**LK-Q1-V**

1. **Analyse von Graphen in verschiedenen Kontexten**

**(a) Grundlegende Begriffe**

**Beispiel (a): Haus des Nikolaus**

|  |  |
| --- | --- |
|  | Wie viele Möglichkeiten gibt es, das Haus des Nikolaus zu zeichnen?  Das Haus des Nikolaus ist ein Zeichenrätsel. Ziel ist es, das spezielle Haus in einem Linienzug zu zeichnen, ohne eine der Strecken mehrfach zu durchlaufen.  Begleitet wird das Zeichnen durch folgenden gesprochenen Reim:  „Das ist das Haus vom Ni-ko-laus“ |

Mathematisch gesehen handelt es sich um ein Problem der Graphentheorie.

Von einem Knoten aus werde alle Kanten nach einer gewissen Vorschrift durchlaufen.

|  |  |  |  |
| --- | --- | --- | --- |
| 120830_NikoHaus0 | Das Haus des Nikolaus als Prinzipzeichnung | 120830_NikoHaus2 | Die Ecken kann man als Knoten und die Linien als Kanten  eines Graphen interpretieren. |

**Grundlegende Definition**

Ein **ungerichteter** **Graph** besteht aus einer Menge V von Knoten (Vertices) und einer Menge E von Kanten (Edges). Eine Kante ist gegeben durch ein ungeordnetes Paar von Knoten.

Ein **gerichteter** **Graph** besteht aus einer Menge V von Knoten (Vertices) und einer Menge E von gerichteten Kanten (Edges). Eine gerichtete Kante ist gegeben durch ein geordnetes Paar von Knoten.

Haus des Nikolaus

Knotenmenge E = { A, B, C, D, E }

Kantenmenge K = { (A/B), (A/C), (A/E), (B/C), (B/E), (C/D), (C/E), (D/E) }

Bei einem ungerichteten Graphen bezeichnen (A/B) und (B/A) dieselbe Kante.

**Datenstrukturen für Graphen**

**Adjazenzmatrix**

Die einfachste Art der Darstellung eines Graphen ist die Darstellung als **n-x-n**-Matrix. Hierbei steht der Eintrag **aij = 1**, falls es eine Kante von **i** nach **j** gibt, und **aij = 0**, falls keine Kante von **i** nach **j** existiert.

Die folgende Tabelle stellt die Adjazenzmatrix für den Graphen „Haus des Nikolaus“ dar:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| A | 0 | 1 | 1 | 0 | 1 |
| B | 1 | 0 | 1 | 0 | 1 |
| C | 1 | 1 | 0 | 1 | 1 |
| D | 0 | 0 | 1 | 0 | 1 |
| E | 1 | 1 | 1 | 1 | 0 |

Für die Adjazenzmatrix werden unabhängig von der Anzahl der Kanten **n2** Speicherplätze benötigt. In einem ungerichteten Graphen ist die Adjazenzmatrix symmetrisch zur Hauptdiagonalen.

**Aufgabe**

Die folgende Klasse Graph implementiert einen Graphen in Form einer AdjazenzMatrix:

1. Analysieren Sie die Klasse Graph und dokumentieren Sie die Klasse.
2. Erzeugen Sie mit der Klasse Graph das Objekt hausDesNikolaus.
3. Implementieren Sie die Methode istRundweg, die genau dann true liefert, wenn die Knotenfolge, die in dem String-Objekt, das als Parameter übergeben wird, ein Rundweg im Graphen ist. (Es muss nicht geprüft werden, ob alle Knoten besucht wurden).
4. Testen Sie die Methode mit dem Graphen hausDesNikolaus anhand verschiedener Wege.

**public** **class** Graph {

**private** **final** **int** anzahl = 5;

**private** **char**[] knoten = **new** **char**[anzahl];

**private** **boolean**[][] adjazenzMatrix = **new** **boolean**[anzahl][anzahl];

**private** **int** anzahlKnoten = 0;

**public** Graph() {

**for** (**int** i = 0; i<anzahl; i++){

knoten[i] = ' ';

}

**for** (**int** i = 0; i<anzahl; i++){

**for** (**int** j = 0; j < anzahl; j++){

adjazenzMatrix[i][j] = **false**;

}

}

}

**public** **void** knotenHinzu(**char** k){

knoten[anzahlKnoten] = k;

anzahlKnoten++;

}

**public** **int** sucheKnotenIndex(**char** zeichen){

**int** kIndex = -1;

**int** zaehler = 0;

**while** (kIndex == -1 && zaehler < anzahl){

**if** (knoten[zaehler] == zeichen){

kIndex = zaehler;

} **else** {

zaehler++;

}

}

**return** kIndex;

}

**public** **void** kanteHinzu(**char** k1, **char** k2){

**int** indexK1 = sucheKnotenIndex(k1);

**int** indexK2 = sucheKnotenIndex(k2);

**if** (indexK1 != -1 && indexK2 != -1) {

adjazenzMatrix[indexK1][indexK2] = **true**;

adjazenzMatrix[indexK2][indexK1] = **true**;

}

}

**public** **boolean** hatKante(**char** k1, **char** k2){

**int** indexK1 = sucheKnotenIndex(k1);

**int** indexK2 = sucheKnotenIndex(k2);

**if** (indexK1 != -1 && indexK2 != -1) {

**return** adjazenzMatrix[indexK1][indexK2];

} **else** {

**return** **false**;

}

}

**public** **boolean** istRundWeg(String weg){

}

}

**Literatur:**

Vöcking B. u.a. (2008) u.a.. Taschenbuch der Algorithmen, S. 29 Die Eulertour

**LK-Q1 –V:**

1. **Analyse von Graphen in verschiedenen Kontexten**

(b) Aufbau und Darstellung von Graphen anhand von Graphenstrukturen(Adjazenzlisten)

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(a) Erarbeitung der Klassen Graph, Vertex und Edge und beispielhafter Anwendung der Operationen.

**Beispiel (b): Soziale Netzwerke**

120415_SozialesNetzwerk1

Der Begriff „Soziales Netzwerk“ bezeichnet eine soziale Struktur, die zwischen menschlichen Akteuren mittels ihrer Interaktion entsteht.

Soziale Netzwerke lassen sich als Graphen modellieren. Die Akteure bilden die Knoten und Interaktionen werden als Kanten dargestellt.

Für soziale Netzwerke gibt es einige Kenngrößen, die zur Klassifizierung dienen.

**Dichte**

Ein wichtiges Merkmal eines sozialen Netzwerks ist seine Dichte. Sie bezeichnet das Verhältnis der Anzahl der Kanten zur Zahl der möglichen Kanten.

Diese Maßzahl verdeutlicht die Verbundenheit des sozialen Netzwerks.

Sei **n** die Anzahl der Akteure und **m** die Anzahl der Kanten ist die Dichte.

**Zentralitätsgrad**

Ein Akteur ist zentral im Sinne der Degree-Centrality, wenn er direkte Beziehungen zu möglichst vielen Akteuren hat. Sei **d(e)** die Anzahl der Kanten des Knotens **e**, dann ist  die Degree-Centrality von **e**.

Die Maßzahl kennzeichnet die Eingebundenheit eines Akteurs in einem Netzwerk. Je näher der Wert bei 1 liegt, desto höher ist das Prestige des Akteurs.

Soziale Netzwerke sind dynamisch. Es können jederzeit weitere Akteure hinzukommen oder nicht mehr beteiligt sein. Außerdem können die Interaktionspartner wechseln.

Einfaches Beispiel



**Datenstrukturen für Graphen**

**Adjazenzlisten**

In einer Adjazenzliste werden sowohl die Knoten als auch die Nachbarknoten eines Knotens in einem Graphen in linearen Listen verwaltet.



In der vertikal dargestellten Liste werden die Knoten des Graphen dargestellt.

Von jedem Knoten geht eine Liste mit erreichbaren Knoten aus. Ausgangsknoten und erreichbarer Knoten bilden die Kante.

Eine Kante kann auch ein **Gewicht** in Form einer Zahl haben. Durch das Gewicht kann z.B. die Entfernung zwischen zwei Knoten in einem Graphen dargestellt werden. Zu jeder Kante sind zwei Eintragungen in die Adjazenzliste notwendig.

Adjazenzlisten mit Kantenbewertung



Wird ein gewichteter Graph in einer Adjazentmatrix verwaltet, werden die Gewichte anstelle von wahr oder falsch als Werte in die Tabelle eingetragen

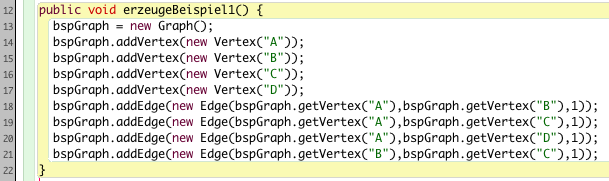
**Graphen im Zentralabitur**

Zur Bearbeitung der Graphaufgaben im Zentralabitur werden drei Klassen zur Verfügung gestellt:

Die Klassen **Graph,** **Vertex** und **Edge**. Die Dokumentationen der drei Klassen finden Sie bei den Materialien zum Zentralabitur im Netz.

Modellierung Graphenklassen

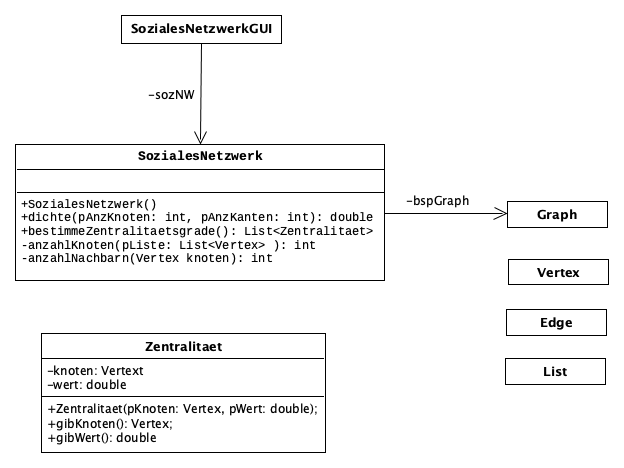
**Erzeugen eines Graphen**



In Zeile 12 wird mithilfe der Abiturklassen ein **Graph** mit Namen **bspGraph** erzeugt. In den Zeilen 13 bis 17 werden Knoten hinzugefügt. In den Zeilen 18 bis 21 werden Kanten des Graphen generiert.

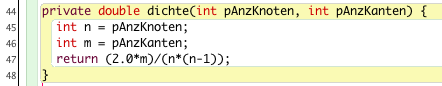
**Aufgabe**

Entwickeln Sie auf Grundlage des folgenden Klassendiagramms ein Java-Programm, mit dem sich die Dichte und die Zentralitätsgrade eines sozialen Netzwerkes, das in einem Graphen gespeichert ist, berechnen und anzeigen lassen.

