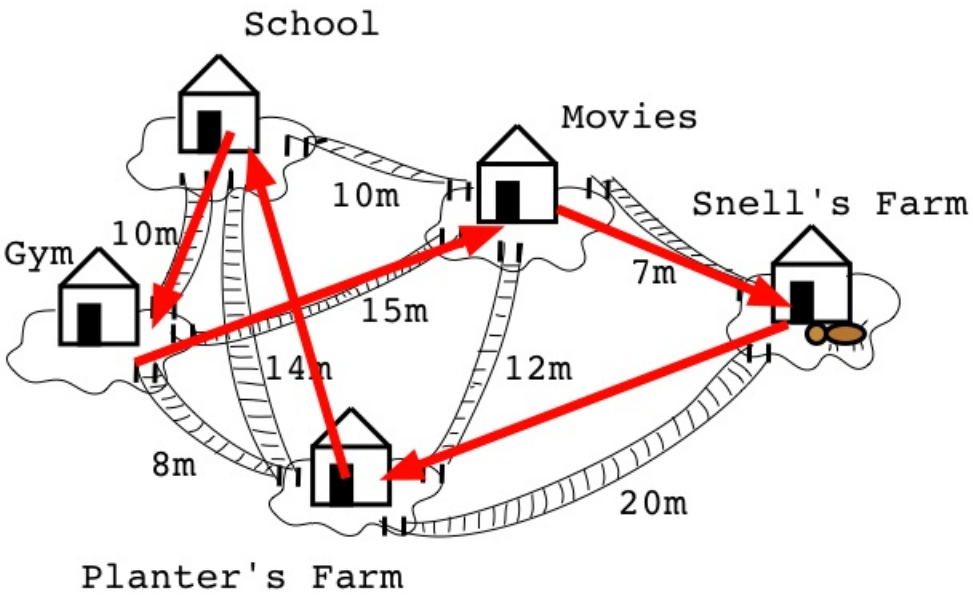
Skript zu Graphen



Lars Wechsler

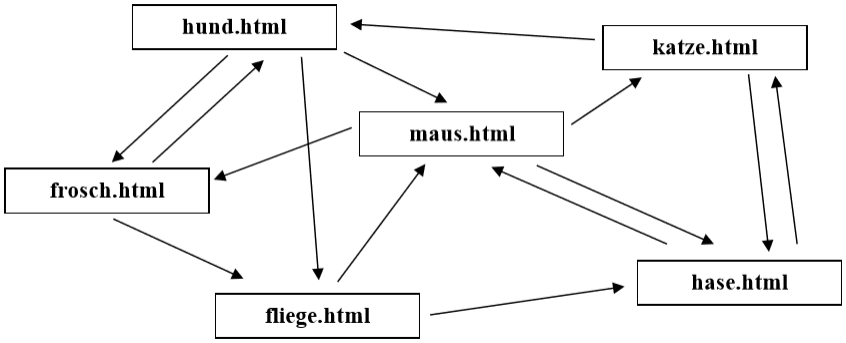
9. März 2023

Inhaltsverzeichnis

1 Grundlagen

1.1 Einleitung

Auch der Begriff des Graphen ist - wie auch der Baum - bereits Thema in der 7. Jahrgangsstufe! Ein typisches Beispiel für einen Graphen ist hier die Struktur einer Website bzw. später allgemein des Internets. Die betrachteten Graphen werden hier z.B. so dargestellt:



Jeder **Knoten** stellt hier eine HTML-Datei dar, die Pfeile (**Kanten**) zeigen eine Verlinkung an - häufig wird statt zwei Pfeilen auch ein Pfeil mit zwei Spitzen verwendet, wenn die Beziehung in beide Richtungen gültig ist.
Ein weiteres typisches Einsatzgebiet von Graphen sind Verbindungen zwischen Orten, z.B. beliebig auf einer Karte:

Hier entspricht jeder Ort von Interesse einem Knoten und die Verbindungswege entsprechen den Kanten - hier ergeben sich bereits die ersten Modellierungsfragen, z.B.:

- Welche Orte sollen Knoten werden?
- Sollen Orte verbunden werden, wenn es einen Weg gibt?
- Falls es mehrere Wege gibt, welcher wird repräsentiert? Der kürzeste oder der „schnellste“? (siehe Navigationsprogramme!)

- etc.

Je nach Modellierung können dann verschiedene Probleme gelöst werden, z.B. die oben bereits erwähnte Routensuche nach dem kürzesten (oder schnellsten) Weg.
Ist die Situation spezieller, so werden manche Fragen automatisch von den Gegebenheiten beantwortet, im Folgenden findet sich beispielsweise ein Plan des U-Bahn-Netzes von münchen:

Modelliert man diese Situation als Graph, so ist offensichtlich jeder Knoten eine Station, jede Kante eine Verbindungsstrecke zwischen zwei Stationen - hier gibt es in der Regel nicht mehrere Wege dazwischen!
Die Graphentheorie ist aber nicht auf „Wegeprobleme“ beschränkt, weitere Beispiele:

- **Soziale Netzwerke**: Graphen können auch Beziehungen zwischen Menschen modellieren.
- **Computernetzwerke**: Ein Graph kann auch z.B. ein lokales Netwerk repräsentieren.
- **Datenbanken**: Modellierung von Zusammenhängen von Daten.
- **„Reale Situationen“**: z.B. Modellierung eines Rohrsystems in einem Gebäude zur Analyse von notwendigen Pumpen.
- etc.

