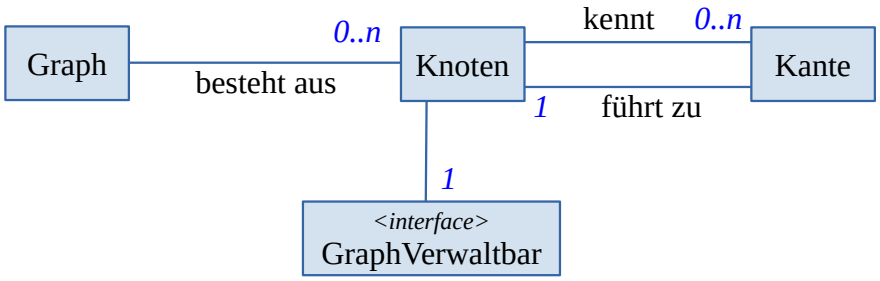
Implementierung eines Graphen

Es gibt grundsätzlich zwei verschiedene Arten, einen Graphen zu implementieren: Mithilfe von Listen oder unter Verwendung von Arrays.

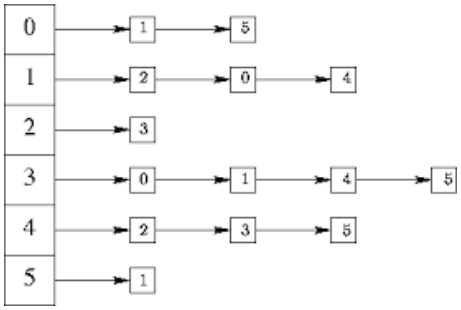
Adjazenzliste

Bei dieser Variante würde man die bereits bekannte Datenstruktur LISTE nutzen um einen Graphen zu schaffen, der grenzenlos wachsen kann.

Ein Graph besteht hier aus einer Liste von Knoten. Jeder Knoten verwaltet außerdem eine Liste von „Nachbarknoten“, zu denen direkt eine Kante verläuft. Diese Liste von direkten „Nachbarn“ nennt man dann ***Adjazenzliste*** (Adjazenz = Nachbarschaft).



Beispiel:

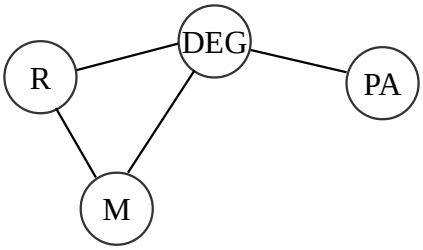


Graphen dieser Struktur sind besonders gut geeignet, wenn sich die Menge der Knoten und Kanten sehr häufig ändert oder stetig zunimmt, wie z.B. bei sozialen Netzwerken.

Wir werden diesen Ansatz hier nicht weiter verfolgen …

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 *boolean* (für ungewichtete Kanten) oder *int* (gewichtete Kanten).



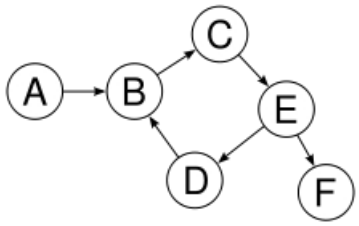
	M	R	PA	DEG
M	0	1	-1	1
R	1	0	-1	1
PA	-1	-1	0	1
DEG	1	1	1	0

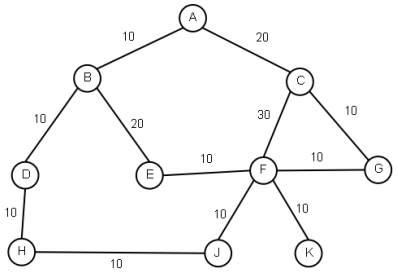
Da in der Matrix sowohl ein Feld für z.B. M→PA als auch ein Feld für PA→M zur Verfügung steht, kann man sehr leicht auch gerichtete Kanten realisieren.

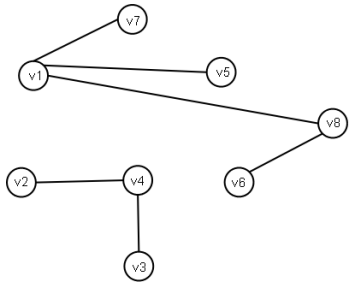
Hinweis: In der Regel gibt es keine Kanten von einem Knoten zu sich selbst, weswegen dieses Feld in der Matrix mit 0 belegt wird, es entsteht die sog. ***Nulldiagonale***. Existiert keine Kante wird das in einer Integer-Matrix durch den Wert -1 gekennzeichnet.