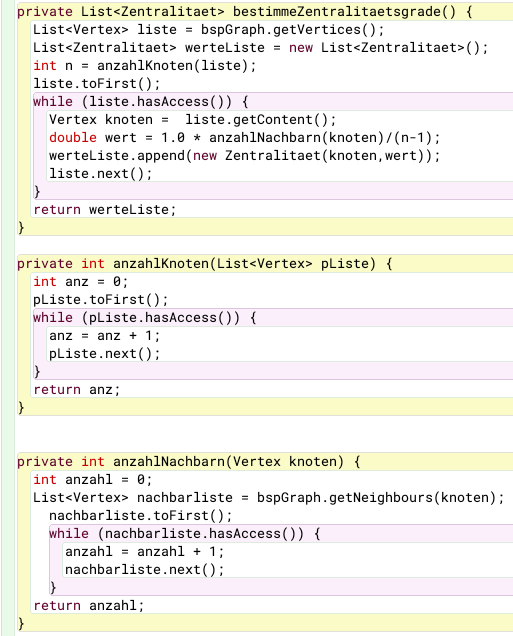


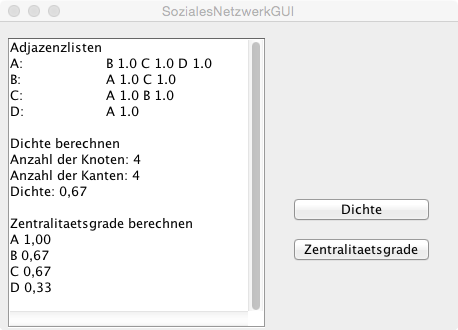
Lösung:

Berechnung der Dichte



Berechnung der Zentralitätsgrade





**LK-Q1-V**

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(c) Bestimmung von Wegen in Graphen im Anwendungskontext (Tiefensuche, Breitensuche)



**Wegsuche im Graphen**

**Aufgabe**

*Entwickeln Sie einen Algorithmus, der einen beliebigen Weg von einem Startknoten S zu einem Zielknoten Z in einem gegebenen Graphen bestimmt. Jeder im Weg enthaltene Knoten darf nur einmal besucht werden, d.h. der Weg darf keine Kreise (Zyklen) haben.*

Lösung:

Es gibt mehrere Algorithmen, um dieses Problem zu lösen:

1. **Tiefensuche - Backtracking**

Ein Backtracking-Algorithmus geht nach dem Versuch-und-Irrtum-Prinzip (*trial and error*) vor. Es wird versucht, eine erreichte Teillösung schrittweise zu einer Gesamtlösung auszubauen. Wenn absehbar ist, dass eine Teillösung nicht zu einer endgültigen Lösung führen kann, wird der letzte Schritt beziehungsweise werden die letzten Schritte zurückgenommen und es werden stattdessen alternative Wege probiert. Auf diese Weise ist sichergestellt, dass alle in Frage kommenden Lösungswege ausprobiert werden können. Mit Backtracking-Algorithmen wird eine vorhandene Lösung entweder gefunden (unter Umständen nach sehr langer Laufzeit), oder es kann definitiv ausgesagt werden, dass keine Lösung existiert. Backtracking wird am einfachsten rekursiv implementiert und ist ein prototypischer Anwendungsfall von Rekursion. Selbstverständlich kann ein Backtracking-Algorithmus mit Hilfe eines Stacks auch iterativ implementiert werden (Übung!).

Ein Beispiel für Backtracking ist die Tiefensuche im Graphen. Bei diesem Verfahren wird versucht den Zielknoten auf möglichst direktem Wege zu finden und sich den zurückgelegten Weg dabei zu merken. Man geht vom Startknoten zum ersten Nachbarknoten, von diesem zu dessen erstem Nachbarknoten, der noch nicht besucht wurde, usw. bis man den Zielknoten erreicht hat. Wenn man in einer Sackgasse landet, d.h. bei einem Knoten der nur Nachbarn hat, die schon besucht wurden, geht man einen oder bei Bedarf auch mehrere Schritte zurück (Backtracking) und geht dann zum nächsten noch nicht besuchten Nachbarknoten usw. Dieses Verfahren wird fortgeführt bis man den Zielknoten erreicht hat, oder kein Schritt mehr möglich ist.

**Beispiel:**

Graph: s.o. Startknoten I, Zielknoten L

Die Nachbarknoten sollen jeweils in alphabetischer Reihenfolge besucht werden:

|  |  |  |
| --- | --- | --- |
| Schritt | Knotenfolge | Kommentar |
| 1 | I |  |
| 2 | IG |  |
| 3 | IGA |  |
| 4 | IGAB |  |
| 5 | IGABC |  |
| 6 | IGABCE |  |
| 7 | IGABCEF |  |
| 8 | IGABCEFK | Sackgasse zurück |
| 9 | IGABCEF | Sackgasse zurück |
| 10 | IGABCE |  |
| 11 | IGABCEL |  |

**Umgangssprachlich formulierter formalisierter Algorithmus**

**SucheWeg** von:start nach:ziel

Markiere start als besucht

start in die Wegliste

Wenn ziel noch nicht erreicht

Solange Weg nicht gefunden und start hat Nachbarknoten

Nächsten knoten aus Liste der Nachbarknoten holen

wenn knoten nicht markiert

**SucheWeg** von:knoten nach:ziel //rekursiver Aufruf knoten wird start

Wenn Weg noch nicht gefunden

gehe letzten Schritt zurück //Backtracking

sonst

ziel in Wegliste einfügen

Mit dem rekursiven Aufruf wird der Nachbarknoten zum neuen Start-Knoten, mit dem die Wegsuche fortgesetzt wird.

Dieser Algorithmus in Java implementiert ergibt folgende Methode:

public boolean sucheWeg(Vertex knoten, Vertex ziel, List<Vertex> weg){

boolean wegGefunden = false;

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (!wegGefunden && nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

wegGefunden = sucheWeg(nachbar, ziel, weg);

if (!wegGefunden) {

//Zug rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

weg.remove();

}

}

nachbarn.next();

}

} else {

wegGefunden = true;

}

return wegGefunden;

}

Die Methode liefert als Rückgabewert true, wenn ein Weg gefunden wurde, und false, wenn kein Weg gefunden wurde. Die Knotenfolge des Weges wird in der Liste verwaltet, die als Parameter übergeben wird. Der Aufruf der Methode muss mit einer leeren Liste als aktueller Parameter erfolgen. Da bei Objekten als Parameter in Java grundsätzlich nur eine Referenz auf das Objekt übergeben wird, enthält die Liste nach Beendigung der Methode die Knotenfolge des gefundenen Weges.

**Aufgabe**

*Analysieren Sie die Java-Methode mit dem Startknoten F und dem Zielknoten L und geben Sie den gefundenen Weg als Knotenfolge an.*

Durch wenige Modifikationen der Java-Implementation lassen sich **alle** Wege vom Start- zum Zielknoten finden.

public void sucheAlleWege(Vertex knoten, Vertex ziel, List<Vertex> weg,

List<List<Vertex>> alleWege){

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

sucheAlleWege(nachbar, ziel, weg, alleWege);

//Zug rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

weg.remove();

}

nachbarn.next();

}

} else {

List<Vertex> einWeg = new List<Vertex>();

copy(weg, einWeg); //Hilfsmethode

alleWege.append(einWeg);

}

}

**Aufgabe**

*Vergleichen Sie die Java-Methode sucheWeg mit der Methode sucheAlleWege und beschreiben Sie die Unterschiede.*

*Begründen Sie, warum sich mit sucheAlleWege alle Wege im Graphen vom Start- zum Zielknoten erzeugen lassen.*

*Erläutern Sie, wie diese Wege in der Methode verwaltet werden.*

**2. Breitensuche**

Der Algorithmus zur Breitensuche vermeidet Sackgassen. Vom Startknoten ausgehend wird geprüft, ob einer seiner Nachbarknoten der Zielknoten ist. Wenn das nicht der Fall ist, werden nacheinander die Nachbarknoten der Nachbarknoten, die bisher nicht besucht wurden, betrachtet usw. bis der Zielknoten gefunden wurde. In einer Liste werden alle Knoten entsprechend der Besuchsreihenfolge gespeichert, bis der Zielknoten gefunden ist.

**Beispiel:**

Wenn von I aus der Zielknoten K gesucht wird, werden, wie man der folgenden Abbildung entnehmen kann, die Knoten in folgender Reihenfolge betrachtet: I GHK AF E B C L



10

9

8

7

6

5

4

3

2

1

Um den Algorithmus zu implementieren, werden die Nachbarknoten eines Knotens, die nicht markiert sind, an eine Schlange (Queue) angefügt, die von vorne nach hinten abgearbeitet wird. Der Algorithmus ist beendet, wenn der Zielknoten aus der Schlange geholt wurde oder wenn die Schlange leer ist. Im zweiten Fall wurde der Zielknoten nicht gefunden.

Die folgende Java-Methode implementiert die Breitensuche in einem Graphen:

public boolean breitenSuche(Vertex knoten, Vertex ziel,

List<Vertex> besucht){

Queue<Vertex> schlange = new Queue<Vertex>();

schlange.enqueue(knoten);

knoten.setMark(true);

boolean gefunden = false;

while (!schlange.isEmpty() && !gefunden){

knoten = schlange.front();

besucht.append(knoten);

schlange.dequeue();

if (knoten != ziel) {

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()){

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()) {

schlange.enqueue(nachbar);

nachbar.setMark(true);

}

nachbarn.next();

}

} else {

gefunden = true;

}

}

return gefunden;

}

**Aufgabe**

*Analysieren Sie die Java-Methode zur Breitensuch mit dem Startknoten E und dem Zielknoten H und geben Sie den Inhalt der Schlange nach jedem Durchlauf der äußeren while-Schleife an.*

Die Liste der besuchten Knoten enthält im Gegensatz zur Tiefensuche keinen Weg vom Start- zum Zielknoten. Bei der Breitensuche von I nach L enthält die Liste die Knotenfolge IGHKAFEBCL. Da es z. B. keine Kante zwischen H und K gibt, kann es sich nicht um einen Weg von I nach L handeln. Dennoch lässt sich aus dieser Liste ein Weg konstruieren, weil alle Knoten eines Weges von I nach L in der Liste enthalten sein müssen, sonst wäre L nicht erreicht worden. Die Strategie besteht darin, vom Zielknoten L aus rückwärts den Weg zu suchen. Wenn man die Liste von hinten nach vorne durchläuft, ist E der erste Knoten, der eine Kante mit H hat, also Vorgängerknoten von L ist. Der erste Vorgänger von E ist F, Vorgänger von F ist K, erster Vorgänger von K ist G und von G I, sodass sich der Weg IGKFEL konstruieren lässt.

In der folgenden Java-Methode ist der beschriebene Algorithmus mithilfe eines Stacks implementiert. Als Parameter wird die Liste der besuchten Knoten an die Methode übergeben.

public List<Vertex> konstruiereWeg(List<Vertex> knotenListe){

Stack<Vertex> stack = new Stack<Vertex>();

knotenListe.toFirst();

while (knotenListe.hasAccess()){

stack.push(knotenListe.getContent());

knotenListe.next();

}

List<Vertex> weg = new List<Vertex>();

Vertex knoten1 = stack.top();

weg.insert(knoten1);

weg.toFirst();

stack.pop();

while (!stack.isEmpty()){

Vertex knoten2 = stack.top();

stack.pop();

Edge edge = bspGraph.getEdge(knoten1, knoten2);

if (edge != null){

weg.insert(knoten2);

weg.toFirst();

knoten1 = knoten2;

}

}

return weg;

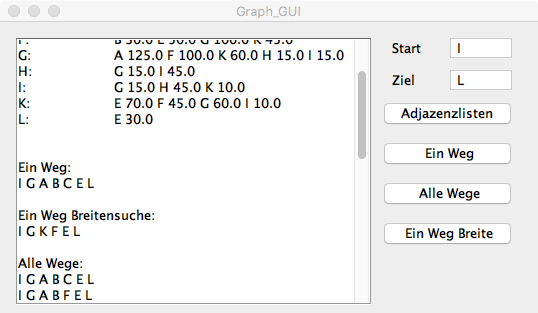
}

**Aufgabe**

*Analysieren und erläutern Sie die Java-Methode* konstruiereWeg *mit der durch Breitensuche von I nach L generierten Liste der besuchten Knoten.*

*Geben Sie den Weg an, den die Methode zurückliefert.*

Alle Algorithmen sind in dem BlueJ-Projekt **WegSuche** implementiert und können getestet werden.