1.2 Grundbegriffe

Zurück zu den Anfängen:

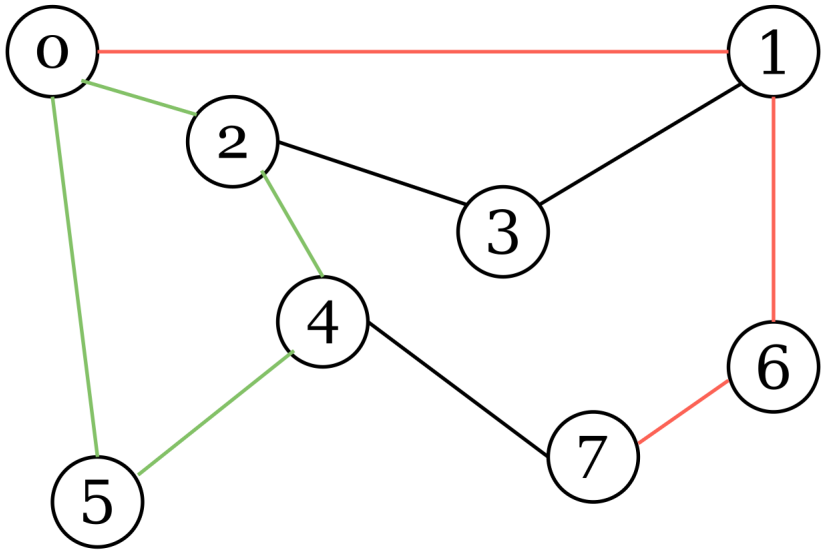
Definition 1. *Ein Graph ist eine Datenstruktur, die eine Menge von Objekten (Knoten - Nodes) repräsentiert, sowie deren Verbindungen (Kanten - Edges).*

Diese Definition macht klar, dass ein Graph im Wesentlichen eine Verallgemeinerung eines Baumes ist, es gibt jetzt keine Einschränkungen mehr bezüglich der Anzahl der Kinder und auch keinen ausgezeichneten Wurzelknoten. Dadurch verlieren wir zwar auch Vorteile, sind in der Modellierung aber wesentlich offener.

Die wichtigsten (offensichtlichen) Eigenschaften bzw. Forderungen an Knoten und Kanten sind:

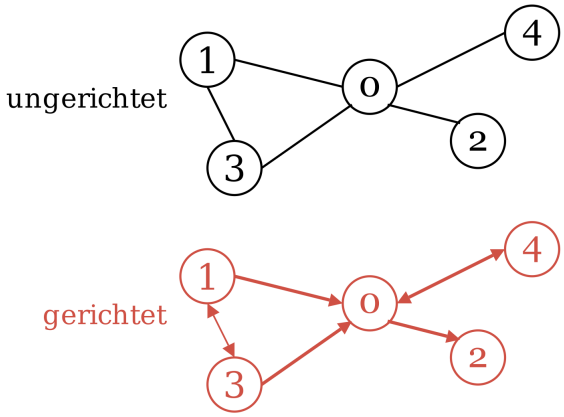
- es gibt nur eine endliche Anzahl an Knoten und Kanten
- jede Kante verbindet genau zwei Knoten
- Knoten und Kanten können jeweils noch Zusatzinformationen verwalten
- Zwischen zwei Knoten kann es Kanten „in beide Richtungen“ geben, dies ist aber nicht zwingend notwendig.
- Knoten und Kanten werden idealisiert dargestellt (z.B. Wegeanalyse: auch wenn ein Weg zwischen zwei Orten „kurvenreich“ ist, werden die beiden Knoten mit einer geraden Linie verbunden, die Länge des Weges kann dann z.B. im **Kantengewicht** gespeichert werden, siehe unten.)

Wir betrachten das untenstehende Beispiel:



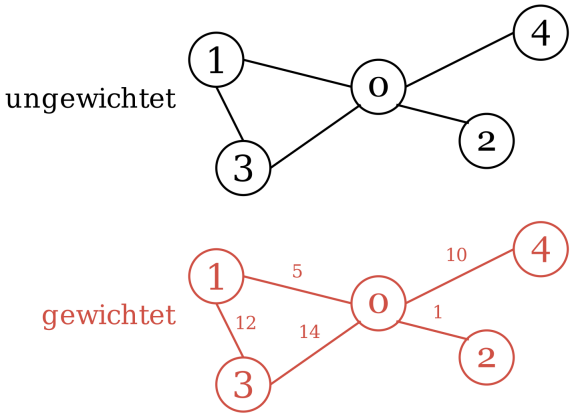
Ein Weg von einem Knoten zu einem anderen Knoten wird **Pfad** genannt, wenn es eine endliche Folge aufeinander folgender, durch entsprechende Kanten miteinander verbundene Knoten gibt. Wenn dabei kein Knoten mehrfach besucht wird, so spricht man von einem **einfachen Pfad**. Im obigen Beispiel wäre 0 1 6 7 ein einfacher Pfad. Ist in einem Graph ein „Rundlauf“ möglich, so spricht man von einem **Zyklus**. Im obigen Graphen gibt es beispielsweise den Zyklus 0 2 4 5 . Im Wesentlichen ist ein Zyklus also ein Pfad, dessen Start- und Endknoten identisch sind.
Weitere wichtige Fachbegriffe

1. **gerichtet vs. ungerichtet**: Selbsterklärend :) Beispiel:

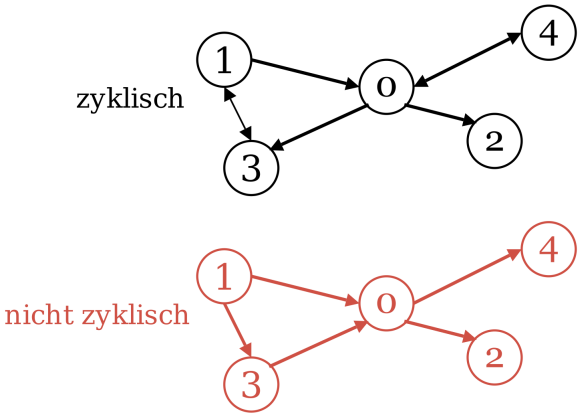


2. **gewichtet vs. ungewichtet**: Die Kante eines Graphen kann einerseits bedeuten, dass die beiden Knoten „nur“ verbunden sind, andererseits kann sie aber auch eine konkrete (physikalische) Größe repräsentieren. In