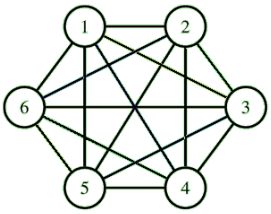
Aufgabe 2:

Gib für jeden der folgenden Graphen die Adjazenzmatrix an.









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 x (Länge der Zeile, sprich: waagrecht) und das zweite für y (Länge der Spalte, sprich: senkrecht).

Ein **Spaltendurchlauf** wird wie folgt implementiert:

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

Ein **Zeilendurchlauf** analog mit:

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

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

```
for (int x = 0 ; x < knoten.length ; x++) {
    for (int y = 0 ; y < knoten.length ; y++) {
```

```
            … kanten[x][y] …
        }
    }
```

Es ergibt sich dadurch folgende Feldernummierung:

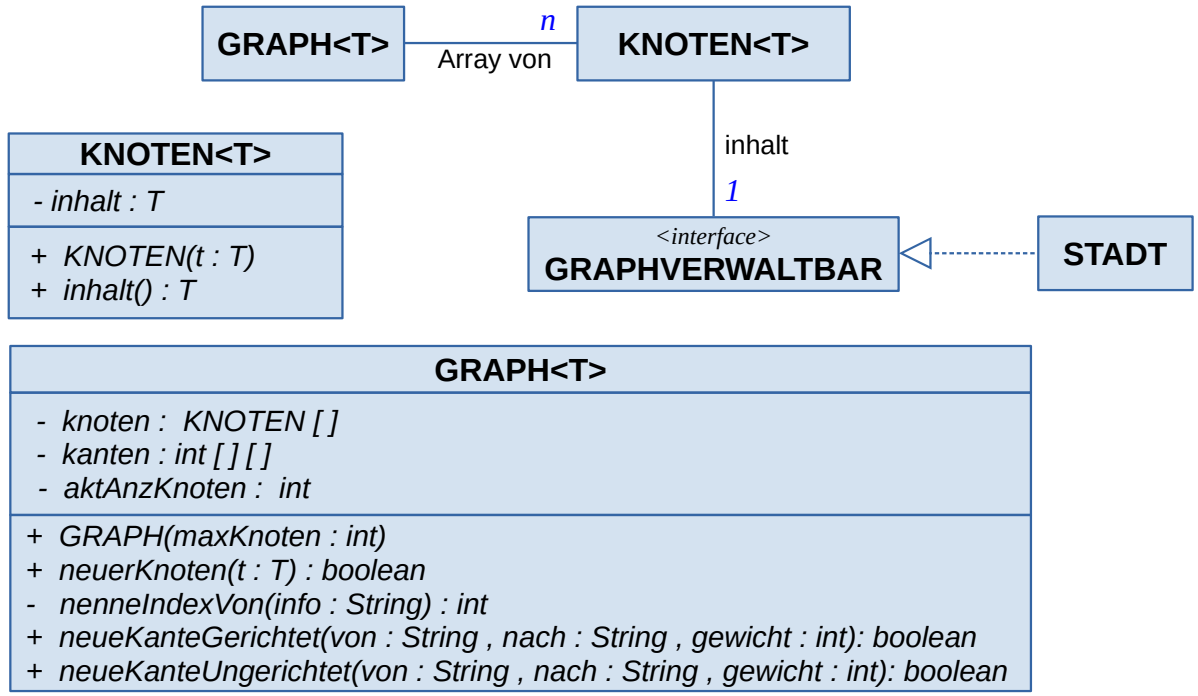
[0][0]	[0][1]	[0][2]	[0][3]
[1][0]	[1][1]	[1][2]	[1][3]
[2][0]	[2][1]	[2][2]	[2][3]
[3][0]	[3][1]	[3][2]	[3][3]

Aufgabe 3:

Beschreibe genau die Reihenfolge der Zellen bei obiger Doppelschleife.