**LK-Q1-V**

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(d) Bestimmung kürzester Wege in Graphen im Anwendungskontext (Backtracking, Dijkstra-Algorithmus)



**Aufgabe**

*Entwickeln Sie einen Algorithmus, der den kürzesten Weg von einem Startknoten S zu einem Zielknoten Z in einem gegebenen Graphen bestimmt.*

1. Backtracking

Anknüpfend an den Backtracking-Algorithmus zur Suche aller Weg in einem Graphen, findet folgender Algorithmus sicher den kürzesten Weg:

Bestimme alle Wege von S nach Z

Bestimme die Länge dieser Wege

Suche den Weg mit der kleinsten Länge

Etwas schneller funktioniert es, wenn der Algorithmus zur Bestimmung aller Wege die Weglaenge mitberechnet und sich den bis dahin kürzesten Weg merkt. In der folgenden Java-Methode sind die Unterschiede im Quelltext zwischen den Methoden sucheAlleWege und sucheKuerzestenWeg grau schraffiert.

public double sucheKuerzestenWeg(Vertex knoten, Vertex ziel, List<Vertex> weg,

List<Vertex> kurzWeg, double wegLaenge, double minWegLaenge){

knoten.setMark(true);

weg.append(knoten);

if (knoten != ziel){

List<Vertex> nachbarn = bspGraph.getNeighbours(knoten);

nachbarn.toFirst();

while (nachbarn.hasAccess()) {

Vertex nachbar = nachbarn.getContent();

if (!nachbar.isMarked()){

double distanz = bspGraph.getEdge(knoten, nachbar).getWeight();

wegLaenge = wegLaenge + distanz;

minWegLaenge = sucheKuerzestenWeg(nachbar, ziel, weg, kurzWeg,

wegLaenge, minWegLaenge);

//Schritt rückgängig machen

weg.toLast();

Vertex loeschKnoten = weg.getContent();

loeschKnoten.setMark(false);

wegLaenge = wegLaenge - distanz;

weg.remove();

}

nachbarn.next();

}

} else {

if (wegLaenge < minWegLaenge){

minWegLaenge = wegLaenge;

copy(weg, kurzWeg);

}

}

return minWegLaenge;

}

**Aufgabe**

*Analysieren Sie die Methode* sucheKuerzestenWeg *mit dem Startknoten I und dem Zielknoten L. Geben Sie die Inhalte der Liste* kurzWeg *sowie die zugehörigen Werte von* minWegLaenge *an, die der Algorithmus generiert.*

Lösung:

|  |  |
| --- | --- |
| kurzWeg | minWegLaenge |
| I G A B C E L | 260 |
| I G F B C E L | 245 |
| I G F E L | 195 |
| I G K E L | 175 |
| I K E L | 110 |

**Aufgabe**

*a) Bestimmen Sie die Anzahl der Wege von einem beliebigen Start- zu einem beliebigen Zielknoten in Graphen mit 2, 3, 4, 5 Knoten, wenn alle Knoten durch Kanten paarweise miteinander verbunden sind.*

*b) Leiten Sie eine Formel für n Knoten her.*

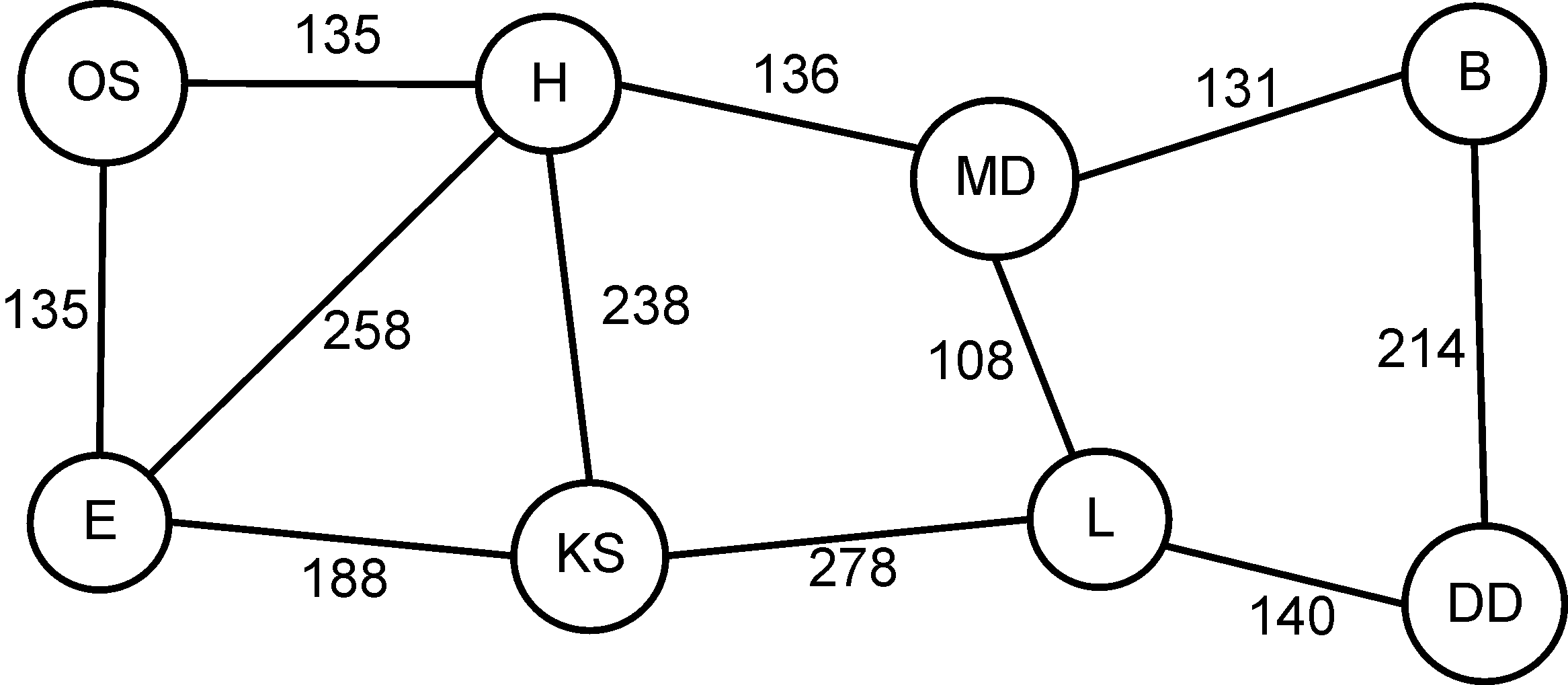
*c) Begründen Sie, warum das Backtracking-Verfahren zur Bestimmung des kürzesten Weges zwischen zwei Knoten in einem Graphen für Navigationssysteme nicht geeignet ist.*

**2. Dijkstra-Algorithmus**

Einen weitaus schnelleren Algorithmus hat Edsger W. Dijkstra entwickelt. Mit folgender Aufgabe wird der Algorithmus erarbeitet.

**Aufgabe**

Ein reduziertes Straßen-/Autobahnnetz zwischen acht deutschen Großstädten lässt sich vereinfacht durch folgenden Graphen darstellen.



|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| B | DD | E | H | KS | L | MD | OS |
| Berlin | Dresden | Essen | Hannover | Kassel | Leipzig | Magdeburg | Osnabrück |

1. *Entwickeln Sie für obigen Graphen die Adjazenzmatrix. Hinweis: Um die Indizes der einzelnen Städte zu ermitteln, überführen Sie diese einfach in alphabetischer Reihenfolge in Zahlen:*

*B → 1, DD → 2, . . .*

1. Von einem Logistikzentrum in Leipzig sollen Routen zu allen anderen Städten ermittelt werden. Führen Sie deshalb dort beginnend eine vereinfachte Breitensuche durch. Die Vereinfachung besteht darin, dass keine Stadt doppelt in der Schlange der noch nicht besuchten Knoten vorkommen soll.

*(1) Ermitteln Sie die Schlange der noch nicht besuchten Städte.*

*(2) Geben Sie an den Städten im Graphen die Besuchsreihenfolge an und markieren Sie die gewählten Straßen farbig.*

*(3) Stellen Sie in der folgenden Tabelle die ermittelte Entfernung jeder Stadt entlang der markierten Wege nach Leipzig dar.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ort | B | DD | EE | H | KS | L | MD | OS |
| Entfernung nach L |  |  |  |  |  |  |  |  |

1. *Beurteilen Sie, ob die Entfernungen von Leipzig zu allen anderen Städten minimal sind.*
2. Zur Berechnung kürzester Wege ist die Auswahl der zu besuchenden Städte entscheidend.

*a) Erläutern Sie, wie sich bei der Breitensuche die Reihenfolge der zu besuchenden Städte in der Schlange ergibt.*

*b)   Entwickeln Sie ein Verfahren für die Auswahl der nächsten Stadt bei der Breitensuche, um einen möglichst kurzen Weg zu finden.*

*c)   Ermitteln Sie die Wege und deren Längen, die sich bei Anwendung des Verfahrens von b) ergeben.*

1. Um nach Durchführung des Verfahrens die Wege zu allen Städten angeben zu können, speichert man auch bei diesem Verfahren für jede Stadt jeweils den direkten Vorgänger auf dem Weg dorthin.

*Stellen Sie in der folgenden Tabelle die Vorgänger jeder Stadt auf dem kürzesten Weg von Leipzig dar.*

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Ort | B | DD | EE | H | KS | L | MD | OS |
| Vorgänger |  |  |  |  |  |  |  |  |

**Der Dijkstra-Algorithmus**

In Wikipedia wird der Dijkstra-Algorithmus wie folgt beschrieben:

„Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht. Andere Kanten werden erst dann verfolgt, wenn alle kürzeren Streckenabschnitte beachtet wurden. Dieses Vorgehen gewährleistet, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existieren kann. Eine einmal berechnete Distanz zwischen dem Startknoten und einem besuchten Knoten wird nicht mehr geändert. Distanzen zu noch nicht abgearbeiteten Knoten können sich hingegen im Laufe des Algorithmus durchaus verändern, nämlich verringern. Dieses Vorgehen wird fortgesetzt, bis die Distanz des Zielknotens berechnet wurde (*single-pair shortest path*) oder die Distanzen aller Knoten zum Startknoten bekannt sind (*single-source shortest path*).“[[1]](#footnote-1)

Wie der Beschreibung zu entnehmen ist, benötigt jeder Knoten die Distanz zum Startknoten als weiteres Attribut. Um nach Abarbeitung des Algorithmus’ den Weg wie bei der Breitensuche vom Ziel zum Startknoten generieren zu können, erhält jedes Knotenobjekt als weiteres Attribut einen Verweis auf seinen Vorgänger. Der Graph enthält zur Anwendung des Dijkastra-Algorithmus’ Knoten-Objekte folgender Klasse.

public class DijkstraVertex extends Vertex {

private double distanz;

private DijkstraVertex vorgaenger;

public DijkstraVertex(String pID, int xPos, int yPos){

super(pID, xPos, yPos);

vorgaenger = null;

distanz = -1;

}

//getter und setter

}

Der Algorithmus soll am Beispiel der Wegsuche von I nach L im Beispielgraphen erläutert werden.