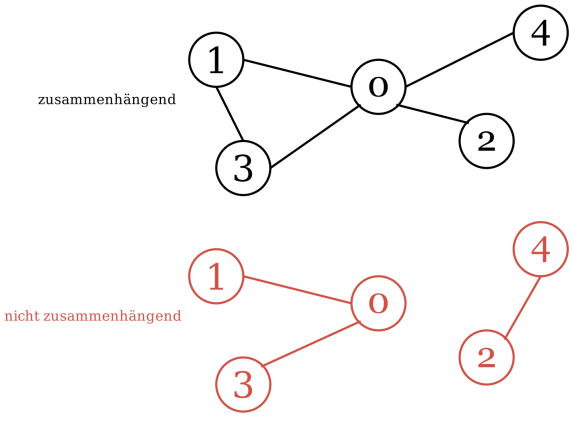
unserem U-Bahn-Beispiel könnten die Kanten Fahrtzeiten in Minuten oder Entfernungen in km darstellen. In einem sozialen Netwerk könnte ein Kantengewicht die Interaktionsfrequenz zweier Menschen beschreiben, usw.:



3. **zyklisch vs. nicht zyklisch**: diese Unterscheidung ist nur für gerichtete Graphen interessant. Der Graph heißt bereits zyklisch, wenn es einen einzelnen Zyklus irgendwo im Graphen gibt.

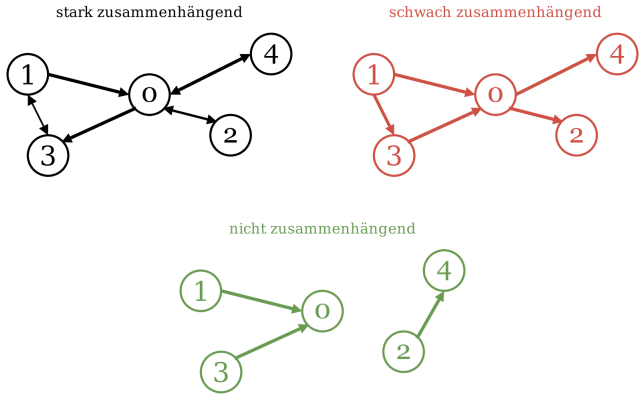


4. **zusammenhängend vs. nicht zusammenhängend**: Hier muss noch einmal weiter zwischen gerichtet und ungerichtet unterschieden werden:
ungerichtet: hier bedeutet **zusammenhängend**, dass jeder Knoten von jedem anderen aus erreicht werden kann, d.h. für zwei beliebige Knoten A und B gibt es einen Pfad von A nach B. Gibt es dagegen mindestens ein Knotenpaar, zu dem kein Pfad existiert, so ist der Graph **nicht zusammenhängend**.

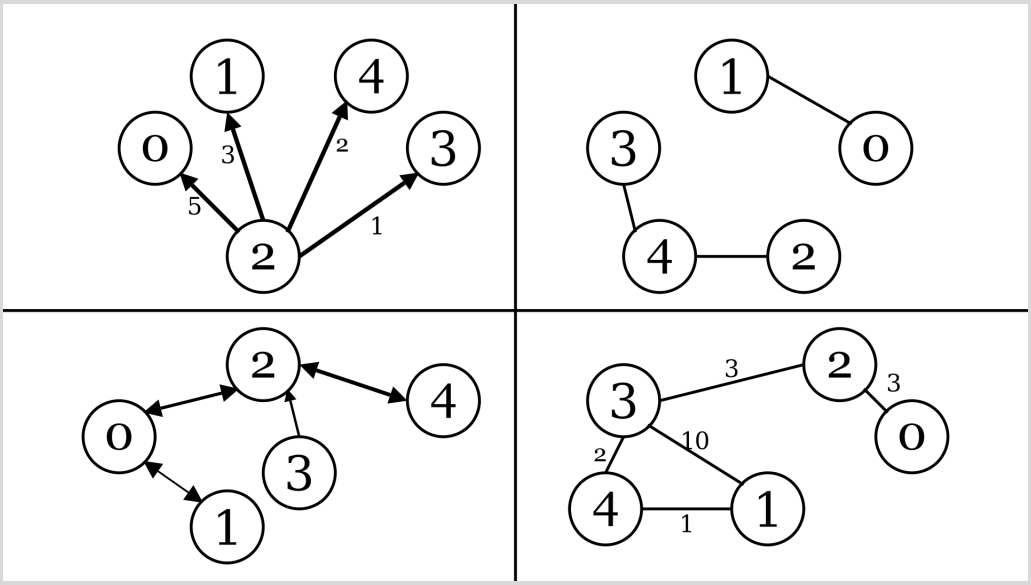


gerichtet: Wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist, so spricht man von einem **stark zusammenhängenden** Graphen. Es wird jetzt aber noch feiner unterteilt, in beiden Fällen gilt, dass mindestens ein Knoten von mindestens einem anderen Knoten aus nicht erreichbar ist, aber:

- Interpretiert man den Graphen als ungerichtet und danach sind wieder alle Knoten von allen anderen aus erreichbar, so spricht man noch von **schwach zusammenhängenden** Graphen.
- Ist auch in einem entsprechenden ungerichteten Graphen mindestens ein Knoten nicht erreichbar, so ist der ganze Graph **nicht zusammenhängend**.