Aufgabe 4:

a) Implementiere nun einen Graphen mit Hilfe einer Adjazenzmatrix nach folgender Vorlage:



- Jede Stadt besitzt nur einen Namen (info-String), der beim Konstruktor übergeben wird.

- Die Größe der Adjazenzmatrix und des Knoten-Arrays wird im Konstruktor gesetzt.

- Die Werte der Adjazenzmatrix werden im Konstruktor auf -1 oder 0 gesetzt.

- `neuerKnoten(…)` fügt einen neuen Knoten in das Knoten-Array ein, falls noch Platz ist. Bei erfolgreichem Einfügen wird true geliefert.

- Die Methode `nenneIndexVon(…)` gibt den Index eines Knotens vom Array knoten zurück, wenn man nach dem Info-String sucht. Es ist eine Hilfsmethode zu den Methoden `neueKanteXXX(…)`, um die Knoten der Matrix zuzuordnen, vgl. hierzu die nachfolgende Veranschaulichung.

- Die Methode `nenneIndexVon(…)` gibt -1 zurück, falls es keinen Knoten mit dem entsprechenden Suchstring gibt.

- Die beiden Methoden neueKanteXXX(…) suchen sich mittels nenneIndexVon(…) die passenden Indizies von den übergebenen Städten. Sofern die Städte bereits im Knoten-Array vorhanden sind und die Kante noch nicht existiert, soll die neue Kante passend in die Matrix eingefügt werden.

Zuordnung über den Index

Die Datenstruktur Graph wird durch zwei Felder realisiert:

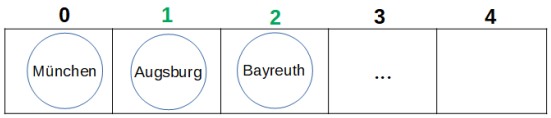
- ***knoten***: speichert die zu vernetzenden T's (hier die graphverwaltbaren Städte)
- ***kanten***: speichert nur die Information ob eine Kante (mit evtl. Gewicht) existiert.

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

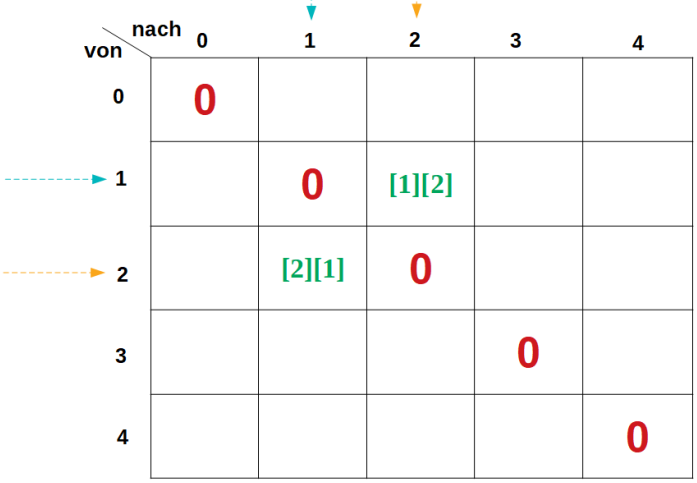
Beispiel:

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:



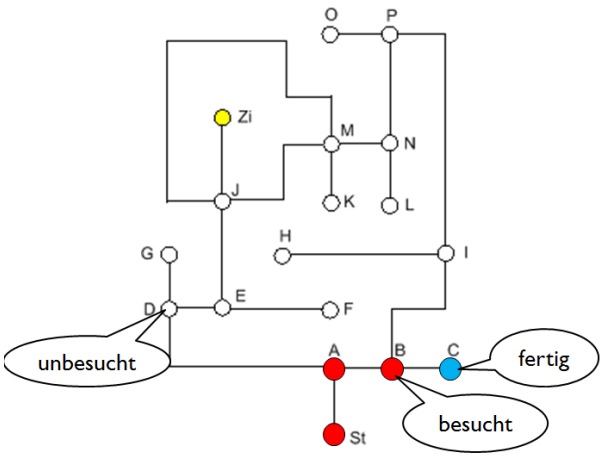
Graphendurchläufe

Es gibt viele Anlässe, die es nötig machen, einen Graphen 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. Theseus sucht den Minotaurus,…

Tiefensuche / Tiefendurchlauf

Eine Strategie einen Graphen zu durchlaufen besteht darin zunächst immer weiter in die Tiefe des Graphen vorzugehen, man spricht vom sog. ***Tiefendurchlauf***.

