

Algorithmen und Datenstrukturen II für GTEC

SS 2009

Dr. Herbert Praehofer
Johannes Kepler Universität Linz
Institut für Systemsoftware

Linz, März 2009

<http://www.ssw.uni-linz.ac.at/General/Staff/HP/>

<http://www.ssw.uni-linz.ac.at/Teaching/Lectures/UZR/Algo2/2009S/index.html>

Inhaltsverzeichnis

1	THEMA, ZIEL, INHALT UND GESTALTUNG DER LEHRVERANSTALTUNG.....	3
1.1	THEMA DER LEHRVERANSTALTUNG	3
1.2	ZIEL DER LEHRVERANSTALTUNG	3
1.3	INHALT (GEPLANT)	3
1.4	GESTALTUNG DER LEHRVERANSTALTUNG	4
1.5	LITERATUR UND UNTERLAGEN.....	4
2	GRUNDSÄTZLICHE BEGRIFFE UND DEFINITIONEN.....	5
2.1	ALGORITHMUSBEGRIFF	5
2.2	DATEN UND DATENTYPEN	5
2.3	DARSTELLUNGSFORMEN VON ALGORITHMEN	6
2.4	KOMPLEXITÄT VON ALGORITHMEN	8
2.5	ZUSAMMENFASSUNG	10
3	ABSTRAKTE DATENTYPEN.....	11
3.1	DEFINITION ABSTRAKTER DATENTYP (ADT).....	11
3.2	FORMALE SPEZIFIKATION VON ADT	12
3.3	JAVA INTERFACES	13
3.4	ADT STACK: STAPEL MIT LAST-IN-FIRST-OUT-STRATEGIE.....	15
3.5	ADT QUEUE: WARTESCHLANGE MIT FIRST-IN-FIRST-OUT-STRATEGIE.....	16
4	BÄUME.....	17
4.1	BEDEUTUNG VON BÄUMEN	17
4.2	BEGRIFFE UND DEFINITIONEN	20
4.3	ABSTRAKTER DATENTYP BAUM.....	24
4.4	DURCHWANDERN VON (BINÄR)BÄUMEN	28
4.5	IMPLEMENTIERUNG VON BÄUMEN.....	30
4.6	ZUSAMMENFASSUNG	36
5	PRIORITÄTSWARTESCHLANGEN UND HEAPS	37
5.1	ADT PRIORITYQUEUE.....	37
5.2	LINKEDLIST-REALISIERUNG VON PRIORITYQUEUE.....	38
5.3	HEAP-REALISIERUNG VON PRIORITYQUEUE	40
5.4	ZUSAMMENFASSUNG	44
6	ADT DICTIONARY UND BINÄRE SUCHBÄUME	45
6.1	ADT DICTIONARY	45
6.2	BINÄRE SUCHE	46
6.3	BINÄRE SUCHBÄUME.....	47
6.4	ZUSAMMENFASSUNG	49
7	GRAPHEN.....	51
7.1	MOTIVATION	51
7.2	DEFINITION GRAPH	52
7.3	WICHTIGE BEGRIFFE	55
7.4	DARSTELLUNGSFORMEN FÜR GRAPHEN	59
7.5	DURCHWANDERN VON GRAPHEN (TRAVERSIERUNG).....	67
7.6	ZUSAMMENFASSUNG	74
8	ALGORITHMEN AUF GRAPHEN.....	76
8.1	MINIMALER SPANNBAUM.....	76
8.2	KÜRZESTER WEG (SINGLE SOURCE SHORTEST PATH PROBLEM).....	85
8.3	FLUSSOPTIMIERUNG	95
8.4	ALGORITHMEN AUF GRAPHEN: ZUSAMMENFASSUNG.....	101
9	GEOMETRISCHE ALGORITHMEN.....	102
10	BACKTRACKING UND NP-VOLLSTÄNDIGKEIT.....	103

1 Thema, Ziel, Inhalt und Gestaltung der Lehrveranstaltung

1.1 Thema der Lehrveranstaltung

Die LVA beschäftigt sich mit:

- komplexen Algorithmen
- Datenstrukturen
- Abstrakten Datentypen (ADT)

mit einer Gewichtung auf Themen, die im Bereich Geoinformatik von Interesse sind:

- Graphen und Wegsuche
- Geometrische Algorithmen

1.2 Ziel der Lehrveranstaltung

Ziel der Lehrveranstaltung ist:

- Wissen über Algorithmen vertiefen
- relevante Algorithmen und Datenstrukturen kennenlernen und verstehen
- komplexe Algorithmen in Programme umsetzen können
- Realisierung von komplexen Datenstrukturen und Abstrakten Datentypen
- Verständnis über die Komplexität von bestimmten Problemstellungen und Verfahren (im Bereich Geoinformatik) gewinnen.

1.3 Inhalt (geplant)

Folgende Themen werden in der LVA behandelt:

- Wiederholung wichtiger Themen aus Algorithmen 1
 - Algorithmusbegriff
 - Notation von Algorithmen
 - Komplexitätsanalyse und O-Notation
 - Abstrakte Datentypen
- Bäume
- Graphen

- Geometrische Algorithmen
- Backtracking und NP-Vollständigkeit

1.4 Gestaltung der Lehrveranstaltung

Abhaltung der LV an unterschiedlichen Terminen:

- VL mit Vortrag
- Übungsstunden mit
 - Projektausarbeitung
 - Projektbesprechung
 - Projektpräsentation

Übungsaufgaben:

- Programmieraufgaben in Java analog zum Stoff

Benotung auf Basis von:

- Projekt
- mündliche Prüfung

1.5 Literatur und Unterlagen

Zur LVA gibt es ein Folienskriptum in dem die Inhalte der LVA kurz zusammengefasst sind.

Darüber hinaus sind folgende Bücher für die Vertiefung des Stoffes wesentlich:

- R. Sedgewick: Algorithms in Java, Part 1-4, Addison-Wesley, 2003
- R. Sedgewick: Algorithmen 2. Aufl., Addison-Wesley, 1992
- R. Sedgewick: Algorithms in C++, Part 5, Addison-Wesley,
- Ottmann/Widmayer: Algorithmen und Datenstrukturen, 3. Auflage, Spektrum Akademischer Verlag, 1996.
- M. A. Weiss: Data structures & problem solving using Java, Addison-Wesley, 1998.
- M. de Berg, M. vom Kreveld, M. Overmars, O. Schwarzkopf: Computational Geometry, Springer, 2000.

Weiteres Material und relevante Informationen werden laufend auf der Homepage der LVA

<http://www.ssw.uni-linz.ac.at/Teaching/Lectures/UZR/Algo2/2009S/index.html> online zur Verfügung gestellt.

2 Grundsätzliche Begriffe und Definitionen

(teilweise Wiederholung aus Algorithmen I)

2.1 Algorithmusbegriff

siehe auch Skriptum Algorithmen I, S 6 - 9

Definition

Algorithmus ist ein endliches, schrittweises Verfahren zur Berechnung von gesuchten aus gegebenen Größen, in dem jeder Schritt aus einer Anzahl ausführbarer, eindeutiger Operationen und einer Angabe über den nächsten Schritt besteht.

Eigenschaften von Algorithmen :

- schrittweise
- gesuchte und gegebene Größen
- eindeutig
- ausführbar
- endlich
 - endliche Beschreibung
 - endliche Ausführung (= Algorithmus terminiert mit einem Ergebnis in endlicher Zeit)
- Aneinanderreihung komplexer Aktionen
 - sind aus Elementaraktionen zusammengesetzt
 - Anordnung von Elementaraktionen in Sequenzen, Verzweigungen, und Iterationen

2.2 Daten und Datentypen

Algorithmen manipulieren Daten:

- Durch die Ausführung von Algorithmen werden Daten verändert, d.h. Algorithmen manipulieren Werte und Objekte
- Daten sind in Variablen gespeichert, d.h. Ausführen von Algorithmen bedeutet Zustandsänderungen von Variablen

- Daten sind in bestimmten Strukturen (Datenstrukturen) organisiert
- Es besteht daher ein wesentlicher Zusammenhang zwischen Algorithmen und Datenstrukturen
 - Daten sind für die Algorithmen in geeigneter Weise strukturiert (Algorithmen bauen auf Datenstrukturen auf)
 - Algorithmen organisieren Daten in bestimmter Weise (z.B. sortiert)

Datentypen

- Daten sind von einem bestimmten Datentyp. Der Datentyp bestimmt den Wertebereich der Daten und welche Operationen zur Verfügung stehen
 - Elementare Datentypen (Zahlen, Zeichen, Booleans, ...)
 - Komplexe Datentypen (Objekttypen = Klassen)

2.3 Darstellungsformen von Algorithmen

siehe dazu Skriptum Algorithmen I, S 11 – 13

- Prosa
- Ablaufdiagramm
- Struktogramm
- Algorithmenbeschreibungssprache ("Pseudocode", Jana)
- Programmiersprache
- Kombination von Programmiersprache und Prosa

Bevorzugte Darstellungsart in dieser LVA:

- Stilisierte Prosa
- Java + Prosa, d.h. Java mit beliebiger Prosa gemischt

Beispiele:

Beispiel SquareRoot: Stilisierte Prosa

gegeben: Zahl a

gesucht: Quadratwurzel aus a : x

1. lege Genauigkeit fest: 6 Kommastellen
2. Beginne mit einem beliebigen, plausiblen Schätzwert für x :
 $x = a/10$
3. Prüfe den Unterschied zwischen $x*x$ und a
Wenn Unterschied innerhalb der geforderten Genauigkeit,

- dann x gefunden, Ende
4. Berechne neuen Wert für x:
 $x = \text{Mittelwert von alten Wert } x \text{ und Wert } a/x$
 Setze bei 3. fort

Beispiel SquareRoot: Ablaufdiagramm

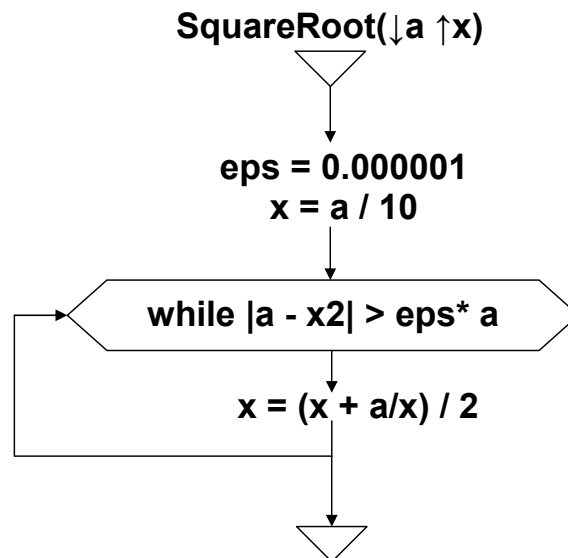


Abbildung 1: Ablaufdiagramm SquareRoot

Beispiel SquareRoot: Struktogramm

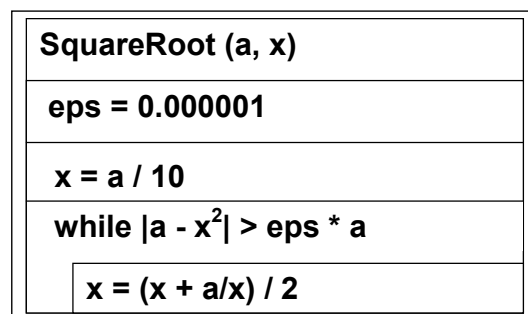


Abbildung 2: Ablaufdiagramm SquareRoot

Beispiel SquareRoot: Java + Prosa

```

static double squareRoot(double angle, double eps) {
    double x
    Initialisiere x mit a / 10
    while (Genauigkeit eps nicht erreicht) {
        setze x auf (x + a/x) / 2
    }
    return x;
}
  
```

Beispiel SquareRoot: Jana

```

SquareRoot (↓a ↑x)
{
    x = a / 10
  }
  
```

```

while (x noch zu ungenau) {
    x = (x + a/x) / 2
}

```

2.4 Komplexität von Algorithmen

Algorithmen zeichnen sich durch mehrere Qualitäten aus:

- Laufzeit
- Speicherbedarf
- Aufwand für Verstehen und Programmierung

Algorithmen werden über diese Qualitäten miteinander verglichen. Der Vergleich erlaubt die Auswahl eines Algorithmus für eine bestimmte Problemstellung aus einer Menge von möglichen.