Aufgabe 1: Klassifizieren Sie die folgenden Graphen mit möglichst vielen Fachbegriffen möglichst genau:



Aufgabe 2: Entwefen und zeichnen Sie einen Graphen, der ...

- ein U-Bahn-Netz modelliert.
- die Wanderwege in einem Nationalpark modelliert.
- ein soziales Netzwerk modelliert.

Verwenden Sie jeweils mindestens 10 Knoten und gegebenenfalls realistische Kantengewichte und erläutern Sie deren Bedeutung.

Aufgabe 3: Entwerfen Sie jeweils einen Graphen mit 5 Knoten und möglichst vielen Kanten bzw. Richtungen (dabei zählt jede Richtung, auch eine ungerichtete Kante zählt also als 2 Richtungen!)

- Der Graph ist ungerichtet und nicht zusammenhängend.
- Der Graph ist gerichtet und zykelfrei.
- Der Graph ist gerichtet und schwach zusammenhängend.

Aufgabe 4: Recherchieren Sie, was es mit dem „Vier-Farben-Satz“ auf sich hat, sowie seine Verbindung zur Graphentheorie! Färben Sie anschließend eine Karte der deutschen Bundesländer passend.

Aufgabe 5: Überlegen Sie sich das Klassendiagramm einer eigenen Implementierung eines Graphen, orientieren Sie sich dabei an den folgenden Fragen:

Grundlagende Fragen/Überlegungen:

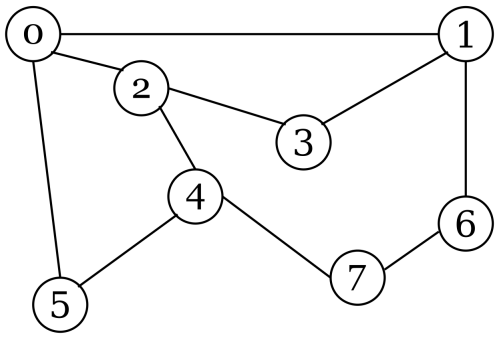
- Welche Klassen sind nötig?
- Wie sollen Knoten repräsentiert werden?
- Wie werden die Kanten repräsentiert?
- Wie können Kantengewichte codiert werden?

Lesen Sie erst anschließend im nächsten Kapitel weiter!

2 Implementierung Adjazenzmatrix

Nach diesen Vorüberlegungen und Grundbegriffen können wir uns der Implementierung eines Graphen widmen. Die Struktur der Knoten bzw. der Speicherung von Daten in den Knoten können wir dabei von Bäumen bzw. Listen übernehmen. Das entscheidende neue Element ist also die Repräsentation der Kanten und deren Gewichte.

Die beiden häufigsten Möglichkeiten der Implementierung verwenden eine **Adjazenzmatrix** bzw. **Adjazenzlisten**, in diesem Kapitel werden wir uns der Matrix widmen.
Wir betrachten wieder ein Beispiel aus den Grundlagen:



Es handelt sich um einen ungerichteten, zusammenhängenden Graphen. Will man die Verbindungen zwischen den einzelnen Knoten darstellen, bietet es sich an „von Knoten zu Knoten“ zu denken. Man betrachtet eine Matrix nach folgendem Schema (die einzelnen Knoten werden zusätzlich zu ihrem Index noch mit k benannt, um später leichtere Lesbarkeit zu gewährleisten):

	k0	k1	k2	k3	k4	k5	k6	k7
k0								
k1								
k2								
k3								
k4								
k5								
k6								
k7								

Wir betrachten exemplarisch die Kante zwischen den Knoten 0 und 2. Häufig interpretiert man die Tabelle so, dass auf der linken Seite - also den Zeilen - die Startknoten abgebildet werden, die Spalten dagegen bilden die Zielknoten. Wollen wir also eintragen, dass eine Kante von 0 nach 2 läuft, suchen wir in der Zeile $k0$ die Spalte $k2$ und markieren die entsprechende Zelle:

	k0	k1	k2	k3	k4	k5	k6	k7
k0			x					
k1								
k2								
k3								
k4								
k5								
k6								
k7								

Da es sich um einen ungerichteten Graphen handelt, geht aber auch ein Weg von 2 nach 0, also muss direkt auch in der Spalte $k0$ und Reihe $k2$ eine Markierung gesetzt werden.
Dies wird für alle Einträge der Fall sein! Man spricht hier auch von einer **symmetrischen** Matrix, d.h. es genügt sich alle Einträge in der „oberen rechten Hälfte“ anzusehen, d.h. von kleineren zu größeren Zahlen. Danach kann „nach unten links“ gespiegelt werden. Zunächst tragen wir aber alle weiteren Kanten von kleinerer zu größerer Zahl ein:

	k0	k1	k2	k3	k4	k5	k6	k7
k0		x	x			x		
k1				x			x	
k2				x	x			
k3								
k4						x		x
k5								
k6								x
k7								

Mit den gespiegelten Werten ergibt sich:

	k0	k1	k2	k3	k4	k5	k6	k7
k0		x	x			x		
k1	x			x			x	
k2	x			x	x			
k3		x	x					
k4			x			x		x
k5	x				x			
k6		x						x
k7					x		x	