- **unbesucht**: Ausgangszustand
- **besucht**: Erfasste Kreuzungsknoten, bei denen noch nicht alle Nachbar-knoten vollständig durchsucht worden sind.
- **fertig**: Alle abgehenden Zweige des Knotens wurden durchsucht.



Tiefendurchlaufalgorithmus

1. Wiederhole bis alle Knoten auf den Zustand fertig gesetzt sind:
2. Setze den aktuellen Knoten auf **besucht** und wähle als Folgeknoten den numerisch bzw. alphabetisch niedrigsten unbesuchten Knoten.

   (ausgehend vom aktuellen Knoten)
3. Hat ein Knoten keine unbesuchten Nachbarknoten (mehr), setze ihn auf **fertig**: Alle abgehenden Zweige des Knotens wurden durchsucht.

   Es wird zum Vorgängerknoten zurückgekehrt. (***Backtracking***)

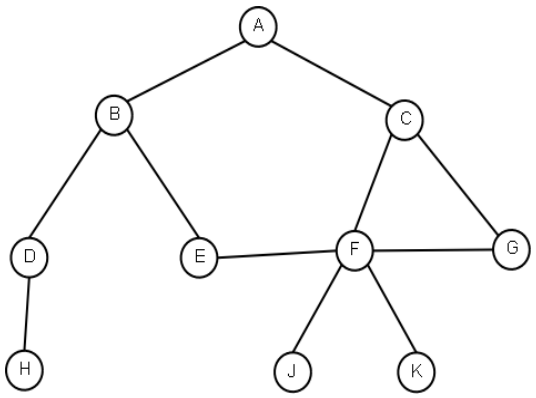
Für jeden Knoten wird exakt der gleiche Algorithmus ausgeführt. Es handelt sich demnach um einen ***rekursiven Algorithmus***:

„Gehe zu einem Knoten an dem du noch nicht warst und führe von dort den Tiefendurchlauf aus.“

Aufgabe 5:

Gib den Tiefendurchlauf für den abgebildeten Graphen an, indem du nacheinander die besuchten Knoten notierst. (Auch doppelt besuchte Knoten!)

Starte bei Knoten A und zeichne den entstehenden Tiefensuchbaum!



Implementierung des Tiefendurchlaufs

Folgende Hilfestellungen erleichtern die Implementierung der Methode `tiefendurchlauf(String startknoten)`:

- jeder Knoten kann besucht und fertig gesetzt werden. Setze dies mit booleschen Attributen in der Klasse KNOTEN um (inkl. sondierende/verändernde Methoden)
- Zum Setting wird jeder Knoten nach dem Aufrufen der Tiefensuche als unbesucht gesetzt und der Startknoten besucht, d.h die Hilfsmethode aufgerufen.
- Lagere die eigentliche Tiefensuche in eine Hilfsmethode `besuchenTiefe(int startnummer)` aus.

- Besuchen setzt den aktuellen Knoten auf besucht (inkl. Konsolenausgabe) und stößt dann die Rekursion auf alle unbesuchten Nachbarknoten an.

  → Überlege dir, wie man über die Kantenmatrix herausfindet, welche Knoten besucht werden müssen!

Beispiel:
Beginn bei München

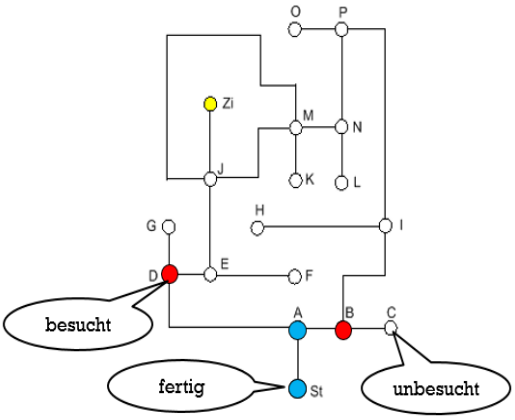
	IN	M	A	UL
IN	0	1	1	-1
M	1	0	1	-1
A	1	1	0	1
UL	-1	-1	1	0

- Sind alle Nachbarknoten besucht, so wird der besuchte Knoten (inkl. Konsolenausgabe) auf fertig gesetzt.
- Teste deine Methode mit dem Testgraphen. Füge hierzu den Code des Konstruktors auf dem Datenaustausch-Server in deine Klasse.