Die Prioritätenschlange enthält die noch nicht bearbeiteten Nachbarknoten von bereits bearbeiteten (markierten) Knoten aufsteigend nach ihrer Entfernung zum Startknoten. Die bereits bearbeiteten Konten sind grau markiert.



0. Initialisierung

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1,-- | G, -1, -- | H, -1, -- | I, -1, -- | K, -1, -- | L,-1,-- |

prioritaetenSchlange

|  |
| --- |
|  |

1. Schritt: Startknoten in Schlange einfuegen

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1, -- | G, -1, -- | H, -1, -- | I, 0, -- | K, -1, -- | L, -1, -- |

Prioritätenschlange

|  |
| --- |
| I, 0, -- |

2. Schritt: I aus Schlange entfernen, Nachbarknoten nach aufsteigender Entfernung zum Startknoten in Schlange einfügen

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, -1, -- | F, -1, -- | G, 15, I | H, 45, I | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| K, 10, I | G, 15, I | H, 45, I |

3. Schritt: Ersten Knoten K aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, -1, -- | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 45, I | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| G, 15, I | H,45,I | F, 56, K | E, 80, K |

4. Schritt: Ersten Knoten G aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140,G | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0, -- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| H, 30, G | F, 56, K | E, 80, K | A, 140, G |

5. Schritt: Ersten Knoten H aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, -1, -- | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| F, 56, K | E, 80, K | A, 140, G |

6. Schritt: Ersten Knoten F aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, 86, F | C, -1, -- | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, -1, -- |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| E, 80, K | B, 86, F | A, 140, G |

7. Schritt: Ersten Knoten E aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 140, G | B, 86, F | C, 120, E | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |  |  |
| --- | --- | --- | --- |
| B, 86, F | L, 110, E | C, 120, E | A, 140, G |

8. Schritt: Ersten Knoten B aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |  |
| --- | --- | --- |
| A, 106, B | L, 110, E | C, 116, B |

9. Schritt: Ersten Knoten A aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |  |
| --- | --- |
| L, 110, E | C, 116, B |

10. Schritt: Ersten Knoten L aus Schlange entfernen. Entfernungen in Prioritätenschlange ggf. nach unten korrigieren, wenn erstes Element Nachbarknoten ist. Andere Nachbarknoten mit berechneter Entfernung zum Startknoten in Schlange aufsteigend einfügen.

DijkstraKnoten(ID, distanz, vorgaenger) des Graphen

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A, 106, B | B, 86, F | C, 116, B | E, 80, K | F, 56, K | G, 15, I | H, 30, G | I, 0,-- | K, 10, I | L, 110, E |

Prioritätenschlange

|  |
| --- |
| C, 116, B |

Da der Zielknoten L erreicht wurde, kann der Algorithmus abgebrochen werden. Die kürzeste Entfernung vom Start- zum Zielknoten ist als distanz im Knoten L gespeichert, hat also den Wert 110. Der Weg lässt sich ausgehend vom Zielknoten über die Verweise auf die Vorgängerknoten erzeugen.

Als kürzester Weg in umgekehrter Reihenfolge ergibt sich damit die Knotenfolge; L -> E -> K -> I

Bricht man den Algorithmus nicht ab, wenn der Zielknoten erreicht ist, sondern erst, wenn die Prioritätenschlange leer ist, enthalten alle Dijkstra-Knoten des Graphen die kürzesten Entfernungen zum Startknoten, wenn es einen Weg gibt.

**Aufgabe**

*Ermitteln Sie den kürzesten Weg im Beispielgraphen von A nach H mithilfe des Dijkstra-Algorithmus’.*

**Java Methode für den Dijkstra-Algorithmus**

private void dijkstra(DijkstraVertex pStart, DijkstraVertex pZiel){

if (pStart!= null && pZiel != null){

PrioritaetsSchlange schlange = new PrioritaetsSchlange();

pStart.setzeDistanz(0);

schlange.fuegeEin(pStart);

DijkstraVertex nachbarKnoten = null;

DijkstraVertex ersterKnoten = null;

while (!schlange.isEmpty() && ersterKnoten != pZiel) {

schlange.toFirst();

ersterKnoten = (DijkstraVertex)schlange.getObject();

schlange.remove();

if (!ersterKnoten.isMarked()){

ersterKnoten.setMark(true);

List<Vertex> nachbarListe = bspGraph.getNeighbours(ersterKnoten);

nachbarListe.toFirst();

while (nachbarListe.hasAccess()){

nachbarKnoten = (DijkstraVertex) nachbarListe.getContent();

Edge kante = bspGraph.getEdge(nachbarKnoten, ersterKnoten);

double gewicht = kante.getWeight();

//Nachbarknoten in Schlage einfügen oder – falls vorhanden –

//Entfernung ggf. nach unten korrigieren

if (nachbarKnoten.gibDistanz() == -1 ||

nachbarKnoten.gibDistanz()>ersterKnoten.gibDistanz()+gewicht){

nachbarKnoten.setzeDistanz(ersterKnoten.gibDistanz()+gewicht);

nachbarKnoten.setzeVorgaenger(ersterKnoten);

schlange.fuegeEin(nachbarKnoten);

//korrigierter Knoten wird überschrieben

}

nachbarListe.next();

}

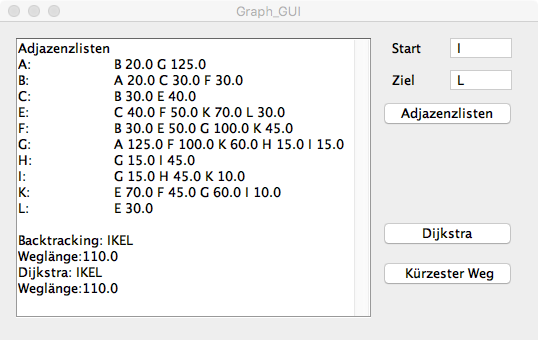
}

}

}

}

Vollständig implementiert sind die beiden Algorithmen zur Suche des kürzesten Weges in einem Graphen im BlueJ-Projekt KuerzesterWeg. Aus Übersichtlichkeitsgründen wurden nicht alle möglichen Fehler abgefangen.



**Aufwand**

Die Laufzeit des Dijkstra-Algorithmus’ ist proportional (n+m)\*log(n), wenn n die Anzahl der Knoten und m die Anzahl der Kanten im Graphen ist, wächst also bei zunehmender Komplexität des Graphen nur sehr langsam an. Daher kann der Dijkstra-Algorithmus zur Berechnung von Weglängen in Navigationssystemen sehr effektiv genutzt werden.

**LK-Q1-V**

**2. Die Datenstruktur Graph im Anwendungskontext unter Nutzung der Klassen Graph, Vertex und Edge.**

(d) Minimale Spannbäume

120502_Spbaum1.emf

**Aufgabe**

*Gegeben sind Verbraucher, die an ein Versorgungsnetz angeschlossen werden sollen. Die Kosten für die Leitung von jedem Punkt zum anderen sind bekannt.*

*Es wird ein Verbindungsnetz mit minimalen Kosten gesucht.*

Lösung:

Man kann sich leicht überlegen, dass ein minimales Versorgungsnetz ein Teilgraph des Graphen ist, der aus einer Menge von Kanten besteht, die alle Knoten miteinander verbinden. Der Teilgraph darf keine Rundwege (Zyklen) enthalten. Denn gäbe es einen geschlossenen Zyklus, könnte man ein billigeres Versorgungsnetz durch Entfernen einer Kante aus diesem Zyklus konstruieren.

**Definitionen**

Ein ungerichteter Graph heißt **zusammenhängend**, wenn es zwischen zwei Knoten mindestens einen Weg gibt.   
Ein ungerichteter Graph heißt **Baum**, falls er zusammenhängend und zyklenfrei ist.

Ein Teilgraph eines ungerichteten Graphen G heißt Spannbaum von G, wenn H ein Baum auf den Knoten von G ist.

Ein Spannbaum eines gewichteten ungerichteten Graphen G heißt minimaler Spannbaum von G, wenn S minimales Gewicht unter allen Spannbäumen von G besitzt.

**Algorithmen zur Bestimmung eines minimalen Spannbaums**

**1. Der Kruskal-Algorithmus**

Generiere nach Gewicht aufsteigend eine sortierte Liste L der Kanten des Graphen

Solange L nicht leer ist

wähle erste Kante e aus L

entferne e aus L

markiere e

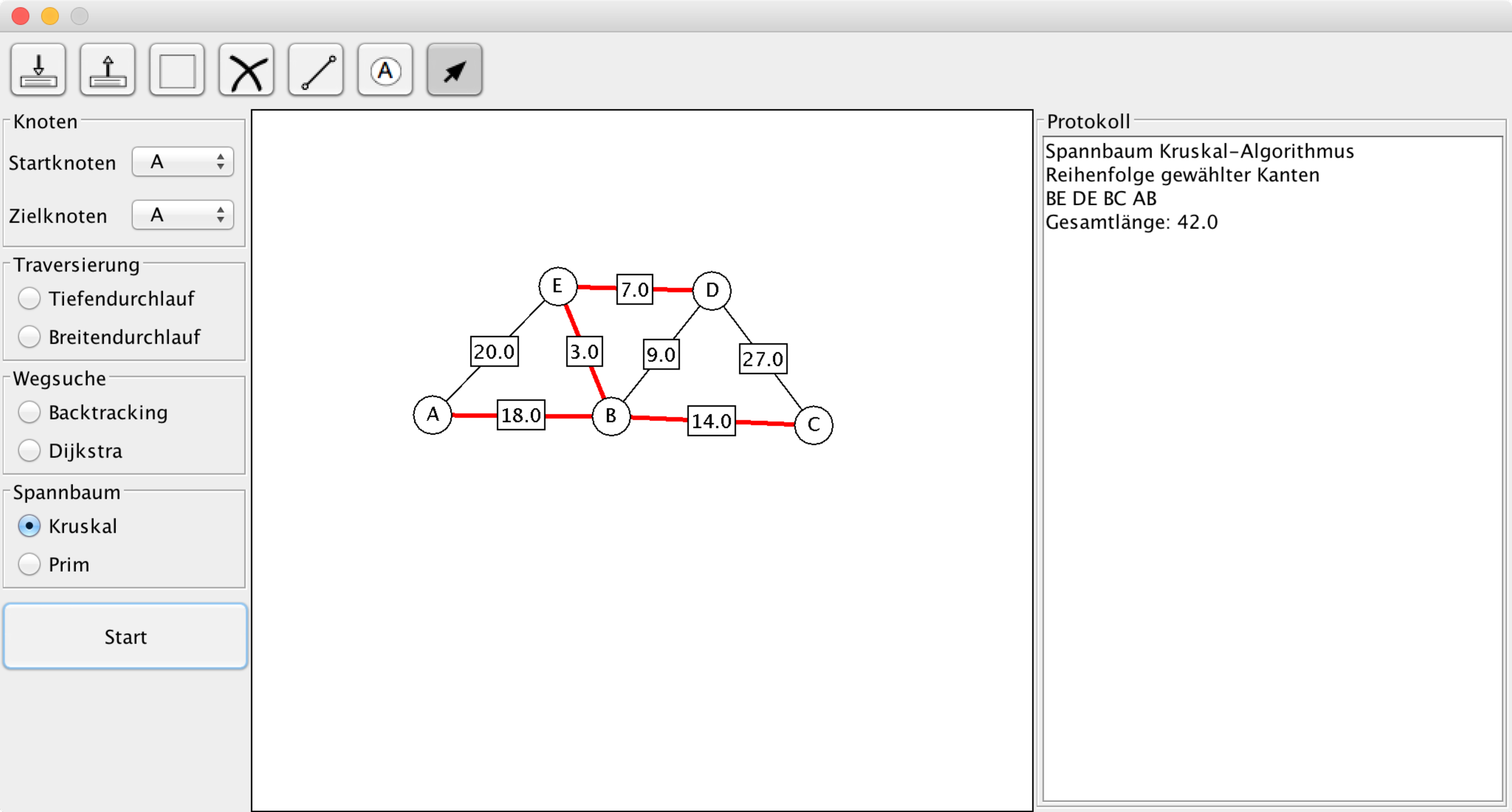
wenn der Graph aus den markierten Kanten und deren Endknoten einen Zyklus enthält