Jetzt stellt sich die Frage, wie eine solche Matrix in einem Programm darstellen lässt. Die einfachste Variante ist ein zweidimensionales Array. In Bezug auf den Datentyp des Feldes gibt es verschiedene Möglichkeiten. Für einen ungewichteten Graphen ist nur wichtig, ob eine Verbindung zwischen zwei Knoten vorhanden ist, da die Kante keine zusätzlichen Informationen enthält, man könnte hier also ein Boolean-array verwenden, die Tabelle sähe also wie folgt aus:

	k0	k1	k2	k3	k4	k5	k6	k7
k0	False	True	True	False	False	True	False	False
k1	True	False	False	True	False	False	True	False
k2	True	False	False	True	True	False	False	False
k3	False	True	True	False	False	False	False	False
k4	False	False	True	False	False	True	False	True
k5	True	False	False	False	True	False	False	False
k6	False	True	False	False	False	False	False	True
k7	False	False	False	False	True	False	True	False

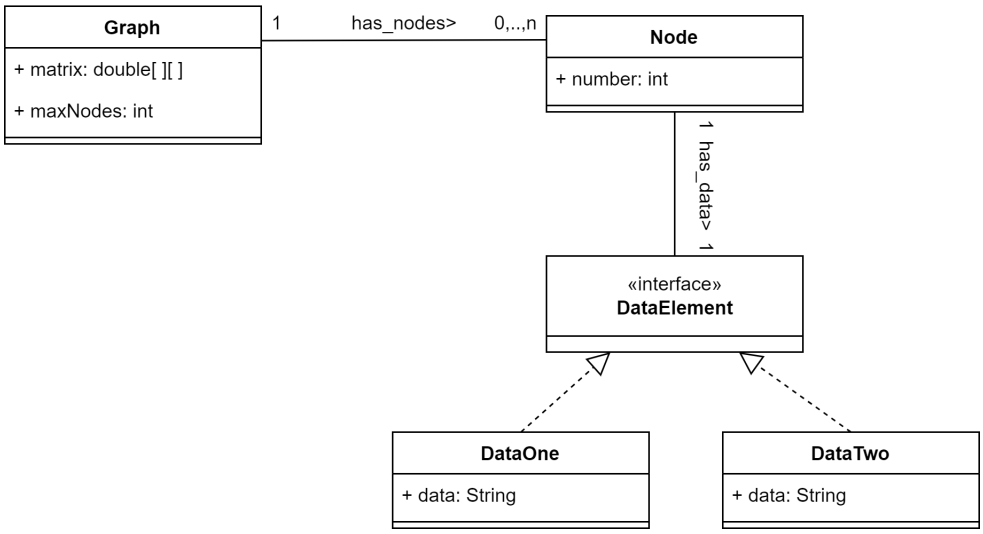
Sobald allerdings Kantengewichte eine Rolle spielen, ist ein boolean-array nicht mehr ausreichend, da die Gewichte nicht mehr anderweitig gespeichert werden sollten. Geht man davon aus, dass sich alle Informationen, die man codieren möchte in irgendeiner Form durch Zahlen ausdrücken lassen, so wäre ein **double**[ ][ ]-array geeignet, um möglichst viele Spielarten von Graphentypen abbilden zu können. In unserem Fall ohne Gewichte könnte dann beispielsweise 1 für „Kante existiert“ und 0 für „Kante existiert nicht“ stehen:

	k0	k1	k2	k3	k4	k5	k6	k7
k0	0	1	1	0	0	1	0	0
k1	1	0	0	1	0	0	1	0
k2	1	0	0	1	1	0	0	0
k3	0	1	1	0	0	0	0	0
k4	0	0	1	0	0	1	0	1
k5	1	0	0	0	1	0	0	0
k6	0	1	0	0	0	0	0	1
k7	0	0	0	0	1	0	1	0

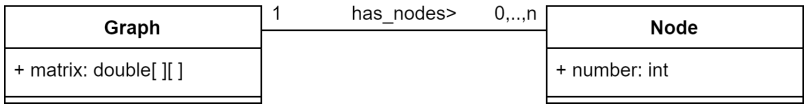
Alternativ könnte auch z.B. -1 und 1 oder jede beliebige andere Kombination verwendet werden.
Diese Implementierung hat natürlich den Nachteil, das bei vielen Knoten, aber wenigen Kanten viele Einträge in der Matrix gleich 0 sind. Man spricht von einer „dünn besetzten Matrix“. Für unsere kleinen Graphen ist das nicht so schlimm, aber möchte man beispielsweise das Straßennetz einer Stadt - oder von ganz Deutschland, oder der Welt! - modellieren, stellt das einen unnötigen Verbrauch von Speicherplatz dar. In der „Realität“ wird häufig die Idee der Adjazenmatrix beibehalten, aber die Speicherung der Information wird auf andere Datenstrukturen als ein $n \cdot n$ - Feld ausgelagert (mit n der Anzahl der Knoten!). Verwendet werden z.B.:

- **Dictionarys** (DOK: Dictionary of keys: in Java z.B. HashTables, HashMaps): Es werden (Zeile, Spalte)-Paare auf den Kantenwert abgebildet.
- **Coordinate list**: es werden direkt (Zeile, Spalte, Kantenwert)-Tripel gespeichert.
- **Compressed sparse row** (Yale format): Es werden drei Arrays gespeichert: eines für die Kantenwerte, eines für die Spaltenindizes an denen diese stehen und eines für die Zeilenindices.