Breitensuche / Breitendurchlauf

Eine Alternative zum Tiefendurchlauf ist der ***Breitendurchlauf***. Bei dieser Strategie werden diesmal zunächst alle (unbesuchten) Nachbarknoten besucht, bevor man in die Tiefe des Graphen vordringt. Wieder gibt es die drei Knotenzustände:

- **unbesucht**: Ausgangszustand
- **besucht**: Erfasste Kreuzungsknoten, bei denen noch nicht alle Nachbarknoten vollständig durchsucht worden sind.
- **fertig**: Alle abgehenden Zweige des Knotens wurden durchsucht.



Breitendurchlaufalgorithmus

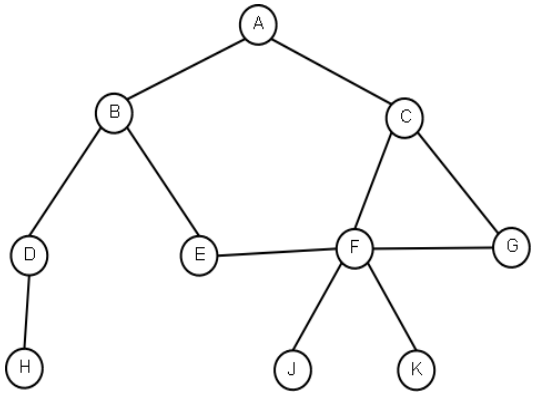
1. Wiederhole bis alle Knoten auf den Zustand fertig gesetzt sind:
2. Setze den aktuellen Knoten auf **besucht** und besuche in numerischer bzw. alphabetischer Reihenfolge alle unbesuchten Nachbarknoten.
3. Hat ein Knoten keine unbesuchten Nachbarknoten setze ihn auf **fertig**: Neuer Ausgangsknoten wird immer der am frühesten besuchte (Nachbar-) Knoten.

Hinweis: Es muss sich gemerkt werden, in welcher Reihenfolge die Nachbarknoten besucht werden, um anschließend den neuen Ausgangsknoten bestimmen zu können.

Aufgabe 6:

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

Starte bei Knoten A und zeichne den entstehenden Breitensuchbaum!



Implementierung des Breitendurchlaufs

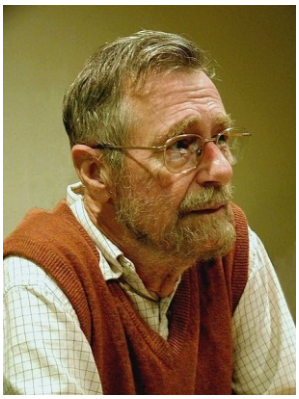
Folgende Hilfestellungen erleichtern die Implementierung der Methode `breitendurchlauf(String startknoten)`:

- jeder Knoten kann besucht und fertig gesetzt werden. Setze dies mit booleschen Attributen in der Klasse KNOTEN um (inkl. sondierende/verändernde Methoden)
- Zum Setting wird jeder Knoten nach dem Aufrufen der Breitensuche als unbesucht bzw. unfertig gesetzt und der Startknoten besucht, d.h die Hilfsmethode aufgerufen.
- Lagere die eigentliche Breitensuche in eine Hilfsmethode `besuchenBreite(int startnummer)` aus.
- Besuchen setzt den aktuellen Knoten auf besucht (inkl. Konsolenausgabe).
- Es muss fortan gespeichert werden, welche Knoten nacheinander vom Ausgangsknoten besucht werden, dies geschieht über ein Array `schlange`, das die Knotennummern speichert. (Ein Index, der mitzählt, speichert die aktuell freie Position)
- Zuerst wird der Ausgangsknoten eingereiht. Anschließend werden alle Nachbarn als besucht gekennzeichnet und ins Array aufgenommen. (+ Konsolenausgabe besucht)

Dijkstra-Algorithmus / Kürzeste Wege in Graphen

Der Dijkstra-Algortihmus findet alle kürzesten Wege in einem Graphen ausgehend von einem Startknoten.