lösche Markierung von e

Der Graph aus den markierten Kanten und deren Endknoten ist ein minimaler Spannbaum

**Der Kruskal-Algorithmus schrittweise**

|  |
| --- |
| kruskal01 |
| kruskal02 |
| kruskal03 |
| kruskal04 |
| kruskal05 |
| kruskal06 |
| kruskal07 |
| kruskal08 |
| kruskal09 |

Mit dem Java-Programm GraphTool lassen sich interaktiv Graphen erzeugen und verschiedene Graphenalgorithmen anwenden und veranschaulichen. 

**Quelltext Kruskal-Algorithmus**

public String spannbaumKruskal(){

bspGraph.setAllVertexMarks(false);

bspGraph.setAllEdgeMarks(false);

List<Edge> kantenListe =this.gibAufsteigendSortiert(bspGraph.getEdges());

String kantenString = "";

kantenListe.toFirst();

while (!kantenListe.isEmpty()){

Edge aktuelleKante = kantenListe.getContent();

Vertex[] knoten = (aktuelleKante.getVertices());

aktuelleKante.setMark(true);

resetStatus(bspGraph.getVertices());

if (hatZyklus((Vertex1)knoten[0], null)){

aktuelleKante.setMark(false);

} else {

kantenString = kantenString + knoten[0].getID()+" "+

knoten[1].getID()+":"+aktuelleKante.getWeight()+"\n";

}

kantenListe.remove();

}

return kantenString;

}

Die Implementation entspricht dem umgangssprachlich formulierten Algorithmus. Die Methode liefert einen String mit den Kanten und ihren Längen zurück, die zum minimalen Spannbaum gehören. bspGraph ist eine Referenz auf den Graphen, von dem ein minimaler Spannbaum erzeugt werden soll. Die Variable ist als Attribut der Klasse Spannbaum, die spannbaumKruskal als Methode enthält, deklariert. Die Methode verwendet drei Hilfsmethoden:

1. gibAufsteigendSortiert erzeugt die sortierte Kantenliste
2. hatZyklus prüft, ob die markierten Kanten einen Kreis enthalten
3. resetStatus setzt den Status aller Knoten auf 0.

Vertex1 ist eine Unterklasse der Klasse Vertex, die das Attribut status vom Typ int enthält, das für die Prüfung, ob ein Zyklus vorliegt, benötigt wird.

**2. Der Prim-Algorithmus**

Ein zweiter Algorithmus zur Bestimmung minimaler Spannbäume ist der Algorithmus von Prim.

**Umgangssprachliche Formulierung**

Lösche alle Knotenmarkierungen

Wähle einen Knoten aus und markiere ihn

Führe den folgenden Schritt so oft aus, bis alle Knoten markiert sind:

Suche die Kante mit dem geringsten Gewicht zwischen einem Knoten aus der Menge der markierten Knoten und einem Knoten aus der Menge der nicht markierten Knoten.

Markiere den nicht markierten Knoten.

**Der Prim-Algorithmus schrittweise**

|  |  |
| --- | --- |
| prim01 | prim02 |
| prim03 | prim04 |
| prim05 |  |

**Quelltext Prim-Algorithmus**

public String spannbaumPrim(Vertex startKnoten){

bspGraph.setAllVertexMarks(false);

startKnoten.setMark(true);

String kantenString = "";

List<Edge> kanten = bspGraph.getEdges(startKnoten);

while (!bspGraph.allVerticesMarked()){

Edge minKante = null;

kanten.toFirst();

Vertex[] endKnoten;

while (kanten.hasAccess()){

Edge kante = kanten.getContent();

endKnoten = kante.getVertices();

if (!endKnoten[0].isMarked() || !endKnoten[1].isMarked()){

if (minKante == null || kante.getWeight() < minKante.getWeight()){

minKante = kante;

}

}

kanten.next();

}

minKante.setMark(true);

endKnoten = minKante.getVertices();

kantenString = kantenString+endKnoten[0].getID()+" "

+endKnoten[1].getID() +" \n";

Vertex neuerKnoten = null;

if (endKnoten[0].isMarked()){

neuerKnoten = endKnoten[1];

} else {

neuerKnoten = endKnoten[0];

}

//Neu verbundenen Knoten markieren.

neuerKnoten.setMark(true);

//Mit dem neuen Knoten inzidente Kanten der Kantenliste hinzufuegen.

List<Edge> neueKanten = bspGraph.getEdges(neuerKnoten);

neueKanten.toFirst();

while (neueKanten.hasAccess()){

kanten.append(neueKanten.getContent());

neueKanten.next();

}

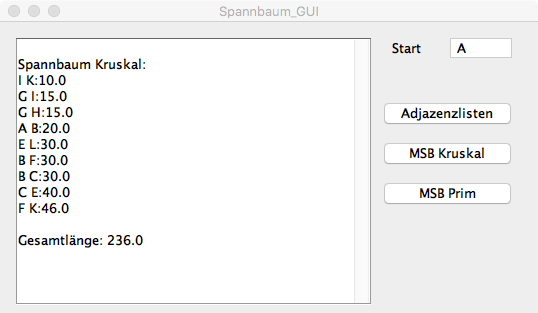
}

return kantenString;

}

Die Implementation entspricht dem umgangssprachlich formulierten Algorithmus. Die Methode liefert einen String mit den Kanten und ihren Längen zurück, die zum minimalen Spannbaum gehören. Die markierten Kanten bilden mit ihren Endknoten den minimalen Spannbaum.

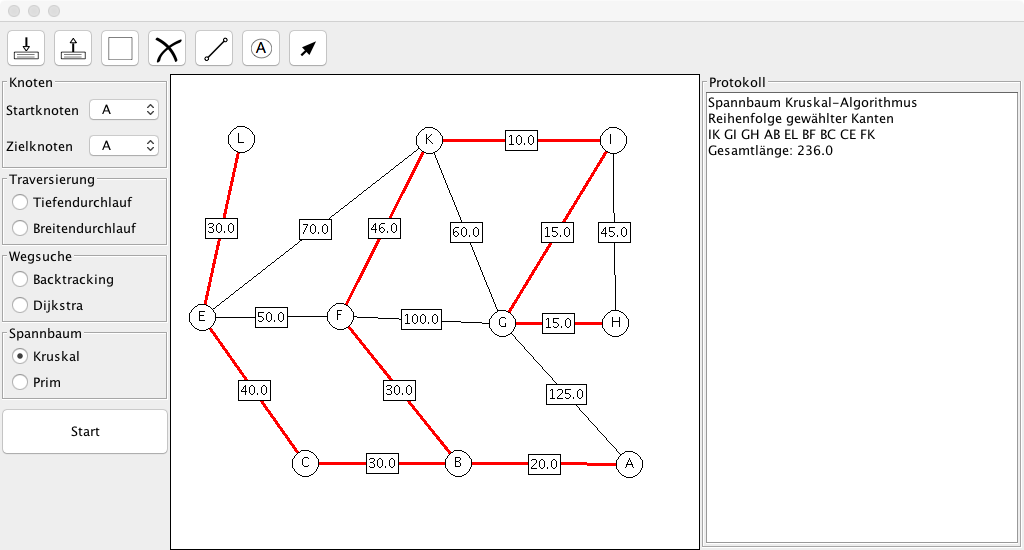
Die vollständige Implementation ist im BlueJ –Projekt Spannbaum enthalten. Um die Quelltexte übersichtlich zu halten, wurden nicht alle möglichen Fehler abgefangen.



**Vergleich Kruskal-Prim**

Ein Graph kann mehrere minimale Spannbäume haben. Diese haben aber immer dieselbe minimale Länge. Der Algorithmus von Prim vermeidet das Sortieren der Kanten, das beim Algorithmus von Kruskal erforderlich ist. Allerdings ist die Auswahl der Kanten beim Algorithmus von Prim komplizierter als beim Algorithmus von Kruskal.

**Aufgabe**

1. *Ermitteln Sie mithilfe des Kruskal-Algorithmus’ einen minimalen Spannbaum für den nebenstehenden Graphen*
2. *Ermitteln Sie mithilfe des Prim-Algorithmus’ einen minimalen Spannbaum mit dem Startknoten A für den nebenstehenden Graphen.*

**LK-Q1-V**

**3. Die Datenstruktur Binärbaum als Spezialfall eines Graphen im Anwendungskontext unter Nutzung der Klasse BinaryTree<ContentType>**

1. Definition eines Binärbaumes und grundlegende Begriffe
2. Erarbeitung der Klasse BinaryTree und beispielhafte Anwendung der Operationen

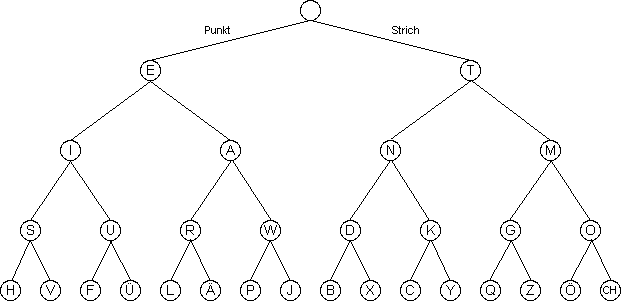
**Aufgabe**

*Modellieren und implementieren Sie ein Java-Programm, das in der Lage ist, eine Zeichenfolge aus Großbuchstaben in das Morsealphabet zu codieren und eine Folge von Morsezeichen, die durch „/“ getrennt sind, zu dekodieren.*

**Das Morsealphabet**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| A ·- | F ··-· | K -·- | P ·--· | U ··- | Z --·- |
| B -··· | G --· | L ·-·· | Q --·· | V ···- | Ä ·-·- |
| C -·-· | H ···· | M -- | R ·-· | W ·-- | Ö ---· |
| D -·· | I ·· | N -· | S ... | X -··- | Ü ··-- |
| E · | J ·--- | O --- | T - | Y -·-- | CH ---- |

Übersichtlich lässt sich der Morsecode in folgender Struktur darstellen.