In allen drei Fällen werden die Null-Werte nicht mehr mitgespeichert, aber zurück zu unserer - deutlich einfacheren - Implementierung. Ein mögliches Klassendiagramm für unseren Graphen sieht wie folgt aus:



Das Attribut **maxNodes** ist nicht zwingend notwendig, es hängt davon ab, ob man den Graphen erweiterbar gestalten möchte oder nicht. Der Vorteil der Verwendung eines Maximums besteht darin, dass das zweidimensionale Feld direkt mit seiner endgültigen Größe erzeugt werden kann. Ansonsten müsste beim Hinzufügen eines Knotens immer die Adjazenzmatrix vergrößert werden, was nur durch Erzeugen eines neuen Arrays und Kopieren des Alten Arrays möglich wäre.
Im Folgenden wollen wir uns außerdem darauf beschränken, dass die Knoten des Graphen keine weiteren Informationen verwalten müssen außer ihrer eigenen Nummer. Wir wollen außerdem bei Erstellung des Graphen festlegen, wie groß er ist, da dies der Regelfall im Einsatz ist, d.h. ein grundlegendes Klassendiagramm unseres einfachstmöglichen Graphen sieht (noch ohne Methoden) wie folgt aus:



Aufgabe 1: Implementieren Sie die Grundstruktur des Graphen mit Adjazenzmatrix nach obigem vereinfachten Klassendiagramm, zusätzlich soll es folgendes geben:

- einen Konstruktor, der einen Graphen mit einer bestimmten Anzahl an Knoten erzeugt.
- einen Konstrukter, der einen Graphen erzeugt, wenn eine Knotenliste und eine Adjazenzmatrix übergeben werden.
- eine Methode, die eine Kante ohne Gewicht hinzufügt.
- eine Methode, die eine Kante mit Gewicht hinzufügt.
- für die beiden vorangehenden Methoden jeweils eine Methode, die außerdem die „Rückrichtung“ gleich mit setzt (für ungerichtete Graphen).
- eine Methode, die einen Knoten hinzufügt, wenn die Maximalzahl noch nicht erreicht ist.
- eine Methode, die prüft, ob zwischen zwei Knoten eine Kante existiert.
- eine Methode, die eine existierende Kante löscht und eine Option, die eine Kante in beide Richtungen löscht!.
- eine Methode, die die Adjazenzmatrix lesbar auf der Konsole ausgibt.

Die Lösung zu dieser Aufgabe findet sich auf der nächsten Seite.

Wir beginnen mit der Knoten-Klasse, hier gibt es aktuell noch nicht viel zu tun:

```
public class Node {

    private int number;

    public Node(int number) {
        this.number = number;
    }
}
```

In der Graphen-Klasse sind die Konstruktoren zuerst an der Reihe:

```
public class Graph {

    private double[][] matrix;
    private Node[] nodes;

    public Graph(double[][] matrix, Node[] nodes) {
        this.nodes = nodes;
        this.matrix = matrix;
    }


    public Graph(int nodeNumber) {
        if(nodeNumber <= 0) nodeNumber = 5;
        nodes = new Node[nodeNumber];
        for(int i = 0; i < nodeNumber; i++) {
            nodes[i] = new Node(i);
        }
        matrix = new double[nodeNumber][nodeNumber];
    }
}
```

Im zweiten Fall werden die Knoten von 0 bis $nodeNumber - 1$ erzeugt und im Knoten-Array gespeichert. Wir können ein Array verwenden, da wir davon ausgehen, dass sich die Größe unseres Graphen nicht mehr verändert. Ansonsten könnten wir natürlich auch eine Liste verwenden!

Die Methoden um die Kanten hinzuzufügen sind vergleichsweise simpel: es muss jeweils in der Matrix an der korrekten Stelle entweder eine 1 platziert werden, oder der Wert des übergebenen Gewichts für diese Kante. Damit ergeben sich die folgenden vier Methoden (jeweils noch eine zusätzlich, um gleich beide Kanten für einen ungerichteten Graphen zu erzeugen). Die vorangestellte checkInput()-Methode wird dazu verwendet, in jedem Fall zu prüfen, ob die übergebenen Indices zu unserem Knotenarray passen:

```
public boolean checkInput(int start, int end) {
    if(start < 0 || end < 0 || start > nodes.length || end > nodes.length) {
        System.out.println("There cannot be an edge here!");
        return false;
    }
    return true;
}

public void addEdge(int start, int end) {
    if(checkInput(start, end)) matrix[start][end] = 1;
}

public void addEdge(int start, int end, double weight) {
```

```
    if(checkInput(start, end)) matrix[start][end] = weight;
}

public void addEdgeBoth(int start, int end) {
    if(checkInput(start, end)) {
    matrix[start][end] = 1;
    matrix[end][start] = 1;
    }
}

public void addEdgeBoth(int start, int end, double weight) {
    if(checkInput(start, end)) {
        matrix[start][end] = weight;
        matrix[end][start] = weight;
    }
}
```

Wir gehen hier davon aus, dass die einzige relevante Information eines Knotens seine Nummer ist, mit der wir ihn erzeugt haben. Da diese Nummer auch der Position im Array entspricht (deswegen der Beginn bei 0), können wir den Methoden hier direkt die Positionen im Array übergeben, um die Kanten zu definieren. Möchte man sich nicht auf diese Reihenfolge verlassen bzw. sie benutzen, so muss jeder Knoten mit einem anderen Attribut eindeutig zuordbar sein. Dann kann jeweils dieses Attribut der *addEdge()* - Methode übergeben werden und die entsprechenden Knoten können im Array gesucht werden.
Bei der Methode, die prüft, ob eine Kante existiert, gibt es nicht viel zu beachten, ebensowenig bei den Methoden zum Löschen:

```
public boolean existsEdge(int start, int end) {
    if(!checkInput(start, end)) return false;
    if(matrix[start][end] != 0) {
        return true;
    }
    return false;
}

public void removeEdge(int start, int end) {
    if(checkInput(start, end)) matrix[start][end] = 0;
}

public void removeEdgeBoth(int start, int end) {
    if(checkInput(start, end)) {
        matrix[start][end] = 0;
        matrix[end][start] = 0;
    }
}
```

Die Ausgabe-Methode schreibt zusätzlich zu den Gewichten noch die Knotennamen in die oberste Spalte bzw. Zeile. Der Tabstop wird verwendet, damit die Zahlen hübscher untereinander angezeigt werden.