Vorgehensweise:

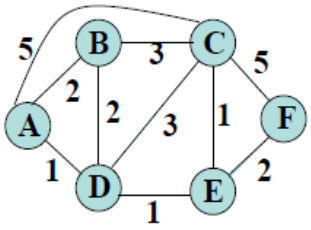


Edsger Wybe Dijkstra
(1930 – 2002)

1. Markiere den Startknoten, weise ihm die Distanz 0 zu und verwende ihn als aktuellen Knoten k.
2. Untersuche alle Nachbarknoten n von k:
   - Berechne den ***Gesamtaufwand*** von n ausgehend von k.
   - Sollte n so durch eine kleinere Distanz als bisher oder erstmalig erreicht werden können, notiere die Distanz und den Vorgängerknoten. (***Aktualisieren***!)
3. Wähle unter den noch nicht erreichten (nicht gelisteten) Knoten den mit dem geringsten Gesamtaufwand und verwende ihn als neuen aktuellen Knoten.
4. Sind noch nicht alle Knoten markiert, so fahre bei 2. fort.

Beispiel:

Führe den Dijkstra-Algorithmus ausgehend vom Knoten A durch.



d(K) = Abstand zum Knoten K

V = Vorheriger Knoten der die aktuell kürzeste Distanz zum betrachteten Knoten besitzt

Bsp: ***d(C),V = 4,F*** bedeutet:

Knoten C wird über den Vorgängerknoten F mit Gesamtaufwand 4 erreicht.

Erreichte Knoten	d(A), V	d(B), V	d(C), V	d(D), V	d(E), V	d(F), V
A						

Implementierung des Dijkstra-Algorithmus

Folgende Hilfestellungen erleichtern die Implementierung der Methode `dijkstra(String startknoten)`:

- Jeder Knoten kann auf besucht gesetzt werden. Außerdem wird für jeden Knoten dessen Gesamtentfernung zum Startknoten und der jeweilige Vorgängerknoten über dessen Indexnummer gespeichert. (nenne- und setze-Methoden nicht vergessen!)
- Zu Beginn wird wieder der „Reset“ durchgeführt, nur um sicher zu gehen…
- Programmiere die Hilfsmethode **`nenneKnotenMitKleinsterEntfernung()`**
  - durchlaufe alle Knoten…
  - … und filtere den am kürzesten zu erreichenden, unbesuchten Knoten heraus.
  - … definiere u.A. eine lokale Variable, die die minimale Distanz speichert. Belege sie zu Beginn mit Integer.MAX_VALUE,
- Programmiere die Hilfsmethode **`besuchenDijkstra(int i)`**
  - durchlaufe alle Knoten…
  - … falls es eine Verbindung gibt und der Knoten noch unbesucht ist,…
  - … prüfe zwei Fälle:
    - der Knoten wird erstmalig erreicht:
      → Entfernung setzen, Vorgänger setzen und Konsolenausgabe
    - der Knoten kann schon erreicht werden: → Erst prüfen, ob die neue Entfernung kürzer ist als die bisher eingetragene: nur dann aktualisieren.
  - Sind alle Knoten durchlaufen, so ist der Ausgangsknoten besucht. Zudem sollte hier der neue Knoten ausgegeben werden, zu welchem vom Startknoten aus eine neue kürzeste Verbindung gefunden wurde.
- Zurück zur Hauptmethode **`dijkstra(…)`**: Nach dem Reset…
  - Initialisiere den Startknoten, d.h. Entfernung, Vorgänger und Besucht setzen.
  - Besuche den Start-Knoten. Dies entspricht den Füllen der ersten Zeile unserer Dijkstra-Tabelle (Bestandsaufnahme)
  - Durchlaufe alle Knoten und suche den Knoten mit der kleinsten Entfernung (falls vorhanden). Besuche diesen Knoten.
- Zusatzaufgabe: Programmiere eine Methode `nenneRoute(String ziel)` mithilfe einer ArrayList, die die zu fahrende Route (anhand der Vorgängerknoten) zu einem Zielknoten ausgibt, nachdem man die Methode dijkstra(...) ausgeführt hatte.