Algorithmenanalyse

Während der Aufwand für Verstehen und Programmierung nur qualitativ abgeschätzt werden kann, möchte man Laufzeit und Speicherbedarf quantitativ abschätzen und angeben.

Algorithmen können in Bezug auf Laufzeit und Speicherkomplexität quantitativ untersucht werden. Man spricht von

=> Algorithmenanalyse

Es gibt 2 Arten von Algorithmenanalyse:

- Empirische Analyse: Der Algorithmus wird in einem Programm implementiert und seine Laufzeit und Speicherbedarf durch Messungen untersucht.
- Mathematische Analyse: Das Verhalten des Algorithmus in Bezug auf Laufzeit und Speicherbedarf wird durch formale Schließen abgeleitet und in Form einer mathematischen Formel angegeben.

Mit der mathematischen Algorithmenanalyse lassen sich Aussagen über Laufzeit und/oder Speicherbedarf unabhängig von der konkreten Anwendung und Programmausführung treffen. Sie ist aber oft eine sehr aufwendige, mathematisch anspruchsvolle Aufgabe. Für viele Standardalgorithmen (für jene in dieser LVA besprochenen Algorithmen) wurden detaillierte Analysen durchgeführt.

O-Notation

Für die Angabe der Komplexität von Algorithmen verwendet man die *O-Notation*. Diese gibt in Form einer Formel die Komplexität des Algorithmus abhängig von der Problemgröße an (vereinfachte Definition):

Die O-Notation gibt die Menge der Ausführungsschritte für eine gegebene Problemgröße an, wobei konstante Faktoren nicht berücksichtigt werden.

Die O-Notation gibt daher eine Art „Größenordnung“ für die Komplexität an:

- Die O-Notation stellt die Größenordnung der Komplexität dar und abstrahiert von den Details
- Die O-Notation erlaubt die Komplexität von Algorithmen zu vergleichen

Die O-Notation ist deshalb so wichtig, weil ab einer gewissen Problemgröße die konstanten Faktoren zusehends weniger Rolle spielen.

Beispiele von algorithmischer Komplexität: Suche in einem geordneten Feld

- Sei N die Problemgröße == die Anzahl der Elemente in einem geordneten Feld
- Der Algorithmus *Sequential Search*, welcher das Array sequentiell durchläuft, braucht durchschnittlich N/2 Schritte; seine Komplexität ist

O(N)

- Der Algorithmus *BinarySearch*, welcher das Element nach dem binären Suchverfahren im sortierten Array sucht, braucht durchschnittlich $\log(N)$ Schritte; seine Komplexität ist

O($\log(N)$)

- was wesentlich besser ist.

Tabelle mit wichtigen Komplexitätstermen

Tabelle 1 zeigt die Entwicklung der Komplexität für unterschiedliche Komplexitätsterme bei steigender Problemgröße. Man beachte, wie bei bestimmten Termen die Komplexität praktisch „explodiert“. Solche Komplexitätszahlen (z.B. $2^{**}N$ oder N!) geben an, dass der Algorithmus für größere Probleme nicht funktioniert (siehe dazu NP-Vollständigkeit im letzten Teil der LVA).

N	$\log(N)$	$N^{**}2$	$N^{**}4$	$2^{**}N$	$4^{**}N$	N!
2	1,00	4	16	4	16	2
3	1,58	9	81	8	64	6
4	2,00	16	256	16	256	24
5	2,32	25	625	32	1024	120
10	3,32	100	10000	1024	1048576	3628800
20	4,32	400	160000	1048576	1,09951E+12	2,4329E+18
50	5,64	2500	6250000	1,1259E+15	1,26765E+30	3,04141E+64
100	6,64	10000	100000000	1,26765E+30	1,60694E+60	9,3326E+157
200	7,64	40000	1600000000	1,60694E+60	2,5822E+120	#ZAHL!

500	8,97	250000	62500000000	3,2734E+150	1,0715E+301	#ZAHL!
1000	9,97	1000000	1E+12	1,0715E+301	#ZAHL!	#ZAHL!
10000	13,29	100000000	1E+16	#ZAHL!	#ZAHL!	#ZAHL!
100000	16,61	10000000000	1E+20	#ZAHL!	#ZAHL!	#ZAHL!

#ZAHL! bedeutet durch Excel nicht berechenbar

Tabelle 1: Komplexitätszahlen bei unterschiedlichen Komplexitätstermen für steigende Problemgrößen N

2.5 Zusammenfassung

- Algorithmus ist ein endliches, schrittweises Verfahren zur Berechnung von gesuchten aus gegebenen Größen
- Algorithmen manipulieren Daten; Daten liegen in Datenstrukturen vor, die durch Datentypen definiert sind.
- Darstellungsformen für Algorithmen sind
 - Stilisierte Prosa
 - Ablaufdiagramme und Struktogramme
 - Pseudocode
 - Java + Prosa
 - Java
- Algorithmen zeichnen sich durch ihre Komplexität in Bezug auf Laufzeit und Speicherbedarf aus; die Größenordnung der Komplexität wird in der O-Notation abhängig von der Problemgröße angegeben.

3 Abstrakte Datentypen

3.1 Definition Abstrakter Datentyp (ADT)

Ein abstrakter Datentyp (kurz und im folgenden ADT) beschreibt Daten in abstrakter Weise durch:

- Wertebereich der Daten (als konstante Funktionen angegeben)
- Operationen definiert für den Datentyp (= Funktionen)
 - Syntax: Schnittstelle der Operation
 - Semantik: Ergebnis der Operation

ADT abstrahieren von der eigentlichen Implementierung:

- nur Schnittstellendefinition
- mehrere Implementierungen eines ADT möglich
- durch "Verlassen" auf die Schnittstellendefinition wird Wiederverwendbarkeit erreicht

Beispiele

ADT: int

int ist der Datentyp der ganzen Zahlen mit den üblichen Operatoren (in Funktionsschreibweise):

```
abstract-data-type int
  ..., -2, -1, 0, 1, 2, ... → int // Wertebereich
  +: int × int → int
  -: int × int → int
  * : int × int → int
  / : int × int → int
  % : int × int → int
  < : int × int → boolean
  == : int × int → boolean
  > : int × int → boolean
end int
```

mit der üblicher Interpretation der Operatoren.

ADT: List

List sei ein ADT, der eine Ansammlung von Objekten, mit Anfügen/Löschen von Objekten und einen direkten Zugriff auf Objekte an bestimmten Positionen definiert:

- Anfügen eines Objekts: `insert`

- Löschen eines Objekts: remove
- Lesen eines Objekts an der i-ten Stelle: at
- Prüfen, ob ein Element enthalten ist: contains
- Anzahl der Objekte in einer List: size

Abstrakte Datentyp List von Objects in Funktionsschreibweise

```

abstract-data-type List
  create:      → List    // Konstante leere Liste
  insert: Object × List → List
  remove: Object × List → List
  at: List × int → Object
  contains: Object × List → boolean
  size: List → int
end List

```

oder in Pseudocode

```

abstract-data-type List = {
  List create();
  void insert(Object o);
  void remove(Object o);
  Object objectAt(int i);
  boolean contains(Object o);
  int size();
}

```

3.2 Formale Spezifikation von ADT

ADT sind ein wichtiges theoretische Gebiet und werden in der theoretischen Informatik behandelt. ADT werden dabei als eine Menge von Funktionen definiert. Für diese Funktionen wird über ein Axiomensystem die genaue Semantik der Operationen angegeben. Vorteil dieses Vorgehens ist, dass mit ADT formale Beweisführung möglich ist. Man kann z.B. formal zeigen, ob eine bestimmte Implementierung konform dem im Axiomensystem definierten Semantik ist.

Beispiel: ADT Stack für Elemente von Typ T

Der ADT Stack ist über die Operationen create, push, pop, top, und empty definiert. Man beachte die Axiome, die genau angeben, was man sich von den Operationen erwartet.

```

abstract-data-type Stack
  operations
    create:      → Stack    // Konstante leerer Stack
    push: T × Stack → Stack
    top: Stack → T
    pop: Stack → Stack
    empty: Stack → Boolean
  axioms (a: T, s: Stack)
    empty(create) == true

```

```
    empty(push(a, s) == false
    top(push(a, s)) == a
    pop(push(a, s)) == s
end Stack
```

Wie obiges Beispiel zeigt, kann die formale Semantik von Operationen sehr aussagekräftig sein. Wir wollen uns aber in dieser LVA nicht weiter mit der formalen Theorie der abstrakten Datentypen beschäftigen.

In dieser LVA wählen wir vielmehr einen sehr praktischen, programmieretechnischen Zugang zu ADTs. Die Semantik der Operationen von ADT wollen wir dabei nicht formal sondern in der Form von **Kommentaren bei Methoden** angeben bzw. mittels JUnit-Tests spezifizieren.

3.3 Java Interfaces

Mit Interfaces existiert in Java ein Sprachkonzept zur Definition von ADTs

- bestehen aus einer Reihe von Methodendeklarationen
- ohne Implementierungen
- erlauben keine formale Semantik
- Mehrfachvererbung bei Schnittstellen möglich

Beispiel: Java-Interface List

Das folgende Beispiel zeigt die Definition eines Java-Interfaces entsprechend dem ADT List. Dieses Interface stellt eine abstrakte Definition dar, die unterschiedlich implementiert werden kann.

Java-Interface List:

```
/**
 * Spezifikation des ADT List
 */
public interface List {

    void addObject(Object o);

    void removeObject(Object o);

    Object objectAt(int i);

    boolean contains(Object o);

    int size();
}
```

Implementierung von List mittels Array

```

public class ArrayList implements List {

    private Object[] arr;
    int nrElems;

    public ArrayList(int initialSize) {
        arr = new Object[initialSize];
        nrElems = 0;
    }

    public void addObject(Object o) {
        if (nrElems == arr.length-1) {
            // dynamisches Vergrößern des Arrays
            Object[] newArr = new Object[nrElems*2];
            for (int i = 0; i < arr.length; i++) {
                ...
            }
        }

        public Object objectAt(int i) {
            if (i >= nrElems) {
                ...
            }
        }
    }
}

```

Implementierung ADT List als verkettete Liste

Arbeitsblatt

3.4 ADT Stack: Stapel mit Last-In-First-Out-Strategie

Ein weiterer wichtiger Abstrakter Datentyp ist der *Stapel* (oder englisch *Stack*, siehe auch formale Definition im Abschnitt 3.2). Die Idee beim Stapel ist, dass Elemente

- oben auf den Stapel (on top) gelegt werden und
- auch wieder von oben entnommen werden.

Diese Strategie des Aufbaus und Abbaus eines Stapels wird mit

***Last-in-First-Out* (kurz: *LIFO*)**

bezeichnet (im Gegensatz zu der *First-In-First-Out*-Strategie einer Queue).