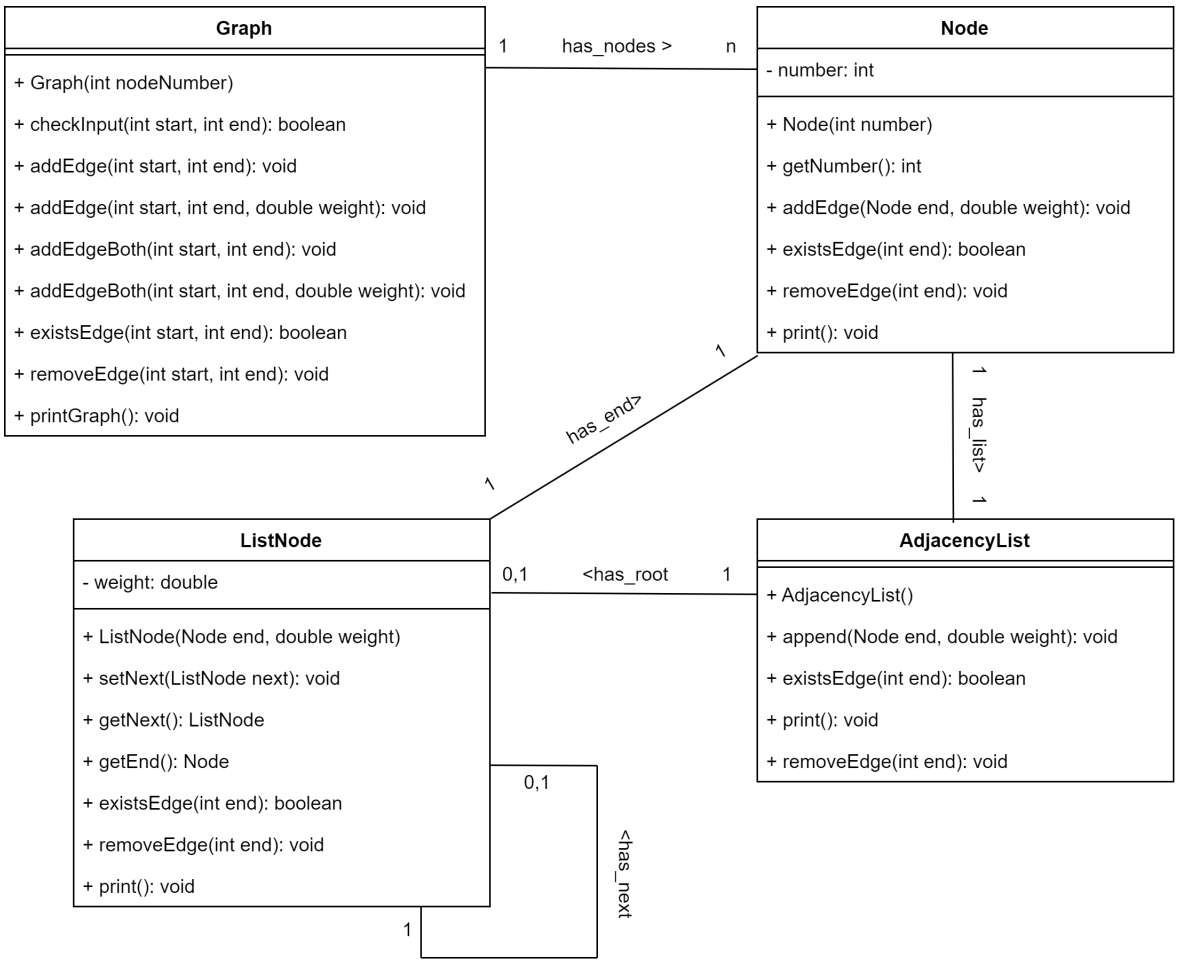
```
public void printMatrix() {
    System.out.print("\t");
    for(int i = 0; i < nodes.length; i++) {
        System.out.print("k" + i + "\t");
    }
    for(int i = 0; i < nodes.length; i++) {
        System.out.println();
        System.out.print("k" + i + "\t");
        for(int j = 0; j < nodes.length; j++) {
```

```
            System.out.print(matrix[i][j] + "\t");
        }
    }
}
```

Alle weiteren möglicherweise interessanten Methoden, werden wir in Kapitel 4 - Graphenalgorithmen - implementieren.

3 Implementierung Adjazenlisten

Eine zweite - wenn auch nicht so häufig verwendete - Möglichkeit der Implementierung ist die Verwendung von **Adjazenzlisten**. Die grundlegende Idee besteht dabei darin, für jeden Knoten eine Liste der anderen Knoten zu speichern. Da bei uns auch Kantengewichte repräsentiert werden sollen, können nicht einfach die Knoten, die von einem Knoten aus erreicht werden können in einem Array gespeichert werden. Ein Array wäre ohnehin für unseren Verwendungszweck ungeeignet, da wir Kanten hinzufügen und gegebenenfalls auch löschen wollen.
Jeder Eintrag der Liste repräsentiert also eine Kante und speichert das Gewicht und den Knoten zu dem diese Kante zeigt als Referenz in einem Attribut. Der Startknoten muss nicht gespeichert werden, wenn wir die Adjazenlisten in den einzelnen Knoten verwalten. Soll nur die Graph-Klasse alle Adjazenzlisten kennen, so könnte der Einfachheit halber der Startknoten zusätzlich gespeichert werden. Wie immer ein Klassendiagramm - diesmal auch mit den entsprechenden grundlegenden Methoden:



Hinweis: Die Adjazenzliste ist in dieser Implementierung nicht mit einem Kompositum versehen, da diese nicht viel „können“ muss. Das Einfügen der zusätzlichen Klassen rechtfertigt weder den Schreibaufwand, noch verbessert es die Performance. Auch der Überblick ist mit den wenigen Methoden nicht gefährdet.