Bei dieser Struktur handelt es sich um einen Baum (zusammenhängender Graph ohne Zyklen), der einen Anfangsknoten hat und von dessen Knoten eine, zwei oder keine Kante ausgehen. Diese Datenstruktur heißt **Binärbaum**.

**Definition 1**

Ein **Binärbaum** ist ein gerichteter Graph mit einem ausgezeichneten Knoten, Wurzel genannt, in dem von jedem Knoten höchstens zwei Kanten ausgehen. Die Nachfolgeknoten jedes Knotens werden Kinder genannt. Es wird zwischen rechtem und linkem Kind unterschieden.

Die Knoten, die keine Kinder haben, heißen **Blätter**.

Beispiel: In dem Morse-Binärbaum hat die Wurzel keinen Inhalt. Der Knoten S ist das linke und der Knoten U das rechte Kind des Knotens I. Blätter sind die Knoten H, V, F, ...

Mit den Klassen Graph, Edge und Vertex ließen sich mit Erweiterungen durch Unterklassen auch Binärbäume verwalten.

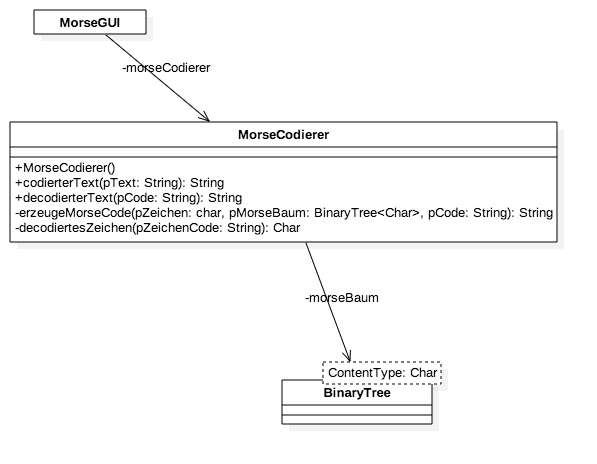
Die Datenstruktur Binärbaum lässt sich aber auch unabhängig vom Graphen wie folgt rekursiv definieren.

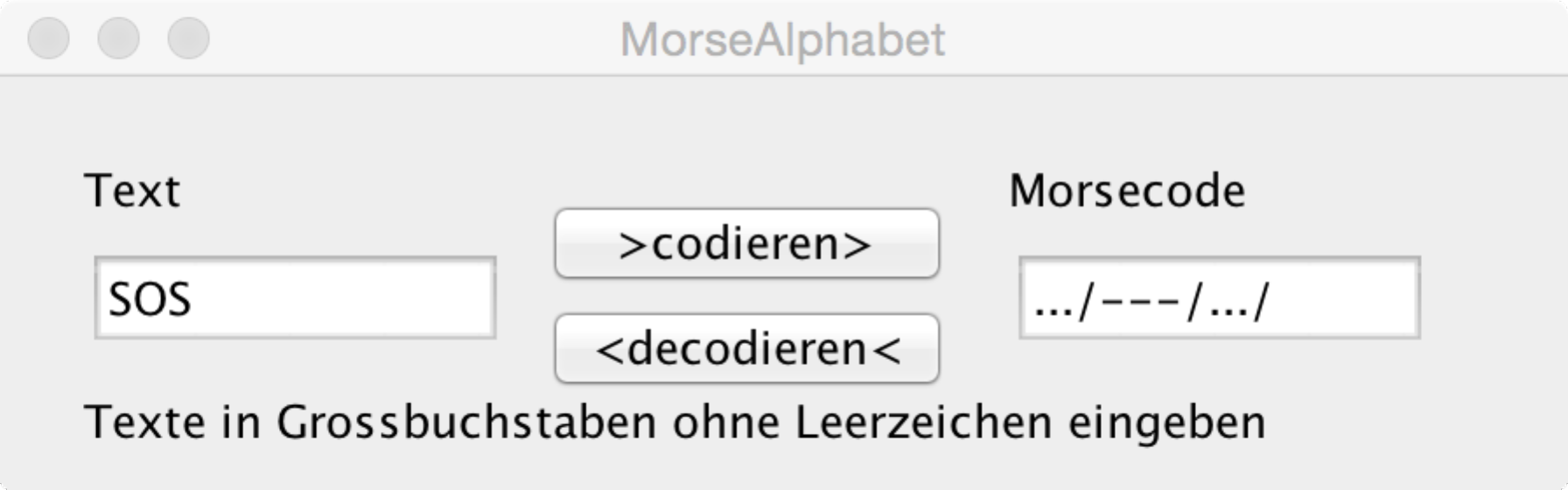
**Definition 2**

Ein **Binärbaum** ist entweder leer oder er besteht aus einer Wurzel und einem linken und rechten Binärbaum, Teilbaum genannt.

Danach sind Blätter also Binärbäume, die zwei leere Teilbäume haben.

Zur Erzeugung und Verwaltung von Binärbaum-Objekten steht die generische Klasse BinaryTree<ContentType> zur Verfügung, die sich an der rekursiven Definition orientiert.

**Implementationsdiagramm für den MorseCodiererDecodierer**

**Benutzungsschnittstelle**

Der Konstruktur MorseCodierer()der Klasse MorseCodierer erzeugt den Morsebaum. Um den ASCII-Code verwenden zu können, wurde auf die Umlaute und das Zeichen „CH“ verzichtet. Man beachte die Verwendung verschiedener Konstruktoren der Klasse BinaryTree.

Die Methode codierterText liefert den in den Morsecode ü*b*ersetzten Text pText. Die von ihr aufgerufene private Methode erzeugeMorsecodesucht den Morsecode eines einzelnen Zeichens im Morsebaum.

public MorseCodierer() {

BinaryTree<Character> lBaum4Links = new BinaryTree<Character>(new Character('H'));

BinaryTree<Character> lBaum4Rechts = new BinaryTree<Character>(new Character('V'));

BinaryTree<Character> lBaum3Links = new BinaryTree<Character>(new Character('S'),

lBaum4Links, lBaum4Rechts);

lBaum4Links = new BinaryTree<Character>(new Character('F'));

lBaum4Rechts = new BinaryTree<Character>();

BinaryTree<Character> lBaum3Rechts = new BinaryTree<Character>(new Character('U'),

lBaum4Links, lBaum4Rechts);

BinaryTree<Character> lBaum2Links = new BinaryTree<Character>(new Character('I'),

lBaum3Links, lBaum3Rechts);

lBaum4Links = new BinaryTree<Character>(new Character('L'));

lBaum4Rechts = new BinaryTree<Character>();

lBaum3Links = new BinaryTree<Character>(new Character('R'), lBaum4Links,

lBaum4Rechts);

lBaum4Links = new BinaryTree<Character>(new Character('P'));

lBaum4Rechts = new BinaryTree<Character>(new Character('J'));

lBaum3Rechts = new BinaryTree<Character>(new Character('W'), lBaum4Links,

lBaum4Rechts);

BinaryTree<Character> lBaum2Rechts = new BinaryTree<Character>(new Character('A'),

lBaum3Links, lBaum3Rechts);

BinaryTree<Character> lBaum1Links = new BinaryTree<Character>(new Character('E'),

lBaum2Links, lBaum2Rechts);

lBaum4Links = new BinaryTree<Character>(new Character('G'));

lBaum4Rechts = new BinaryTree<Character>();

lBaum3Rechts = new BinaryTree<Character>(new Character('O'), lBaum4Links,

lBaum4Rechts);

lBaum2Rechts = new BinaryTree<Character>(new Character('M'),

lBaum3Links, lBaum3Rechts);

BinaryTree<Character> lBaum1Rechts = new BinaryTree<Character>(new Character('T'),

lBaum2Links, lBaum2Rechts);

morseBaum = new BinaryTree<Character>(new Character(' '), lBaum1Links,

lBaum1Rechts);

}

public String codierterText (String pText) {

int lZaehler = 0;

String lCode = "";

while ((lZaehler < pText.length()) && (pText.charAt (lZaehler) >= 'A') &&

(pText.charAt (lZaehler) <= 'Z')) {

lCode = lCode + erzeugeMorsecode (pText.charAt(lZaehler), morseBaum, "") + "/";

lZaehler++;

}

return lCode;

}

private String erzeugeMorsecode (char pZeichen, BinaryTree<Character> pMorseBaum,

String pCode) {

if (!pMorseBaum.isEmpty()){

if (pZeichen == pMorseBaum.getContent()){

return pCode;

}

else {

String lCodeLinkerTeilbaum = erzeugeMorsecode(pZeichen,

pMorseBaum.getLeftTree(), pCode+".");

if (lCodeLinkerTeilbaum.equals(""))

return erzeugeMorsecode(pZeichen, pMorseBaum.getRightTree(), pCode+"-");

else

return lCodeLinkerTeilbaum;

}

}

else

return "";

}

Die Methode decodierterText liefert die Decodierung des Morsecodes pCode. Die von ihr aufgerufene private Methode decodiertesZeichendecodiert ein einzelnes Zeichen mit Hilfe des Morsebaums.

public String decodierterText (String pCode) {

String lText = "";

String lMorsezeichen;

if(pCode.charAt(pCode.length()-1) != '/');

pCode = pCode + '/';

do {

lMorsezeichen=pCode.substring(0, pCode.indexOf('/'));

pCode = pCode.substring(pCode.indexOf('/')+1,pCode.length());

if (lMorsezeichen != "") {

lText=lText+decodiertesZeichen(lMorsezeichen);

}

} while (!pCode.equals(""));

return lText;

}

}

public Character decodiertesZeichen( String pZeichenCode){

int lZaehler = 0;

BinaryTree<Character> lBaum = morseBaum;

while (lZaehler < pZeichenCode.length()) {

if (pZeichenCode.charAt(lZaehler) == '.')

lBaum = lBaum.getLeftTree();

else

lBaum = lBaum.getRightTree();

lZaehler++;

}

return lBaum.getContent();

}

**LK-Q1-V**

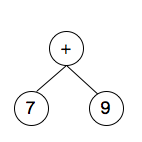
1. **Die Datenstruktur Binärbaum als Spezialfall eines Graphen im Anwendungskontext unter Nutzung der Klasse BinaryTree<ContentType>**
2. Implementierung der Traversierung eines Binärbaums im Pre-, In- und Postorderdurchlauf

**Termbäume**

Mathematische Terme lassen sich entsprechend der Prioritäten der Operatoren in einem Binärbaum darstellen.

Beispiel 1:

Term: 7 + 9

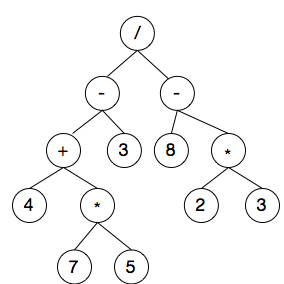
Termbaum:

In der Wurzel steht der Operator, in den Blättern die Operanden. Bei einem längeren Term steht der zuletzt auszuführende Operator in der Wurzel, der 1. Operand bildet den linken Teilbaum, der 2. Operand den rechten Teilbaum.