Wesentliche Operationen eines Stapels sind

- push: ein Element oben auf den Stapel legen
- pop: das oberste Element vom Stapel entnehmen
- top: lesender Zugriff auf das oberste Element (d.h. ohne es zu entnehmen)
- empty: prüfen, ob der Stapel leer ist

Java-Interface Definition Stack

```
public interface Stack {  
  
    /**  
     * Legt ein Objekt element zu oberst auf den Stack  
     */  
    void push (Object element);  
  
    /**  
     * Liefert das oberste Element im Stack; wirft eine  
     * StackEmptyException, wenn der Stack leer ist.  
     */  
    Object top() throws StackEmptyException;  
  
    /**  
     * Nimmt das oberste Element vom Stack; wirft eine  
     * StackEmptyException, wenn der Stack leer ist.  
     */  
    Object pop() throws StackEmptyException;  
  
    /**  
     * Prüft, ob der Stack leer ist.  
     */  
    boolean empty();  
}
```

3.5 ADT Queue: Warteschlange mit First-In-First-Out-Strategie

Im Gegensatz zum ADT Stapel, erfolgt beim ADT Queue (Warteschlange) die Entnahme der Elemente in der Reihenfolge, wie sie in die Warteschlange eingefügt wurden. Das heisst, wer zuerst in die Warteschlange eingefügt wurde wird auch zuerst entnommen, d.h. nach der Strategie

First-In-First-Out (kurz FIFO)

Wesentliche Operationen der Queue sind

- enqueue: ein Element hinten in die Queue einfügen
- dequeue: das erste, vorderste Element aus der Queue entnehmen
- first: lesender Zugriff auf das erste Element (d.h. ohne es zu entnehmen)
- empty: prüfen, ob die Queue leer ist

Java-Interface Definition Queue

```
public interface Queue {  
  
    /**  
     * Fügt das Objekt element hinten in die Queue an.  
     */  
    void enqueue (Object element);  
  
    /**  
     * Liefert das erste Element der Queue; wirft eine  
     * QueueEmptyException, wenn die Queue leer ist.  
     */  
    Object first() throws QueueEmptyException;  
  
    /**  
     * Nimmt das erste Element aus der Queue; wirft eine  
     * QueueEmptyException, wenn die Queue leer ist.  
     */  
    Object dequeue() throws QueueEmptyException;  
  
    /**  
     * Prüft, ob die Queue leer ist  
     */  
    boolean empty();  
}
```

4 Bäume

Literatur: R. Sedgewick: Algorithms in Java, § 5.4 ff

4.1 Bedeutung von Bäumen

Bäume (englisch *Trees*) sind eine mathematische Abstraktion, welche in der Informatik eine große Rolle spielt. Sie kommt häufig zum Einsatz als

- natürliche Datenstruktur zur Darstellung von hierarchischen Strukturen bei unterschiedlichsten Anwendungsfällen:
 - Darstellung von hierarchischen Organisationen (z.B.: Stammbäume von Familien, Firmenorganisationen oder hierarchische Organisationen in der Geographie)
 - Entscheidungsbäume
 - Syntaxbäume von Ausdrücken
 - ...
- Datenstrukturen für effiziente Lösungen bei bestimmten Problemstellungen
 - Suchbäume für schnelle Suche
 - geordnete Mengen für schnellen Zugriff aufs erste Element

Beispiele hierarchischer Organisationen

Hierarchisches Dateisystem

Dateisysteme auf Computer sind hierarchisch organisiert, wobei die Directories als innere Knoten fungieren, die als Söhne Dateien und weitere Directories haben.

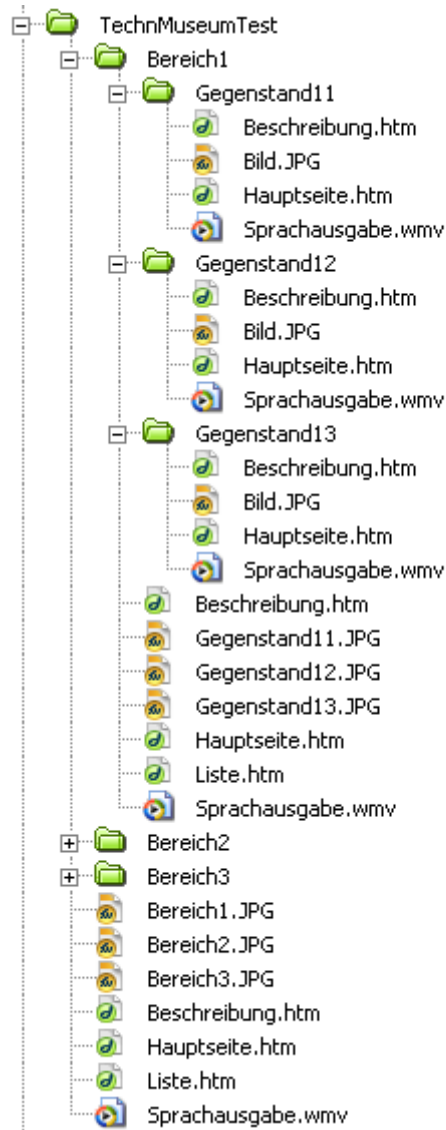


Abbildung 3: Beispiel hierarchisches Dateisystem

Hierarchische Organisation eines Buches

```
Buch  ___ Titel
      |_ Kurzfassung
      |_ $1___Titel1
          |_ $1.1___Titel1.1
              |_ Absatz1.1.1
              |_ Absatz1.1.2
              |_ Absatz1.1.3
              |_ Absatz1.1.4
          |_ $1.2___Titel2.1
              |_ Absatz1.2.1
              |_ Absatz1.2.2
```

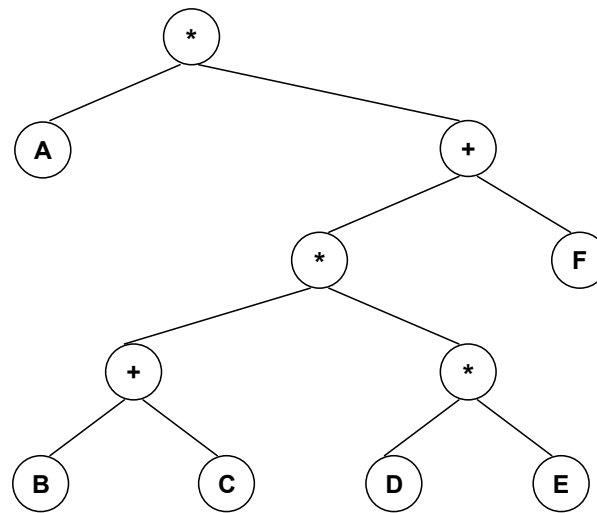



Abbildung 4: Syntaxbaum

4.2 Begriffe und Definitionen

Definition Baum

Definition 1:

Ein Baum besteht aus Knoten und gerichteten Kanten, wobei Kanten von einem Knoten zu dessen Nachfolger führen. Ein Knoten enthält Informationen, eine Kante stellt eine Verbindung zwischen Knoten dar.

Der Baum enthält keine Schleifen und Zyklen:

- es gibt einen ausgezeichneten Knoten, die *Wurzel (root)*, der keinen Vorgänger hat
- ein Knoten in einem Baum enthält eine beliebige Anzahl von Nachfolgern aber nur höchstens einen Vorgänger

Definition 2:

Ein Baum ist ein azyklischer einfacher, zusammenhängender Graph

Terminologie

Für Bäume in der Informatik gibt es eine Reihe von wichtigen Konzepten und Begriffen. Diese werden im Folgenden anhand des Beispielbaums in Abbildung 5 eingeführt:

- Wurzelknoten (root): ist der oberste Knoten im Baum der keinen Vorgänger hat
 - A ist Wurzel
- Elternknoten, Vater oder Mutter (parent): ist der direkte Vorgänger eines Knotens
 - C ist Vater von D und G

- Kindknoten, Sohn, Tochter (*child*): sind die Nachfolger eines Knotens
 - B und C sind Söhne von A
 - E und F sind Söhne von D
 - D und G sind Söhne von C
 - B und C sind Söhne von A

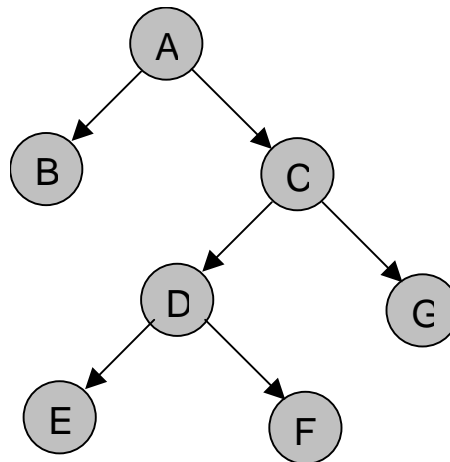
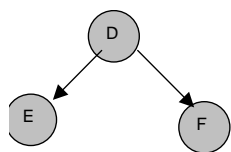




Abbildung 5: Beispiel Baum

- Geschwister-, Bruder oder Nachbarknoten (*sibling*): Knoten mit gleichem Vater
 - E ist Bruder von F
- Unter- oder Teilbaum (*subtree*): ist der gesamte Baum mit einem Sohn als Wurzel



-  ist linker Unterbaum von C
-  ist rechter Unterbaum von C
- Ebenen (*levels*): Bäume lassen sich in mehrere hierarchische Ebenen (*levels*) gliedern. Die Wurzel stellt die oberste Ebene 0 dar, alle direkten Nachfolgeknoten die Ebene 1, alle Nachfolger der Knoten auf Ebene 1 die Ebene 2, usw.
 - A ist auf Ebene 0
 - B und C sind auf Ebene 1
 - D und G sind auf Ebene 2
 - E und F sind auf Ebene 3
- Blatt oder externe Knoten (*leaf or terminal node*): Ein Knoten welcher keine Nachfolger hat, wird als externer Knoten oder Blatt bezeichnet. Die Blätter eines Baums sind alle Knoten ohne Nachfolger
 - B, E, F und G sind Blätter
- Innerer Knoten (*internal or non-terminal node*): sind die Knoten, die einen Nachfolger haben.
 - A, C und D sind innere Knoten

- Grad des Baumes: Der Grad eines Knotens ist die max. Anzahl der Kinder eines Knotens
 - Der Grad des Beispielgraphen ist 2
- Tiefe bis zu Knoten x: Zahl der Kanten von Wurzel bis Knoten x
 - Die Tiefe zu G ist 2
- Höhe eines Baumes: maximale Tiefe von der Wurzel zu einem beliebigen Knoten
 - Die Höhe des Beispielbaums ist 3
- Wald (forest): eine Menge von Bäumen bezeichnet man Wald.

Arten von Bäumen

Geordnete Bäume

Ein Baum heisst geordnet, wenn die Nachfolger der Knoten geordnet (~sortiert) sind, d.h. alle Kinder jedes Knotens nach einem bestimmten Schema geordnet sind (die Ordnung der Nachfolger eine Rolle spielt). Bäume sind fast immer geordnet.

Bei geordneten Bäumen spricht man von

- 1., 2. 3. Kind oder Unterbaum

Binärbäume

Eine wichtige Klasse von geordneten Bäumen sind die sog. **Binärbäume**. Sie sind dadurch charakterisiert, dass jeder Knoten maximal zwei Kinder (= direkte Unterknoten) hat, d.h.

Grad des Baumes = 2

Der Baum aus Abbildung 5 ist ein Binärbaum.

Bei Binärbäumen spricht man

- vom linken und rechten Nachfolger bzw. vom linken und rechten Unterbaum.

Ausgeglichene Bäume

Es ist oft von Vorteil, wenn Bäume ausgeglichen sind, d.h. ihre Höhe soll gering bei gegebener Knotenzahl sein.

Ein Baum ist *vollständig ausgeglichen*, wenn sich bei jedem Knoten die Anzahl der Knoten des rechten und linken Teilbaums maximal um 1 unterscheidet.

! Merke: Ausgeglichene Bäume haben eine minimale Tiefe für eine gegebene Knotenanzahl.

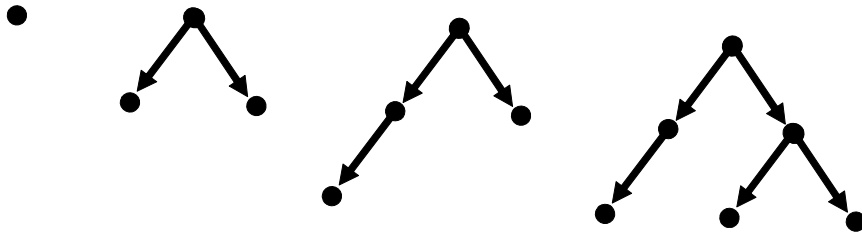


Abbildung 6: Ausgeglichene Bäume

4.3 Abstrakter Datentyp Baum

ADT Tree

Die ADT-Definition eines Baums (Tree) soll die grundlegenden Eigenschaften von Bäumen widerspiegeln. Die folgende beispielhafte ADT-Definition für Tree baut auf dem ADT Knoten (TreeNode) auf.