Aufgabe - Die Einzige hier: Implementieren Sie das oben stehende Klassendiagramm.

Nachfolgend die Lösung, jeweils geordnet nach Klassen mit Kommentaren:

```
public class Graph {
    private Node[] nodes;

    //Analog zur anderen Implementierung
    public Graph(int nodeNumber) {
        if(nodeNumber <= 0) {
            nodeNumber = 5;
        }
        nodes = new Node[nodeNumber];
        for(int i = 0; i < nodes.length; i++) {
            nodes[i] = new Node(i);
        }
    }

    public boolean checkInput(int start, int end) {
        if(start < 0 || start >= nodes.length || end < 0 || end >= nodes.length) {
            System.out.println("Hier kann es keine Kante geben!");
            return false;
        }
        return true;
    }

    //Im Gegensatz zu vorher müssen wir bei allen Methoden auf dem Startknoten der
    //Kante eine Methode aufrufen, damit für die Adjazenzliste gegebenenfalls
    //ebenfalls eine passende Methode gestartet wird.
    public void addEdge(int start, int end) {
       if(checkInput(start, end)) nodes[start].addEdge(nodes[end], 0);
    }

    //Hier nur zusätzlich mit Gewicht!
    public void addEdge(int start, int end, double weight) {
        if(checkInput(start, end)) nodes[start].addEdge(nodes[end], weight);
    }

    public void addEdgeBoth(int start, int end) {
        if(checkInput(start, end)) {
            nodes[start].addEdge(nodes[end], 0);
            nodes[end].addEdge(nodes[start], 0);
        }
    }

    public void addEdgeBoth(int start, int end, double weight) {
        if(checkInput(start, end)) {
            nodes[start].addEdge(nodes[end], weight);
            nodes[end].addEdge(nodes[start], weight);
        }
    }

    public boolean existsEdge(int start, int end) {
        if(!checkInput(start, end)) return false;
        if(nodes[start].existsEdge(end)) {
            return true;
        }
        return false;
```

```
    }

    public void removeEdge(int start, int end) {
        if(checkInput(start, end)) {
            nodes[start].removeEdge(end);
        }
    }
    //for(Type t : array) - eine Kurzschreibweise, mit der alle Einträge
    // eines Arrays durchlaufen werden, man könnte es auch "for each element ..." nennen.
    public void printGraph() {
        for(Node node : nodes) {
            node.print();
            System.out.println();
        }
    }
}
```

```
public class Node {
    private int number;
    private AdjacencyList adjL;

    //Jeder Knoten kennt seine Nummer und hat eine Adjazenzliste, die am Anfang leer ist.
    public Node(int number) {
        this.number = number;
        adjL = new AdjacencyList();
    }

    //Alle weiteren Methoden "leiten" nur zur Adjazenzliste weiter.
    public void addEdge(Node end, double weight) {
        adjL.append(end, weight);
    }

    public boolean existsEdge(int end) {
        return adjL.existsEdge(end);
    }

    public void removeEdge(int end) {
        adjL.removeEdge(end);
    }

    public int getNumber() {
        return number;
    }

    public void print() {
        System.out.println("This is node number " + number + " and here are my neighbours: ");
        adjL.print();
    }
}
```

```
public class AdjacencyList {
    private ListNode root;

    //Wie oben bereits erwähnt ist dies eine Liste mit null-Referenz am Ende!
    public AdjacencyList() {
```

```
        root = null;
    }

    //Wir fügen vorne an, da dies laufzeittechnisch besser ist und Reihenfolge
    //für uns keine Rolle spielt!
    public void append(Node end, double weight) {
        if(root != null) {
            ListNode tmp = new ListNode(end, weight);
            tmp.setNext(root);
            root = tmp;
        } else {
            root = new ListNode(end, weight);
        }

    }

    //Die folgenden Methoden steuern jeweils wieder eine Rekursion
    public boolean existsEdge(int end) {
        if(root != null) {
            return root.existsEdge(end);
        }
        return false;
    }

    public void removeEdge(int end) {
        if(root != null) {
            if(root.getEnd().getNumber() == end) {
                root = root.getNext();
            } else {
                root.removeEdge(end);
            }
        } else {
            System.out.println("This node has no edges to delete");
        }
    }

    public void print() {
        if(root != null) {
            root.print();
        }
    }
}
```

```
public class ListNode {
    private Node end;
    private double weight;
    private ListNode next;

    //Es würde auch nichts dagegen sprechen verschiedene Gewichte zu speichern,
    //dann müsste allerdings auch die Interaktion mit der Kante anders modelliert werden.
    public ListNode(Node end, double weight) {
        this.end = end;
        this.weight = weight;
        this.next = null;
    }
```

```
    public void setNext(ListNode next) {
        this.next = next;
    }

    public ListNode getNext() {
        return next;
    }

    public Node getEnd() {
        return end;
    }

    //Die folgenden Methoden entsprechen von der Struktur der Rekursion genau unserer
    //ersten LinkedList
    public boolean existsEdge(int end) {
        if(this.end.getNumber() == end) {
            return true;
        }
        if(next == null) {
            return false;
        }
        return next.existsEdge(end);
    }

    public void removeEdge(int end) {
        if(next == null) {
            System.out.println("There is no edge to remove here");
        }
        if(next.getEnd().getNumber() == end) {
            next = next.getNext();
        }
    }

    public void print() {
        System.out.println("Neighbour: " + end.getNumber() + " with weight " + weight);
        if(next != null) {
            next.print();
        }
    }
}
```

4 Graphenalgorithmen