Beispiel 2:

Term: (4 + 7 \* 5 - 3) / (8 – 2 \* 3)

Unter Berücksichtigung der Regel „Punkt- vor Strichrechnung“ ergibt sich folgender

Termbaum:

Es gibt drei Möglichkeiten den Baum vollständig zu durchlaufen. Man spricht von Traversierungen.

**Pre-Order-Traversierung**

Wurzel

Linker Teilbaum

Rechter Teilbaum

Besuchsreihenfolge der Knoten: / - + 4 \* 7 5 3 - 8 \* 2 3

**In-Order-Traversierung**

Linker Teilbaum

Wurzel

Rechter Teilbaum

Besuchsreihenfolge der Knoten: 4 + 7 \* 5 - 3 / 8 - 2 \* 3

**Post-Order-Traversierung**

Linker Teilbaum

Rechter Teilbaum

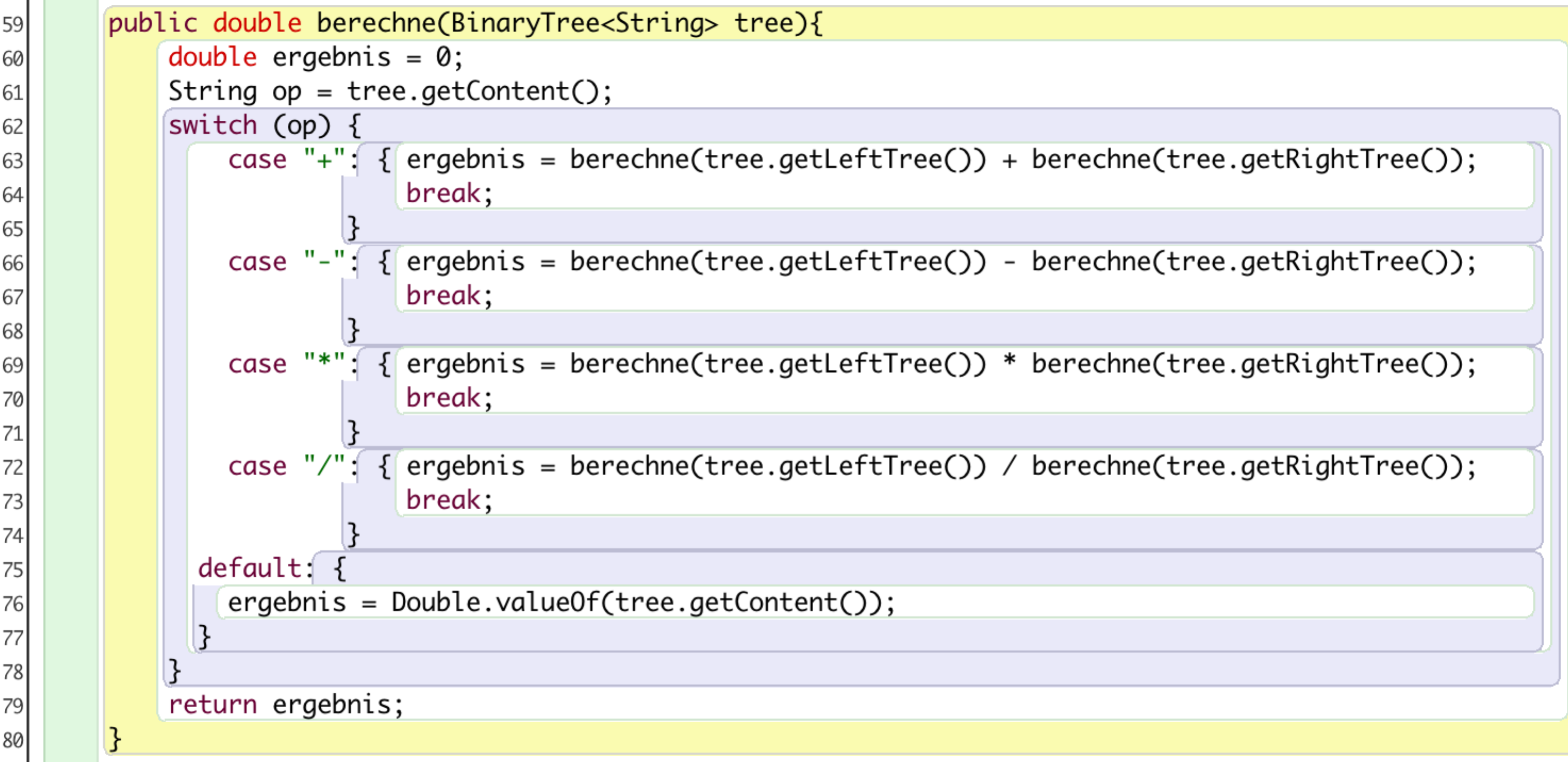
Wurzel

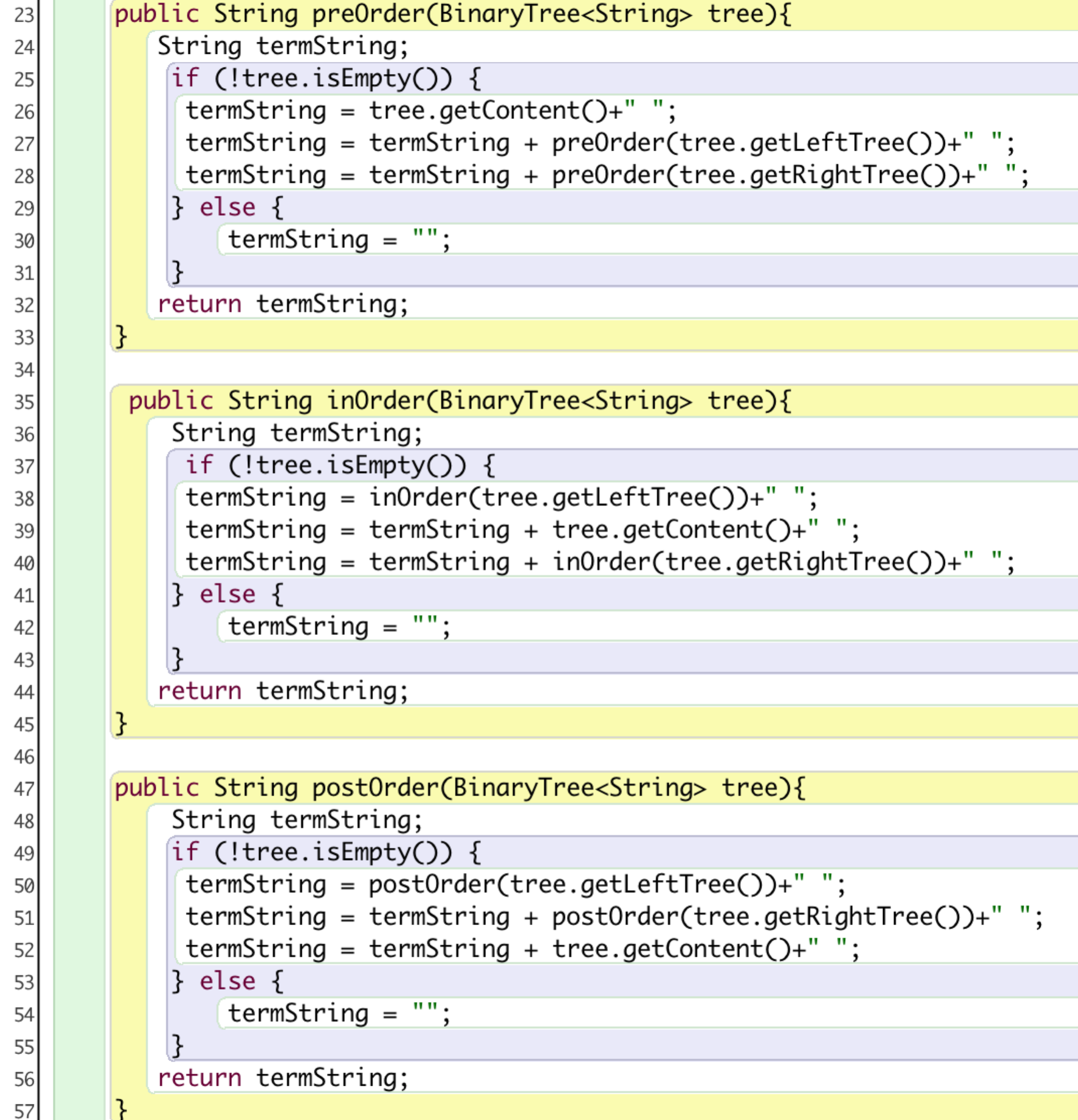
Besuchsreihenfolge der Knoten: 4 7 5 \* + 3 - 8 2 3 \* - /

Die In-Order-Traversierung erzeugt den Term in der Infix-Notation für arithmetische Ausdrücke, es fehlen allerdings die Klammern, sodass dieser Term ein anderes Ergebnis liefern würde, als der ursprüngliche Term.

Die Post-Order-Traversierung liefert den Term in UPN (umgekehrt polnischer Notation). Die ersten Taschenrechner von der Firma HP erforderten die Eingabe in UPN. Der Vorteil lag darin, dass keine Klammern eingegeben werden mussten.

Die Pre-Order-Traversierung ist am besten geeignet, um einen in einem Binärbaum gespeicherten arithmetischen Ausdruck zu berechnen, wie die Java-Implementierung zeigt:

****

**Java-Implementierungen der Traversierungslagorithmen**

**Aufgabe**

a) *Stellen Sie den Term 3\*(4+7/9)-3+(5\*6-8) in einem Binärbaum dar.*

*b) Geben sie die Ergebnisse der Pre-, Post- und In-Order Traversierungen des von Ihnen erzeugten Binärbaums an.*

*c) Analysieren Sie die Methode berechne mit diesem Term.*

**LK-Q1\_VI**

**3. Die Datenstruktur Binärbaum als Spezialfall eines Graphen im Anwendungskontext unter Nutzung der generischen Klasse BinaryTree<ContentType>**

(d) Modellierung und Implementierung einer Anwendung unter Verwendung der Datenstruktur Binärbaum

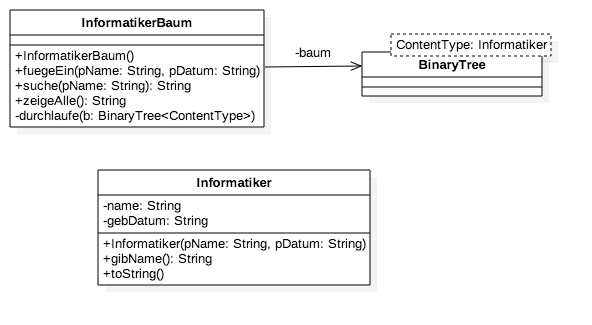
**Informatiker-Baum**

**Aufgabe**

Die Namen und Geburtsdaten von Informatikern sollen so in einem Binärbaum gespeichert werden, dass sie mit Hilfe der inorder-Traversierung nach Namen alphabetisch sortiert ausgelesen werden können.

Lösung:

**Implementationsdiagramm**



Hinweis: Die Methode toString()liefert eine Zeichenkette, die sich aus dem Namen und dem Geburtsdatum des Informatikers zusammensetzt, getrennt durch die Zeichenfolge „ \*“.