ADT TreeNode:

Ein ADT TreeNode für Knoten von Bäumen habe folgende Operationen

- `Object value()` - liefert den Wert des Knotens
- `TreeNode[] children()` - liefert die Nachfolgerknoten des Knotens
- `TreeNode parent()` - liefert den Vaterknoten
- `boolean isRoot()` - prüft, ob dieser Knoten Wurzel ist
- `boolean isInternal()` - prüft, ob dieser Knoten interner Knoten ist
- `boolean isLeaf()` - prüft, ob dieser Knoten Blatt ist

Dies entspricht folgender Java-Interface-Definition:

```
public interface TreeNode {  
  
    /** liefert den Wert des Knotens */  
    Object value();  
  
    /** liefert die Nachfolgerknoten des Knotens */  
    TreeNode[] children();  
  
    /** liefert den Vaterknoten */  
    TreeNode parent();  
  
    /** prüft, ob dieser Knoten Wurzel ist */  
    boolean isRoot();  
  
    /** prüft, ob dieser Knoten interner Knoten ist */  
    boolean isInternal();  
  
    /** prüft, ob dieser Knoten Blatt ist */  
    boolean isLeaf();  
  
}
```

ADT Tree:

Ein ADT Tree für Bäume habe folgende Operationen

- `int size()` - Anzahl der Knoten des Baums

- `int height()` - Höhe des Baums
- `Object[] elements()` - liefert die Werte aller im Baum enthaltenen Knoten
- `TreeNode root()` - liefert den Wurzelknoten des Baums
- `TreeNode[] children(TreeNode n)` - liefert die Nachfolgerknoten des Knotens `n`
- `TreeNode parent(TreeNode n)` - liefert den Vaterknoten des Knotens `n`
- `boolean isRoot(TreeNode n)` - prüft, ob Knoten `n` Wurzel ist
- `boolean isInternal(TreeNode n)` - prüft, ob dieser Knoten interner Knoten ist
- `boolean isLeaf(TreeNode n)` - prüft, ob dieser Knoten Blatt ist

Dies entspricht folgender Java-Interface-Definition:

```
public interface Tree {

    /** liefert den Wurzelknoten des Baums */
    TreeNode root();

    /** liefert die Nachfolgerknoten des Knotens n */
    TreeNode[] children(TreeNode n);

    /** liefert den Vaterknoten des Knotens n */
    TreeNode parent(TreeNode n);

    /** prüft, ob Knoten n Wurzel ist */
    boolean isRoot(TreeNode n);

    /** prüft, ob dieser Knoten interner Knoten ist */
    boolean isInternal(TreeNode n);

    /** prüft, ob dieser Knoten Blatt ist */
    boolean isLeaf(TreeNode n);

    /** liefert die Werte aller im Baum enthaltenen Knoten */
    Object[] elements();

    /** liefert die Anzahl der Elemente im Baum */
    int size();

    /** liefert die Höhe des Baums */
    int height();

}
```

! Man beachte, dass in der ADT Definition keine Methoden zum Aufbau von Bäumen definiert sind. Diese sind implementierungsspezifisch und daher in nur in den Implementierungen definiert.

ADT BinaryTree

Auf dieser allgemeine ADT-Definition für Bäume kann man ADT-Definitionen für binäre Bäume wie folgt einführen:

ADT BinaryNode

ADT BinaryNode erweitert ADT TreeNode und definiert folgende zusätzliche Methoden:

- `BinaryNode left()` - liefert den linken Nachfolger
- `BinaryNode right()` - liefert den rechten Nachfolger
- `boolean hasLeft()` - prüft, ob der linke Nachfolger vorhanden
- `boolean hasRight()` - prüft, ob der linke Nachfolger vorhanden

Dies entspricht folgender Java-Interface-Definition:

```
public interface BinaryTreeNode extends TreeNode {  
  
    /** liefert den linken Nachfolger */  
    BinaryTreeNode left();  
  
    /** liefert den rechten Nachfolger */  
    BinaryTreeNode right();  
  
    /** prüft, ob der linke Nachfolger vorhanden */  
    boolean hasLeft();  
  
    /** prüft, ob der linke Nachfolger vorhanden */  
    boolean hasRight();  
  
}
```

ADT BinaryTree extends Tree

ADT BinaryTree erweitert ADT Tree und definiert folgende zusätzliche Methoden

- `BinaryNode left(BinaryNode n)` - liefert den linken Nachfolger von Knoten n
- `BinaryNode right(BinaryNode n)` - liefert den rechten Nachfolger von Knoten n
- `boolean hasLeft(BinaryNode n)` - prüft, ob der linke Nachfolger von n vorhanden
- `boolean hasRight(BinaryNode n)` - prüft, ob der linke Nachfolger von n vorhanden

Dies entspricht folgender Java-Interface-Definition:

```
public interface BinaryTree extends Tree {  
  
    /** liefert den linken Nachfolger des Knotens n */  
    BinaryTreeNode left(BinaryTreeNode n);  
  
    /** liefert den rechten Nachfolger des Knotens n */  
    BinaryTreeNode right(BinaryTreeNode n);  
  
}
```

```
/** prüft, ob der linke Nachfolger des Knotens n vorhanden ist*/  
boolean hasLeft(BinaryTreeNode n);
```

```
/** prüft, ob der rechte Nachfolger des Knotens n vorhanden ist*/  
boolean hasRight(BinaryTreeNode n);
```

```
}
```

4.4 Durchwandern von (Binär)bäumen

Durchwandern (traversing) eines Baums bedeutet das Besuchen aller Knoten in einer bestimmten Reihenfolge. Durchwandern von Bäumen ist die Basis der meisten Algorithmen auf Bäumen.

Durchwandern von Bäumen kann erfolgen durch

- Rekursive Algorithmen
- Nicht-rekursive Algorithmen

Rekursives Durchwandern

Bäume sind rekursive Datenstrukturen. Ein Knoten ist die Wurzel seines Teilbaums. Die Nachfolger des Knotens sind wieder Wurzel ihrer Teilbäume.

In dieser Art sind auch die rekursiven Algorithmen auf Graphen definiert. Die rekursive Prozedur wird mit einem Knoten als Parameter aufgerufen und es erfolgen folgende Aktionen:

- Verarbeitung des Knotens
- rekursive Aufrufe der Prozedur für jeden Nachfolgeknoten

Man unterscheidet die Algorithmen nach der Reihenfolge, in der die Knoten bearbeitet werden. Im folgenden wollen wir die 3 Arten von rekursiven Durchwandern für Binärbäume skizzieren:

- **inorder traversal:** zuerst wird der linke Unterbaum verarbeitet, dann der Knoten selbst, dann der rechte Unterbaum

```
void inorderVisit(BinaryTreeNode node) {
    if (node.hasLeft())
        inorderVisit (node.left());
    node.doOperation();
    if (node.hasRight())
        inorderVisit (node.right());
}
```

- **preorder traversal:** zuerst wird der Knoten selbst verarbeitet, dann der linke, dann der rechte Unterbaum

```
void preorderVisit(BinaryTreeNode node) {
    node.doOperation();
    if (node.hasLeft())
        preorderVisit (node.left());
    if (node.hasRight())
        preorderVisit (node.right());
}
```

- **postorder traversal:** zuerst der linke, dann der rechte Unterbaum, dann der Knoten

```
void postorderVisit (BinaryTreeNode node) {
    if (node.hasLeft())
        postorderVisit (node.left());
    if (node.hasRight())
        postorderVisit (node.right());
    node.doOperation();
}
```

! Man beachte, dass die Algorithmen auf der Basis der ADT BinaryTreeNode ohne Kenntnis der Implementierung formuliert sind.

Nicht-Rekursives Durchwandern

Die obigen rekursiven Algorithmen Durchwandern den Baum nach einem Depth-First-Strategie (*Tiefentraversierung*). Die rekursiven Aufrufe gehen zuerst in die Tiefe, d.h., es werden weiter in der Tiefe liegende Knoten vor den oberen Knoten behandelt.

Aufgabe: Überzeugen Sie sich davon.

Demgegenüber steht die Breitentraversierung oder Breadth-First-Traversal. Hier werden beginnend von der Wurzel alle Knoten einer Ebene behandelt, bevor die Knoten auf der nächsten Ebene behandelt werden.

Breitentraversierung kann nicht rekursiv gelöst werden. Sie verwendet eine Warteschlange (*Queue*), in die die Knoten in der Bearbeitungsreihenfolge gereiht werden. Breitentraversierung erfolgt nach folgendem Schema:

- Der Startknoten wird in die Queue als einziges Element eingereiht.
- Dann wird in einer Schleife solange sich Knoten in der Queue befinden
 - Der erste Knoten in der Queue entfernt.
 - Die Operation für den Knoten ausgeführt
 - Die Nachfolger des Knotens **hinten** in die Queue angefügt.

Algorithmus Breitentraversierung (für Binärbäume)

```
void breath-first(Node start) {
    queue.enqueue(start);
    while (! queue.isEmpty()) {
        Node node = queue.dequeue();
        node.doOperation();
        if (node.hasLeft())
            queue.enqueue(node.left());
        if (node.hasRight())
            queue.enqueue(node.right());
    }
}
```

4.5 Implementierung von Bäumen

Bäume lassen sich gut mit Objekten realisieren, wobei die Knoten einzelne Objekte sind und die Kanten durch Referenzen vom Vater zu den Kindknoten und vom Kindknoten zurück zum Vater aufgebaut werden. Eine weitere, sehr effiziente Form der Realisierung von Binärbäumen ist mittels Arrays.

Baum mittels Objektstruktur

Um einen Baum mittels einer verketteten Objektstruktur zu realisieren, gehen wir wie folgt vor (Abbildung 7):

- Der Baum wird durch ein Objekt realisiert, welche eine Referenz auf den Wurzelknoten speichert. Die Klasse `LinkedListTree` realisiert alle Operationen für den ADT `Tree`.
- Jeder Knoten ist ein eigenes Objekt und enthält die folgenden Referenzen zu anderen Knoten
 - Eine Liste `children` von Referenzen zu den Nachfolgeknoten
 - Eine Referenz `parent` zu dem Vorgängerknoten
 - Eine Referenz auf den Wert beim Knoten, der vom Typ `Object` ist (damit sind alle Objektwerte erlaubt)