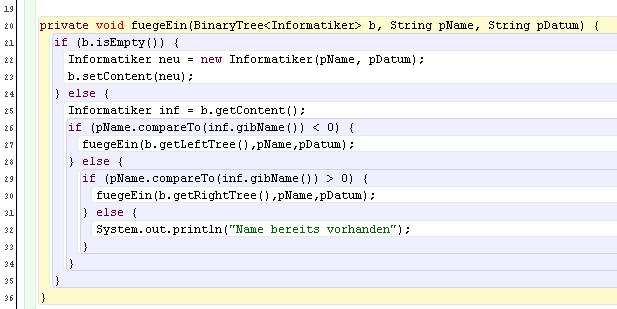


**Einfügen in den Informatiker-Baum**

Damit bei einer inorder-Traversierung die Informatiker-Namen in alphabetischer Reihenfolge geliefert werden, müssen für jeden Teilbaum des Binärbaums folgende Bedingungen erfüllt sein:

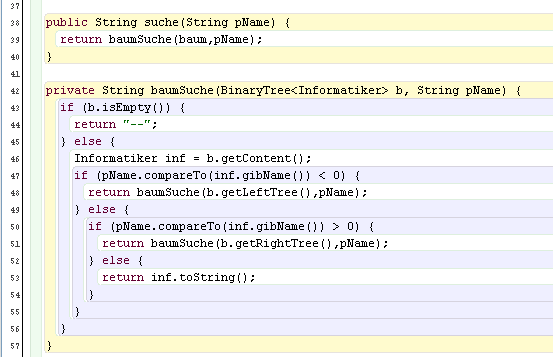
1. Alle Informatiker im linken Teilbaum der Wurzel des Teilbaums liegen in lexikographischer Ordnung vor dem Informatiker in der Wurzel oder die Wurzel hat einen leeren linken Teilbaum
2. Alle Informatiker im rechten Teilbaum der Wurzel des Teilbaums liegen in lexikographischer Ordnung hinter dem Informatiker in der Wurzel oder der Knoten hat einen leeren rechten Teilbaum.

Die folgende Methode fügt einen neuen Datensatz rekursiv in den Binärbaum ein. Bei jedem Teilbaum wird entsprechend der oben angegebenen Regeln geprüft, ob der neue Informatiker in den linken oder rechten Teilbaum eingefügt werden muss. Der neue Knoten wird erzeugt, wenn der Teilbaum, in den der Datensatz eingefügt wird, leer ist. Damit der Baum eindeutig ist, wird verhindert, dass Namen doppelt eingefügt werden.

****

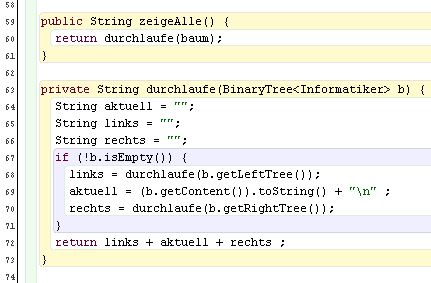
**Suchen im Informatiker-Baum**

Der Baum ist nach Namen sortiert. Durch Vergleich des gegebenen Namens mit dem Namen im Inhaltsobjekt des Teilbaums kann der passende Suchweg im Baum gefunden werden.



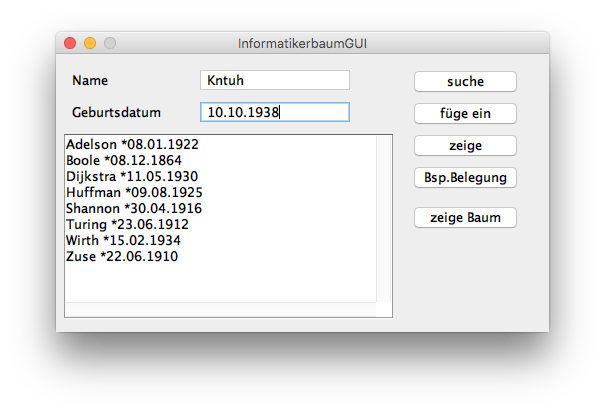
**Sortierte Ausgabe**

Der Baum wird nach der inorder-Strategie durchlaufen und von jedem Inhaltsobjekt des Teilbaums wird der Name und das Geburtsdatum ausgegeben.



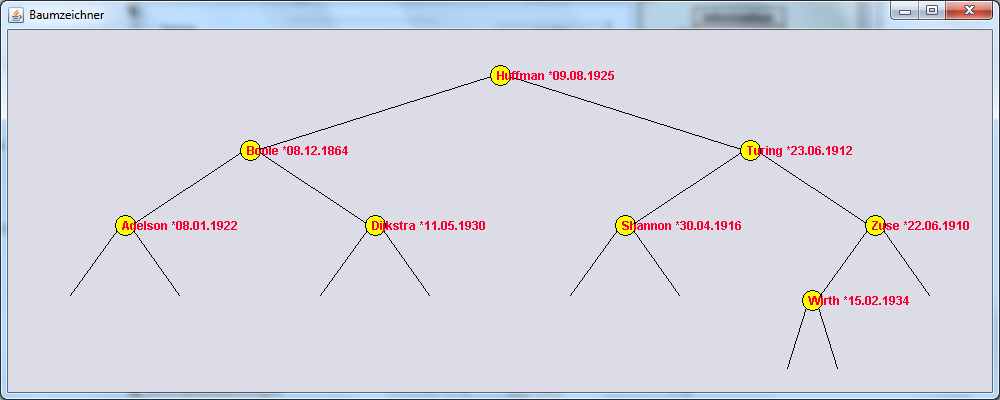
**Benutzungsoberfläche** (mit zusätzlichen, schon implementierten Möglichkeiten, die das Testen erleichtern)

Die im Informatiker-Baum gespeicherten Daten (Name, Geburtsdatum) werden in sortierter Reihenfolge ausgegeben.



Zusatz:Darstellung des Binärbaums

In einem seperaten Fenster wird der Binärbaum ausgegeben.



**LK-Q1-V**

1. **Erarbeitung, Implementierung und Verwendung der Datenstruktur binärer Suchbaum im Anwendungskontext**

(d) Implementierung eines Anwendungsbeispiels einschließlich der sortierten Ausgabe eines binären Suchbaumes

**Stichwortregister**

Bei Sach- und Fachbüchern findet sich häufig am Ende des Buches ein sogenanntes Stichwortregister, in dem für wichtige (alphabetisch sortierte) Fachbegriffe die Seitenzahlen ihres Auftretens aufgelistet sind. Für ein Informatik-Fachbuch könnte dieses Stichwortregister etwa folgendermaßen aussehen (Es sind nur einige Stichwörter dargestellt.):

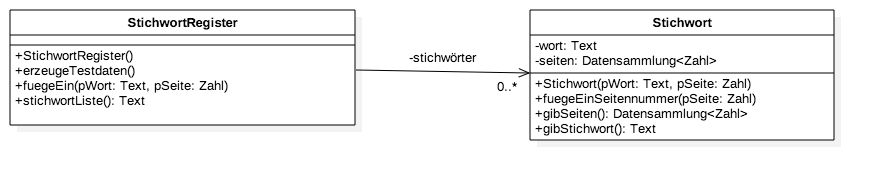


Es soll eine Anwendung entwickelt werden, die anhand von Stichwörtern und zugehörigen Seitennummern ein Stichwortregister erstellt. Der Durchlauf durch das Buch soll automatisch erfolgen, da davon auszugehen ist, dass Bücher heutzutage in elektronischer Form vorliegen.

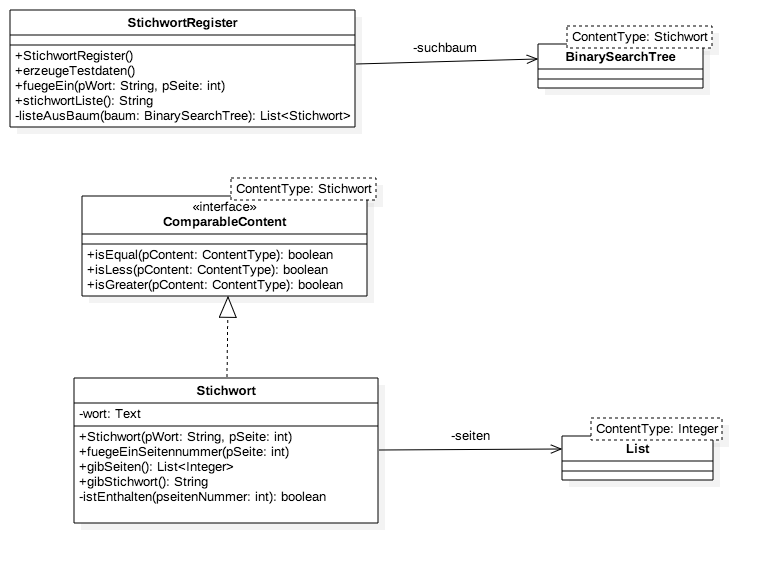
Kommt ein Stichwort mehrfach im Buch vor, so müssen für dieses Wort die entsprechenden Seitennummern übernommen werden. Mehrfaches Auftreten eines Stichwortes auf einer Seite wird nur einmal berücksichtigt.

Die Gesamtheit aller Stichwörter mit Seitenangaben soll in einer geeigneten Datenstruktur erfasst werden.

**Entwurfsdiagramm**



**Implementationsdiagramm**



Bezüglich der Datenstrukturen sind folgende Entscheidungen gefallen:

Klasse **StichwortRegister**

Da die Stichwörter bei der Analyse des Buches häufig gesucht werden müssen, werden sie in der Klasse StichwortRegister als Objekt der generischen Klasse BinarySearchTree<Stichwort> in der Instanzvariablen suchbaumverwaltet.

Klasse **Stichwort**

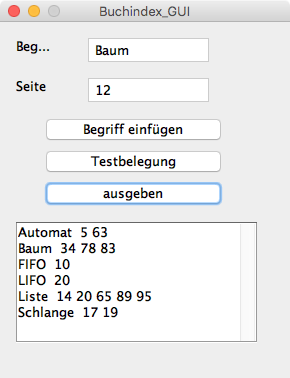
Da die Stichwörter in einem Objekt der Klasse BinarySearchTree verwaltet werden, muss die Klasse Stichwort das Interface ComparableContentimplementieren.

Ein Stichwort kann auf mehreren Seiten im Buch vorkommen. Daher werden die Seitennummern in einer Liste, einem Objekt der generischen Klasse **List**, verwaltet.

Zur Ausgabe können die Seitennummern als Aufzählung hinter dem jeweiligen Stichwort in einem Objekt der Klasse String dargestellt werden, z.B. “Schlange 27 29“.

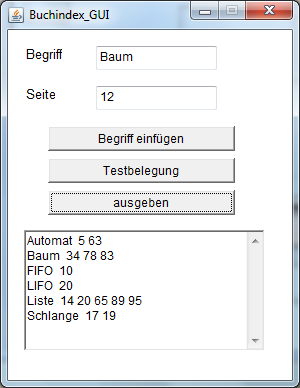
**Benutzungoberfläche**

Eine mögliche Benutzungsoberfläche für eine Testumgebung wird als BlueJ-Projekt vorgegeben.



**Aufgabe**

Implementieren Sie die Methoden der Klasse **StichwortRegister**.



1. de.wikipedia.org Dijkstra-Algorithmus (27.01.2019) [↑](#footnote-ref-1)