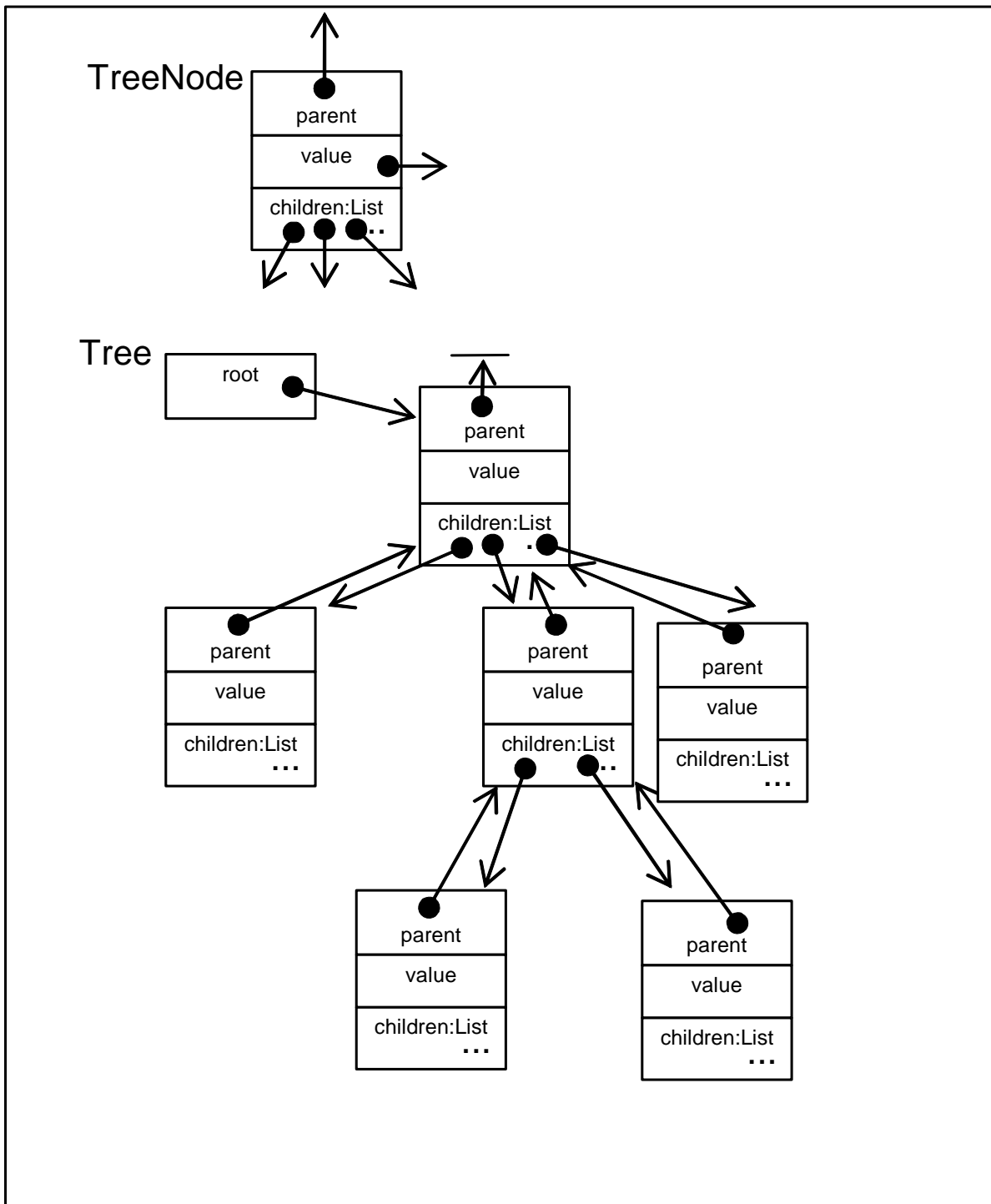


Abbildung 7: Verkettete Struktur für die Darstellung von Binärbäumen

Class LinkedTreeNode

Folgende Klasse `LinkedTreeNode` skizziert eine Implementierung von `TreeNode` mittels einer verketteten Objektstruktur.

```
public class LinkedTreeNode implements TreeNode {

    private java.util.List children;
    private TreeNode parent;
    private Object value;

    /** Konstruktor mit zugehörigem Tree und Wert */
    public LinkedTreeNode(LinkedTree tree, Object value) {
```

```

        children = new java.util.LinkedList();
        parent = null;
        this.value = value;
        this.tree = tree;
    }

    /** liefert den Wert des Knotens */
    public Object value() {
        return value;
    }

    /** liefert das Array aller Nachfolgeknoten */
    public TreeNode[] children() {
        return ((TreeNode[]) (children.toArray(new TreeNode[0])));
    }

    /** liefert den Vaterknoten */
    public TreeNode parent() {
        return parent;
    }

    /** prüft, ob dieser Knoten Wurzel ist */
    public boolean isRoot() {
        return parent == null;
    }

    /** prüft, ob dieser Knoten interner Knoten ist */
    public boolean isInternal() {
        return ! children.isEmpty();
    }

    /** prüft, ob dieser Knoten Blatt ist */
    public boolean isLeaf() {
        return children.isEmpty();
    }

    /** Fügt einen Knoten mit Wert o bei diesem Knoten an */
    public void addChild (Object o) {
        children.add(new LinkedTreeNode(tree, o));
    }
}

```

Class LinkedTree

Folgende Klasse `LinkedTreeNode` skizziert eine Implementierung von `TreeNode` mittels einer verketteten Objektstruktur.

```

public class LinkedTree implements Tree {

    private LinkedTreeNode root;

    public LinkedTree() {
        root = null;
    }

    public LinkedTree(LinkedTreeNode root) {
        this.root = root;
    }
}

```

```

/** liefert den Wurzelknoten des Baums */
public TreeNode root() {
    return root;
}

/** liefert die Nachfolgerknoten des Knotens n */
public TreeNode[] children(TreeNode n) {
    return n.children();
}

/** liefert den Vaterknoten des Knotens n */
public TreeNode parent(TreeNode n) {
    return n.parent();
}

/** prüft, ob Knoten n Wurzel ist */
public boolean isRoot(TreeNode n) {
    return n.isRoot();
}

/** prüft, ob dieser Knoten interner Knoten ist */
public boolean isInternal(TreeNode n) {
    return n.isInternal();
}

/** prüft, ob dieser Knoten Blatt ist */
public boolean isLeaf(TreeNode n) {
    return n.isLeaf();
}

/** liefert die Werte aller im Baum enthaltenen Knoten */
public Object[] elements() {
    Object[] elems;
    ...
    return elems;
}

/** liefert die Anzahl der Elemente im Baum */
public int size() {
    return elements().length;
}

/** liefert die Höhe des Baums */
public int height() {
    int h;
    ...
    return h;
}

// Methoden zum Anfügen von Knoten

/** Definiert eine neue Wurzel mit Wert o */
public void defineRoot(Object o) {
    root = new LinkedTreeNode(this, o);
}

/** Fügt einen neuen Knoten mit Wert o bei Knoten n an */
public void addChild(LinkedTreeNode n, Object o) {
    if (n != null) {

```

```

        n.addChild(o);
    } else {
        System.err.println("Node is null ");
    }
}
}

```

Implementierung Binärbaum als Linkstruktur

Die Implementierung von Binärbäumen unterscheidet sich von der Implementierung von allgemeinen Bäumen dadurch, dass für die Nachfolger statt einer Liste von Nachfolgern zwei Referenzen left und right ausreichen. Dadurch ergibt sich eine Struktur wie in Abbildung8 gezeigt.

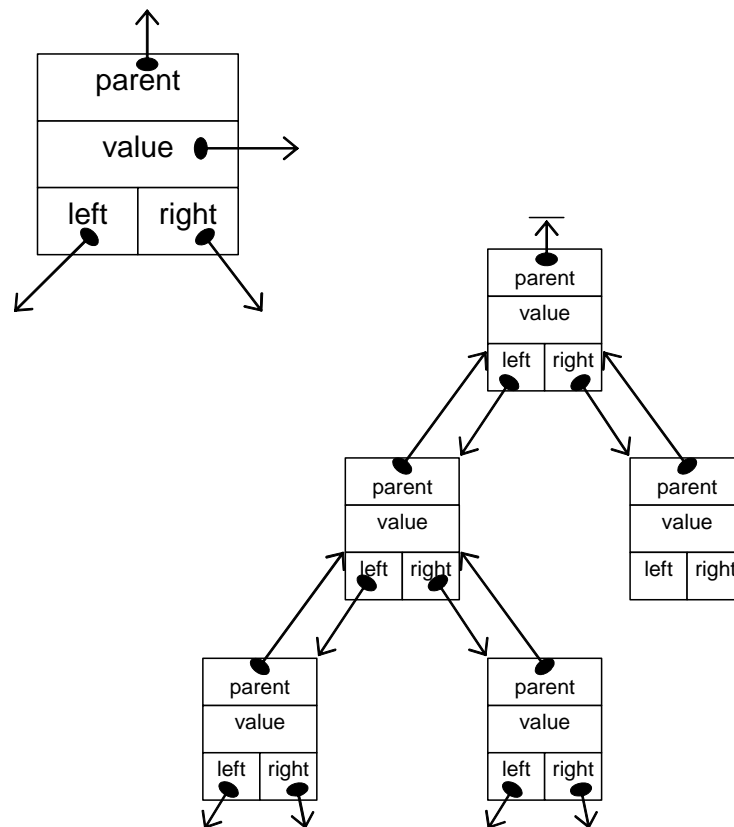


Abbildung8: Verkettete Struktur für die Darstellung von Binärbäumen

Implementierung eines Binärbaums mittels Array

Binärbäume lassen sich auch sehr gut mit Arrays realisieren. Dabei macht man sich die binäre Struktur zunutze. Man geht wie folgt vor:

- Datenstrukturen:
 - Man verwendet ein Array `val ues`, welches die Werte der Knoten enthält
 - Eine Integervariable `last` wird verwendet um die Position des aktuell letzten Elements zu speichern
- Die Wurzel des Baums ist bei Position **1** gespeichert (Position 0 wird nicht verwendet)

- linker Sohn von Knoten an Stelle i steht an Position $2*i$, d.h. der linke Nachfolger der Wurzel befindet sich auf Position 2
- rechter Sohn von Knoten an Stelle i steht an Position $2*i+1$, d.h. der rechte Nachfolger der Wurzel befindet sich auf Position 3
- Vater von Knoten an Stelle i steht an Position $i / 2$ (Integer-Division)

Abbildung 9 zeigt die Anordnung der Elemente für einen Beispielbaum. Vergewissern Sie sich, dass man durch die Positionsrechnungen wie oben angegeben auf die Nachfolger und Vorgänger eines Knotens zugreifen kann.

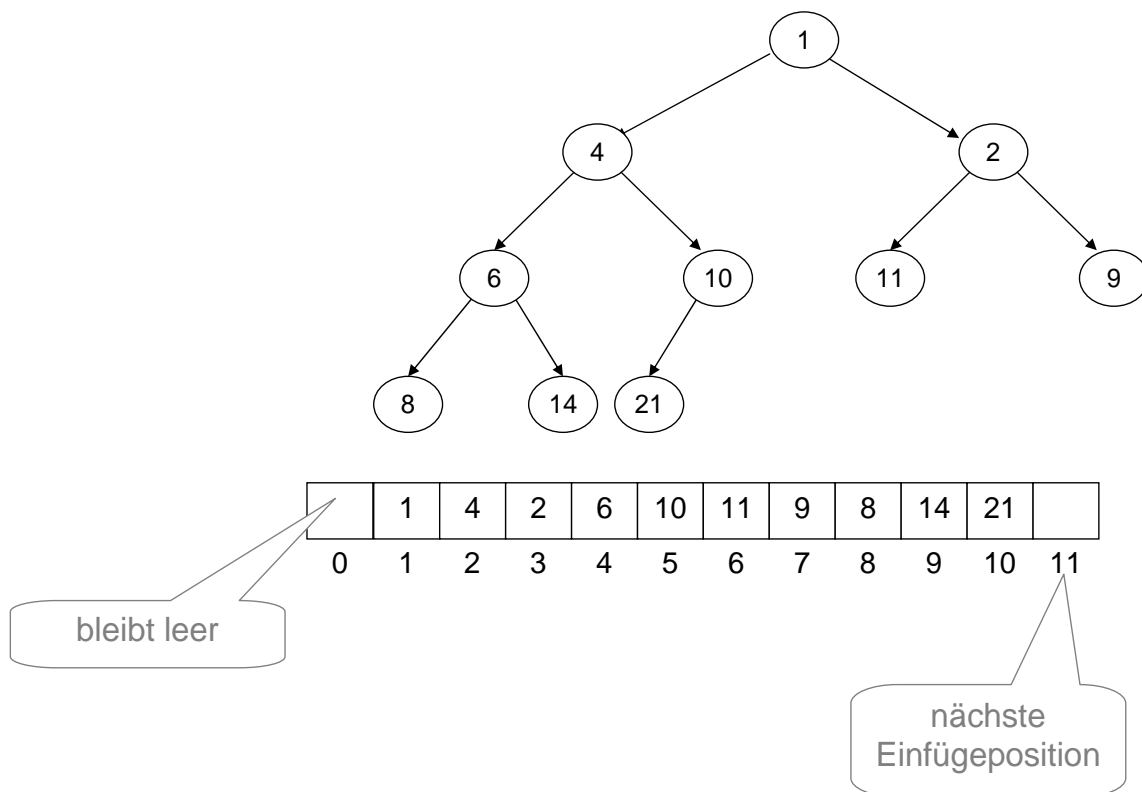


Abbildung 9: Binärbaum in Array

Datenstrukturen

Für die Implementierung von ADT mit Arrays braucht man folgende Datenstrukturen

- Werte in Array val ues (ausreichender Größe)
- Index last zeigt auf die Position im Array, wo letzter Wert zu speichern wurde (initial 0).
- Knoten sind nicht permanent vorhanden, sondern werden über Zugriffe dynamisch erzeugt
 - Knoten beinhalten Position im Array
 - erlauben Zugriff auf Wert und linken und rechten Knoten

Implementierung

In der Übungsstunde!

4.6 Zusammenfassung

Bäume sind eine wichtige Datenstruktur in der Informatik

- als natürliche Datenstruktur für gewisse Problemfälle
- als effiziente Datenstrukturen

Viele Algorithmen lassen sich auf Ebene der ADT (ohne Implementierung) formulieren

- preorder-, postorder-, inorder-Algorithmen

Implementierung von Bäumen

- mittels verketteter Objektstrukturen
- Binärbäume mittels Arrays mit Indexrechnungen für den Zugriff auf Knoten und Werte

5 Prioritätswarteschlangen und Heaps

5.1 ADT PriorityQueue

Idee von PriorityQueue ist die Reihung der Elemente nach einer Priorität (Ordnung, Sortierkriterium)

Zu speichernden Elemente haben somit 2 Aspekte:

- Inhalt (Objekte)
- Ordnung (\leq)

Realisierung der Ordnung:

- eigener Schlüssel, der eine Ordnung definiert
- Objekte implementieren Comparable
- eigener Comparator, der die Objekte vergleicht (siehe `java.util.Comparator`)

ADT PriorityQueue:

PriorityQueue ist ein ADT, der eine Menge von Elementen nach einem Sortierkriterium sortiert verwaltet. Er bietet einen schnellen Zugriff auf das erste (kleinste) Element:

- Fügt ein Objekt nach dem Sortierkriterium key ein: `insert(obj, key)`
- Zugriff auf das kleinste Element: `findMin()`
- Löschen des kleinsten Elements: `deleteMin(min)`
- Prüfen, ob Queue leer ist: `isEmpty()`

Abstrakte Datentyp PriorityQueue von Objects mit Key in Funktionsschreibweise

```
abstract-data-type PriorityQueue
  create:      → PriorityQueue
  insert: Object × Key × PriorityQueue → PriorityQueue
  findMin: PriorityQueue → Object
  deleteMin: PriorityQueue → PriorityQueue
  isEmpty: PriorityQueue → boolean
end PriorityQueue
```

Interface Item mit key-value-Pair

Das folgende Klasse Item definiert eine Struktur mit order, welcher vom Typ Comparable e ist, und val ue, welcher vom generischen Typ E ist. Diese Struktur wird für die Elemente in der Pri ori tyQueue<E> verwendet.

```
public class Item<O extends Comparable<O>, E> {
    O order;
    E value;

    public Item(O order, E value) {
        this.order = order;
        this.value = value;
    }

    O order() {
        return order;
    }

    E value() {
        return value;
    }
}
```

Interface Pri ori tyQueue:

```
public interface PriorityQueue<O,E> {
    /**
     * Fuegt ein Fuegt einen Wert unter Ordnung order ein
     */
    void insert(O order, E elem);

    /**
     * Zugriff auf das kleinste Element
     */
    E findMin();

    /**
     * Loescht das kleinste Element
     */
    void deleteMin();

    /**
     * Prüft, ob Queue leer ist
     */
    boolean isEmpty();
}
```

5.2 LinkedList-Realisierung von PriorityQueue

Implementierung einer Pri ori tyQueue kann mit einer verketteter Liste erfolgen. Die Elemente (vom Typ I tem) werden aufsteigend sortiert eingefügt, sodass an vorderster Stelle sich immer das kleinste befindet.

```
public class LinkedPriorityQueue<O
    extends Comparable<O>,E>
    implements PriorityQueue<O, E> {

    private class Node<O extends Comparable<O>, E> {

        Item<O, E> item;
        Node<O, E> next;

        public Node(O order, E value) {
            super();
            this.item = new Item<O,E>(order, value);
        }
    }
}
```

```

    }
} // Node

Node<O, E> head;

public LinkedPriorityQueue () {
    head = null;
}

public void insert(O order, E item) {
    Node<O, E> prev, curr;
    curr = head;
    prev = null;
    while (curr != null) {
        if (order.compareTo(curr.item.order()) > 0) {
            prev = curr;
            curr = curr.next;
        } else {
            break;
        }
    }
    Node<O,E> n = new Node<O,E>(order, item);
    n.next = curr;
    if (prev == null) {
        head = n;
    } else {
        prev.next = n;
    }
}

public E findMin() {
    if (! isEmpty())
        return head.item.value();
    else
        return null;
}

public void deleteMin() {
    if (! isEmpty()) {
        head = head.next;
    }
}

public boolean isEmpty() {
    return head == null;
}
}

```

Komplexität:

Bei der obigen Implementierung ergibt sich folgende Laufzeitkomplexität für die einzelnen Operationen:

- Zugriff auf minimales Element: $O(1)$
- Löschen des minimalen Elements: $O(1)$
- Einfügen eines neuen Elements: $O(n)$

Zugriff und Löschen ist mit einer Komplexität $O(1)$ optimal, das Einfügen mit einer Komplexität $O(n)$ schlecht.

5.3 Heap-Realisierung von PriorityQueue

Eine wesentlich bessere Implementierung von PriorityQueues lässt sich mittels eines sogenannten Heaps erreichen. Hier ist die Komplexität

- des Zugriffs auf das erste Element auch $O(1)$,
- Löschen des ersten $O(\text{Id}(N))$
- aber auch das Einfügen in den Heap $O(\text{Id}(N))$

Definition Heap

Ein Heap (BinaryHeap) ist ein binärer Baum mit folgenden Eigenschaften:

- **Ordnung:** Die Elemente im Baum sind geordnet, dass das Element beim Vater immer kleiner als das Element beim Sohn ist
- **Vollständigkeit und Ausgeglichenheit:** Ein Heap muss ausgeglichen und die unterste Ebene von links voll aufgefüllt sein

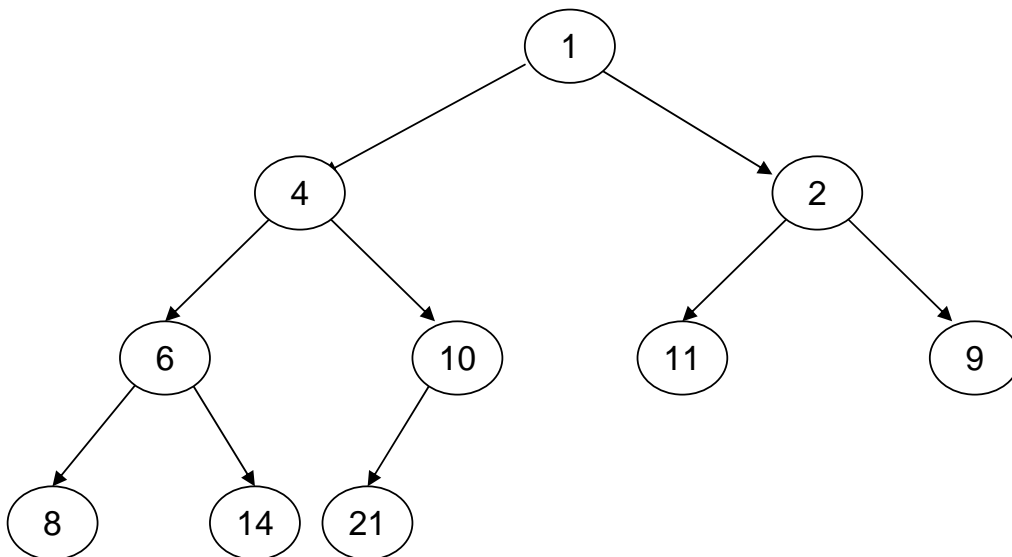


Abbildung 10: Heap

Operationen bei Heap

Einfüge-Operation bei Heap

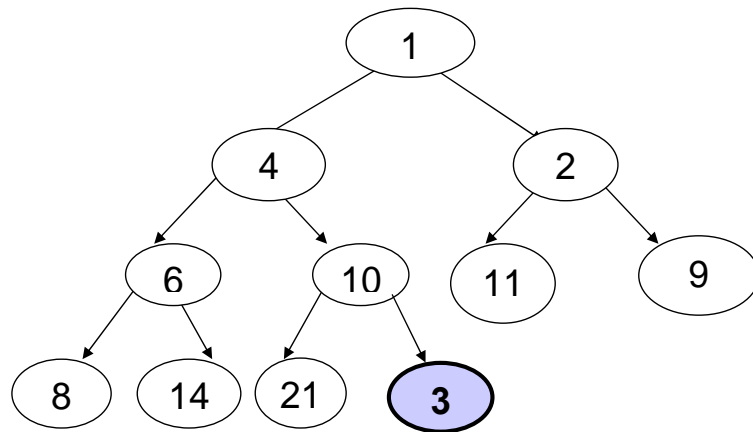
- zuerst wird ein Knoten an der nächsten Position (entsprechend der Vollständigkeit und Ausgeglichenheit) erzeugt und der neue Wert gespeichert
- Dann wird die Ordnungseigenschaft wieder hergestellt, indem solange der Wert des Knotens mit dem Wert des Vaters vertauscht wird, bis Ordnungseigenschaft erfüllt ist (upHeap)

Einfüge-Operation Pseudocode:

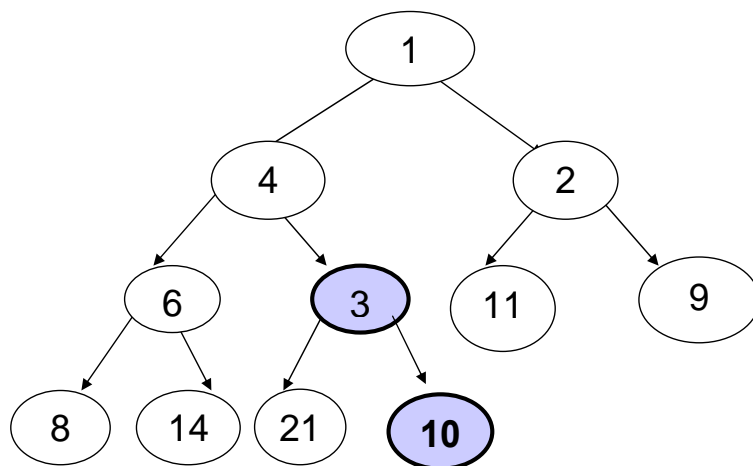
```
void insert(O order, E value){
    // Erzeugen des nächsten Knotens
    BinaryTreeNode node = createNextNode(new Item<O,E>(order, value));
    // Ordnung wieder herstellen
    upHeap(node);
}

void upHeap(BinaryTreeNode node) {
    if (!node.isRoot && node.item.order() <= node.parent.item.order())
    {
        swapValues(node, node.parent)
        upHeap(node.parent())
    }
}

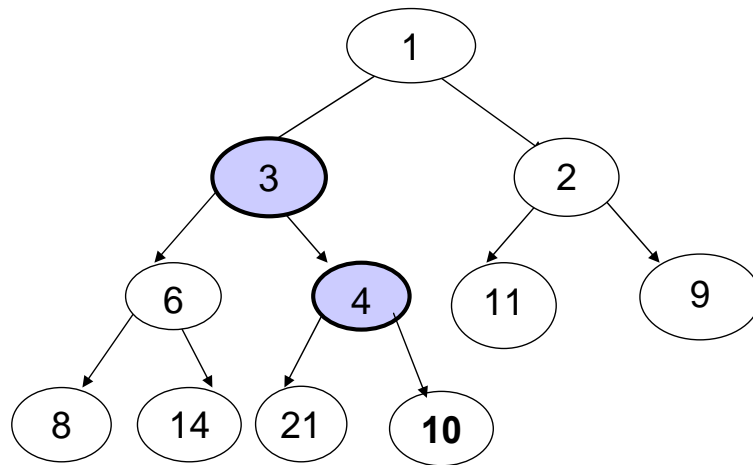
void swapValues(BinaryTreeNode node1, BinaryTreeNode node2) {
    temp = node1.item
    node1.item = node2.item
    node2.item = temp
}
```



(a) neuen Knoten mit Wert 3 einfügen



(b) upHeap: vertausche 10 und 3



(c) upHeap: vertausche 3 und 4

Abbildung 11: Einfüge-Operationen

Entnehmen des minimalen Elements beim Heap

- Minimaler Wert ist bei Wurzel gespeichert; dieser Wert definiert den Rückgabewert
- Wert an Wurzel wird durch Wert des letzten Knotens ersetzt (zuletzt angefügter Knoten) und letzter Knoten wird gelöscht
- Dann wird die Ordnungseigenschaft wieder hergestellt , indem solange der Wert des Knotens mit dem Wert des Sohnes mit kleinerem Wert vertauscht wird, bis Ordnungseigenschaft wieder erfüllt ist (downHeap)

Lösche Min-Operation Pseudocode:

```

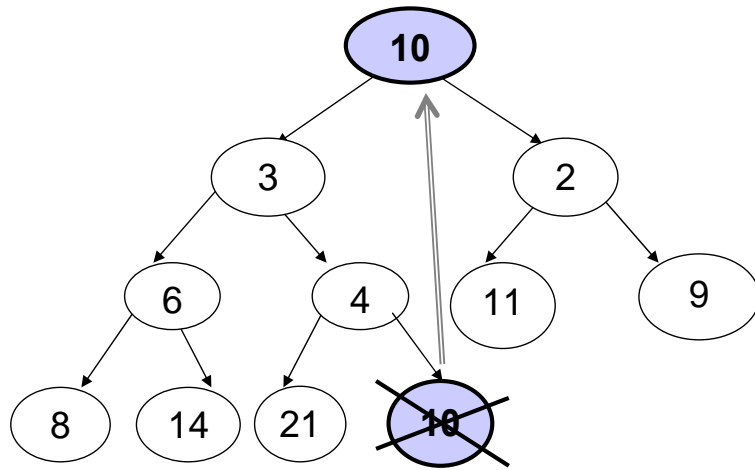
void deleteMin() {
    returnValue = root.item
    last = lastElementInHeap()
    root.item = last.item
    delete last;
    downHeap(root)
}

```

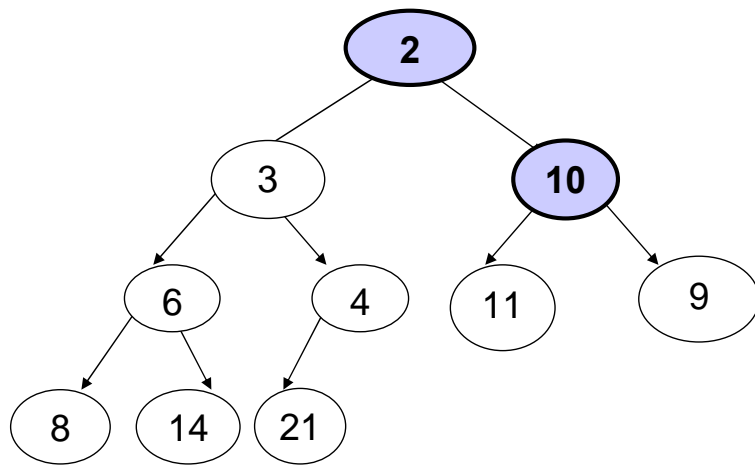
```

void downHeap(BinaryTreeNode node) {
    // Suchen des Sohnes mit minimalen Wert
    minSon = findMinSon(node)
    if (minSon existiert && minSon.item.order <= node.item.order) {
        // Vertauschen der Werte
        swapValues(node, minSon)
        // Fortsetzen der downHeap-Operation
        downHeap(minSon)
    }
}

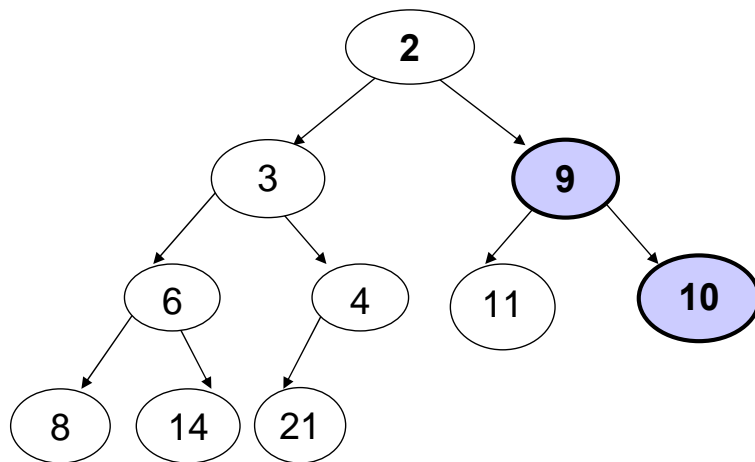
```



(a) Wert des letzten Knotens in Wurzel speichern; letzten Knoten löschen



(b) downHeap: vertausche 2 und 10



(c) downHeap: vertausche 9 und 10

Abbildung 12: Einfüge-Operationen

Implementierung Heap (Version mit Array)

Folgend ist eine Implementierung von BinaryHeap aufbauend auf Positional BinaryTree angedeutet (siehe Übungsstunde zu Implementierung von Positional BinaryTree).

```
public class Heap<O extends Comparable<O>, E> implements PriorityQueue<O, E> {  
  
    public static final int MAX_ELEMS = 1000;  
    Item<O,E>[] values = (Item<O,E>[]) new Item[MAX_ELEMS];  
    int last;  
  
    public void insert(O order, E value) {  
        last++;  
        values[last] = new Item<O, E>(order, value);  
        upHeap(last);  
    }  
  
    public E findMin() {  
        return values[1].value;  
    }  
  
    public void deleteMin() {  
        values[1] = values[last];  
        values[last] = null;  
        last--;  
        downHeap(1);  
    }  
  
    public boolean isEmpty() {  
        return last == 0;  
    }  
  
    private void upHeap(int pos) {  
        int parent = pos / 2;  
        if (parent > 0 && isSmaller(pos, parent)) {  
            swapValues(pos, parent);  
            upHeap(parent);  
        }  
    }  
  
    private void downHeap(int pos) {  
        int minSon = findMinSon(pos);  
        if (minSon > 0 && isSmaller(minSon, pos)) {  
            swapValues(pos, minSon);  
            downHeap(minSon);  
        }  
    }  
  
    ...  
}
```

5.4 Zusammenfassung

PriorityQueue ist ein ADT zur Verwaltung von Elementen mit effizientem Zugriff auf das Minimum

- LinkedLi st-Implementierung mit $O(N)$ Komplexität für Einfügen
- Heap-Implementierung mit $O(\log(N))$ Komplexität für Einfügen, was bei vielen Elementen eine wesentliche Effizienzsteigerung bedeutet (siehe dazu Abschnitt 2.4).

6 ADT Dictionary und binäre Suchbäume

6.1 ADT Dictionary

Die Verwaltung einer Menge von Elementen und der selektive Zugriff auf die Elemente über einen eindeutigen Schlüssel, ist ein wichtiger ADT, der üblicherweise mit `Dictionary` (oder auch `Map`) bezeichnet wird.

Der ADT `Dictionary` erlaubt die Ablage und Suche von Werten nach einem Schlüssel. Dabei wollen wir annehmen, dass der Schlüssel beliebig sein kann, aber eindeutig sein und eine Ordnung definieren muss (die durch die Implementierung des Interfaces `java.util.Comparable` realisiert sein soll).

ADT Dictionary

`Dictionary` ist ein ADT mit folgenden Operationen:

- Einfügen eines Objekts `obj` nach dem Sortierkriterium `key`: `insert(obj, key)`
- Zugriff das Element für einen gegebenen Schlüssel: `Object search(key)`
- Löschen des Elements mit Schlüssel `key`: `remove(key)`
- Anzahl der Elemente der Menge: `int size()`

Abstrakte Datentyp `Dictionary` in Funktionsschreibweise

```
abstract-data-type Dictionary
  create:      → Dictionary
  insert: Object × Key × Dictionary → Dictionary
  search: Dictionary × Key → Object
  remove: Dictionary × Key → Dictionary
  size: Dictionary → int
end Dictionary
```

Interface Dictionary

Ein Interface für ADT `Dictionary` könnte folgend aussehen.

```
/** Interface Dictionary für Item-Objekte */
public interface Dictionary {

    /**
     * fügt ein Objekt value unter Schlüssel key ein
     */
    void insert(Comparable key, Object value);

    /**
     * Zugriff auf das Element mit Schlüssel key
     */
}
```

```

Object search(Comparable key);

/**
 * löscht das Element mit Schlüssel key
 */
void remove(Comparable key);

/**
 * liefert die Anzahl der Elemente
 */
int size();
}

```

6.2 Binäre Suche

Bei Implementierungen des ADT `Dictionary` ist der Zugriff auf ein Element für einen gegebenen Schlüssel die wichtigste Operation, die möglichst schnell implementiert werden muss. Implementierungen von `Dictionary` machen sich daher die Sortierreihenfolge des Schlüssels zu nutze, d.h. die Elemente werden nach dem Schlüssel sortiert verwaltet und nach dem binären Suchverfahren mit einer Komplexität von $O(\log(N))$ gesucht.

Zur Wiederholung, die binäre Suche funktioniert wie folgt:

- gegeben ein zu suchender Wert
- in der gesamten Menge betrachtet man das Element in der Mitte
- ist das mittlere Element gleich dem gesuchten Wert, dann gefunden
- ist das mittlere größer als der gesuchte Wert, sucht man nur in der unteren Hälfte weiter (die nur mehr halb so groß ist)
- ist das mittlere kleiner als der gesuchte Wert, sucht man nur in der oberen Hälfte der Menge weiter (die nur mehr halb so groß ist)

Beim binären Suchen kann man daher in jedem Schritt die Größe der zu durchsuchenden Menge halbieren!!

Rekursiver Algorithmus für die binäre Suche in einem sortierten Feld:

Folgendes rekursive Programm sucht in einem Array von Items (= Paare von key und value), wobei die Elemente im Array nach key aufsteigend sortiert sind, nach dem binären Suchverfahren.

Anmerkung: Rückgabewert ist das Element (value von Item), null wenn nicht gefunden.

```

public static Object binarySearch
(Item items[], int from, int to, Comparable key) {
    if (from > to) {
        return null;
    }
    int m = (from + to) / 2;
    int comparison = items[m].key().compareTo(key);
    if (comparison == 0) { // found
        return items[m].value();
    } else if (comparison < 0) { // Suche im unteren Bereich
        return binarySearch(items, from, m-1, key);
    }
}

```

```

    } else { // Suche im oberen Bereich
        return binarySearch (items, m+1, to, key);
    }
}

```

Komplexität des Algorithmus ist **$O(\log(N))$** für die Suche. Einfügen bei einer Array-Implementierung ist aber sehr aufwendig. Man verwendet daher für die Realisierung von Dictionary mit binären Suche binäre Suchbäume.

6.3 Binäre Suchbäume

Definition binärer Suchbaum:

Ein binärer Suchbaum ist ein binärer Baum, mit folgenden Eigenschaften

- für die Werte bei den Knoten ist ein Sortierkriterium definiert (z.B. `key()` von `Items`)
- es gilt für jeden Knoten,
 - dass die Werte aller Knoten im **linken** Unterbaum **kleiner gleich** als der Wert beim Knoten sind und
 - die Werte aller Knoten im **rechten** Unterbaum **größer** als der Wert beim Knoten sind
 - Das heißt die Werte bei den Knoten sind von links nach rechts geordnet.

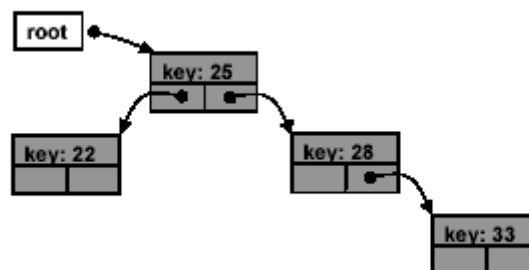


Abbildung 13: Binärer Suchbaum

Implementierung eines binären Suchbaums

Eine Implementierung des ADT Dictionary mittels eines binären Suchbaums ist einfach und effizient (siehe dazu Skriptum Algorithmen 1 und Sedgewick. Algorithms in Java).

Problem beim binären Suchbaum: Ausgeglichenheit

Ein binärer Suchbaum wie oben dargestellt funktioniert gut, wenn durch die Reihenfolge des Einfügens der Elemente der Baum annähernd ausgeglichen bleibt.

Beispiel:

Werden die Elemente mit den Keys A, B, C, D, E, F, G in der folgenden Reihenfolge eingefügt

D, B, F, A, C, E, G

Dann bleibt der Baum schön ausgeglichen, wie in Abbildung 14 gezeigt.

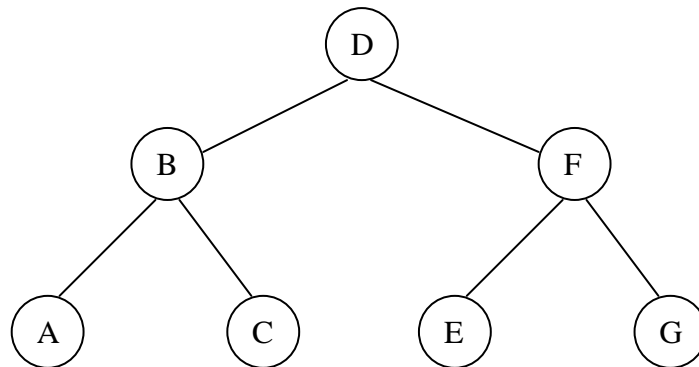


Abbildung 14: ausgeglichener Suchbaum

Werden aber die Elemente mit den Keys A, B, C, D, E, F, G in der Reihenfolge Ihrer Sortierung eingefügt

A, B, C, D, E, F, G

dann entsteht ein völlig unausgeglichener Baum wie in Abbildung 15 gezeigt.

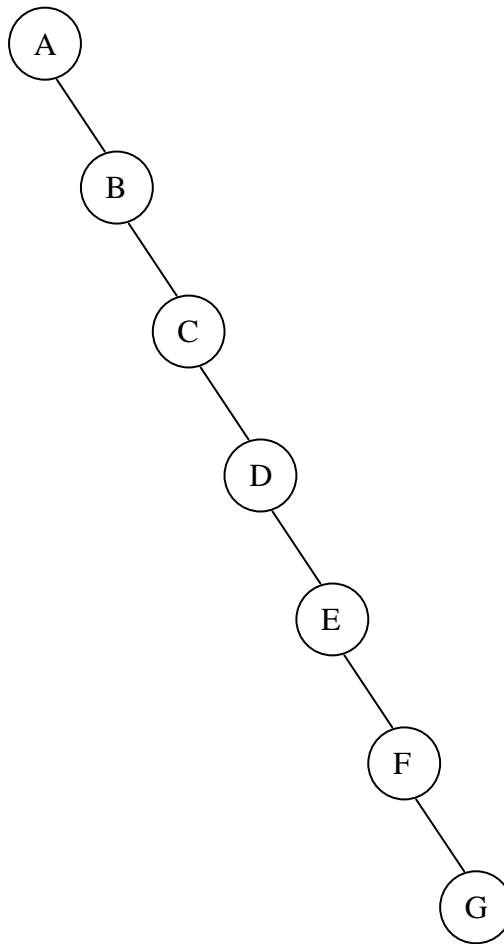


Abbildung 15: Unausgeglichener Suchbaum

Frage: Welche Komplexität hat bei einem derartigen Suchbaum die Suche?

In der Praxis werden binäre Suchbäume unausgeglichen. Abhilfe dazu bieten die ausgeglichenen Bäume wie die Rot-Schwarz-Bäume (Red-Black-Trees), 2-3-4-Bäume etc., die den Baum durch spezielle Mechanismen beim Einfügen und Löschen möglichst ausgeglichen halten (mehr dazu siehe Sedgwick: Algorithms in Java, Kapitel 13).

6.4 Zusammenfassung

- Suche in großen Datenbeständen ist ein wichtiges Verfahren, das man oft braucht und effizient gelöst werden muss.
- Array und LinkedList-Implementierungen sind effizient in der Suche aber ineffizient beim Einfügen
- Binäre Suchverfahren haben eine Komplexität von $O(\log(N))$
- Binäre Suchbäume haben auch für das Einfügen eine (theoretische) Komplexität von $O(\log(N))$
- Binäre Suchbäume können aber unausgeglichen werden, was die Komplexität der Operationen erhöht. Abhilfe dafür sind so genannte ausgeglichene Bäume, die die Ausgeglichenheit durch spezielle Mechanismen aufrecht erhalten.
- Bei geometrischen Verfahren spielen Suchbäume eine große Rolle für die Bereichssuche. Allerdings braucht man dazu Suchbäume im 2-dimensionalen

(oder auch 3-dimensionalen) Räumen. Wir werden daher beim Kapitel über geometrischen Algorithmen Suchbäume im 2-dimensionalen Raum, so genannte

- 2-D Bäume,
- kennen lernen.

7 Graphen

7.1 Motivation



Abbildung 16: Beispielanwendung Graphen

Graphen sind eine mathematische Abstraktion sehr allgemeiner Natur und mit vielen Anwendungen in der Informatik. Graphen können darstellen

- Objekte und
- Zusammenhänge zwischen diesen Objekten

Anwendungsbereiche sind sehr breit und umfassen z.B.:

- Flugpläne
- Straßenkarten
- Aufgabenlisten
- Fertigungsstraßen
- Netzpläne
- Operationsreihenfolgen
- uvm.

In der Geoinformatik sind Graphen das Grundkonzept für die Darstellung von Wegnetzen, Straßennetzen, Fahrtenplänen und dergleichen, können aber auch als Abstraktion für geographische Zusammenhänge verwendet werden.

Beispiel: Informationsstruktur um zu speichern, welche Regionen miteinander verbunden sind; Regionen bilden die Knoten, die Zusammenhänge der Regionen die Kanten.

Für Graphen existieren eine Vielzahl von theoretischen Konzepten und Algorithmen, die breit eingesetzt werden können. Ein Beispiel ist die Wegsuche in einem Graphen. Für diese gibt es unterschiedliche Verfahren. Eingesetzt wird diese nicht nur bei der klassischen Suche einer Route von einem Ort zum anderen, sondern zum Beispiel auch bei der Planung für die Erfüllung einer Aufgabe, wobei die Aufgabe in eine Folge von Operationen aufgelöst wird.

7.2 Definition Graph

Graph

Ein Graph ist ein Tupel

$$G = (V, E)$$

mit

V ist eine Menge von Knoten (*vertices* oder *nodes*)

$E \subseteq V \times V$ ist die Menge von Kanten (*edges*)

Beispielgraph:

```
G = (V, E)
  mit
V = {v1, v2, v3, v4, v5}
E = {(v1, v3), (v1, v2), (v3, v4),
      (v2, v4), (v2, v5), (v4, v5)}
```

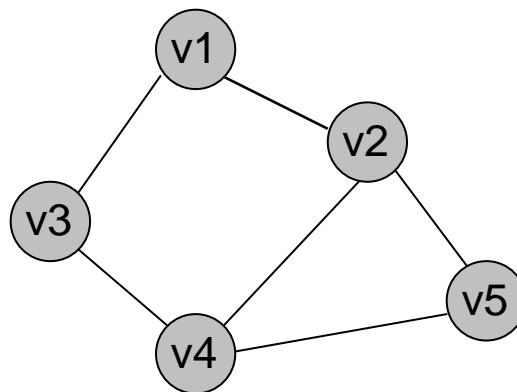


Abbildung 17: Beispielgraph

Gerichtete Graphen (Digraph)

Eine gerichtete Kante $(v1, v2)$ ist ein geordnetes Paar – geschrieben oft

$\langle v1, v2 \rangle$

d.h. verbindet die beiden Knoten in einer bestimmten Richtung.

Bei einem gerichteten Graph – oder

Digraph

genannt – sind die Kanten gerichtet.

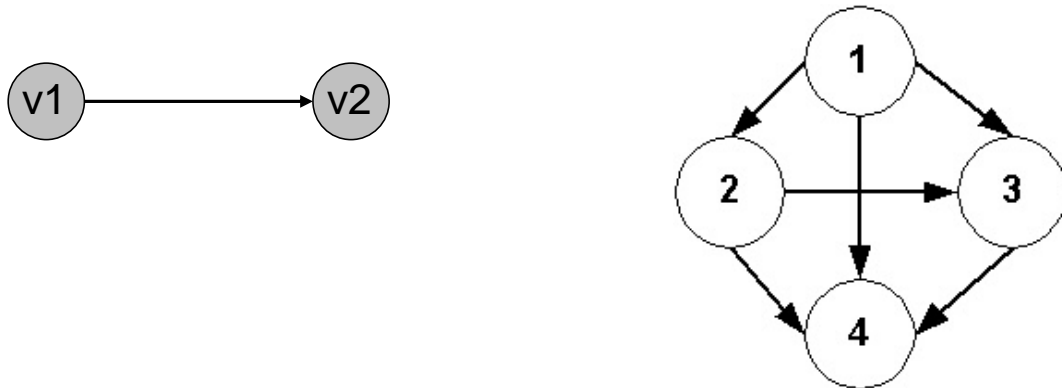


Abbildung 18: Gerichtete Kante und gerichteter Graph

Beispiele:

- Produktionsplan (Abfolge der Produktionsschritte)
- Fahrplan
- Wegbeschreibung in einer Stadt mit Einbahnstraßen
- Baum: ein Baum ist ein gerichteter Graph ohne Zyklen und zusammenhängend

Ungerichtete Graphen

In einem ungerichteten Graphen ist jede Kante (v_1, v_2) ein ungeordnetes Paar (v_1, v_2) , wobei $v_1 \neq v_2$, die diese beiden Knoten verbindet ohne eine Richtung vorzuschreiben.

Beispiele: Wanderkarte

- Knoten = Ausgangs- und Zielpunkte, Gipfel, Hütten, Wegkreuzungen, ...
- Kanten = Wege zwischen den Knoten

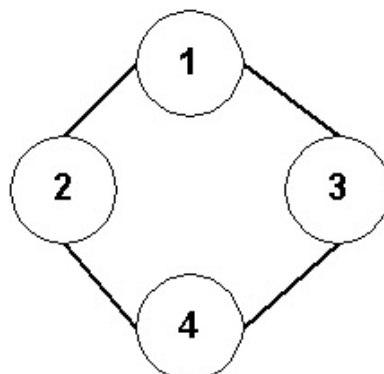


Abbildung 19: Ungerichteter Graph

! Ungerichteter Graph kann als „Spezialfall“ eines gerichteten Graph gesehen werden, wobei zu jeder (gerichteten) Kante $\langle v1, v2 \rangle$ auch eine Kante $\langle v2, v1 \rangle$ existiert.

Gewichtete Graphen

Bei gewichteten Graphen werden den Kanten *Gewichte* zugeordnet, die "Kosten"

der Kanten darstellen. Damit bedeutet das Gewicht bei einer Kante von A nach B, ganz allgemein die Kosten um von A nach B zu kommen.

Beispiele

- Entfernungen der Wege
- Zeitdauern für eine Operation

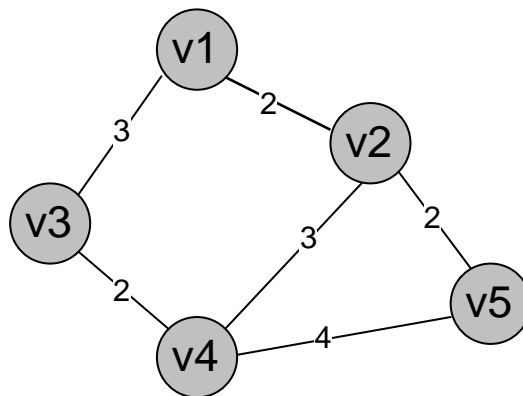


Abbildung 20: Gewichteter Graph

Beispiel: Straßenkarte

- Knoten = Orte
- Kanten = Straßen
- Gewichte = Entfernungen in km oder Fahrzeit zw. den Orten

Netzwerke

Gerichtete, gewichtete Graphen werden auch als Netzwerke (Networks) bezeichnet.

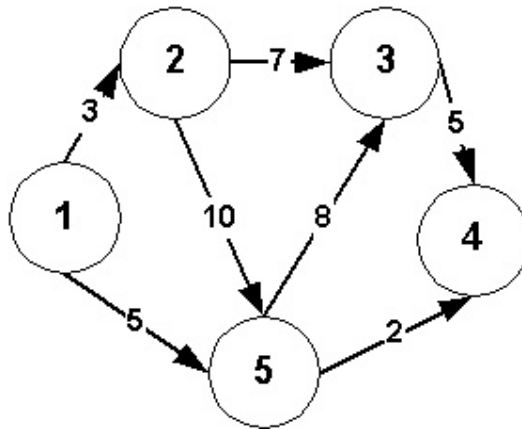


Abbildung 21: Netzwerk

Beispiele

- Projektplan, wobei die Knoten die Meilensteine darstellen, die Kanten die durchzuführenden Tätigkeiten und die Gewichte die Zeit für die Tätigkeiten.
- Öl-Pipelines, wobei die Knoten Kreuzungspunkte darstellen, die Kanten die Leitungen und die Gewichte die Kapazität der Leitungen.
- ...

7.3 Wichtige Begriffe

Adjazente Knoten und inzidente Kanten

adjazent:

Zwei Knoten v_1 und v_2 heißen *adjazent* zueinander, wenn sie durch eine Kante (v_1, v_2) verbunden sind

inzident:

Kante (v_1, v_2) , die zwei adjazente Knoten v_1 und v_2 verbindet, heißt *inzident* zu diesen Knoten.

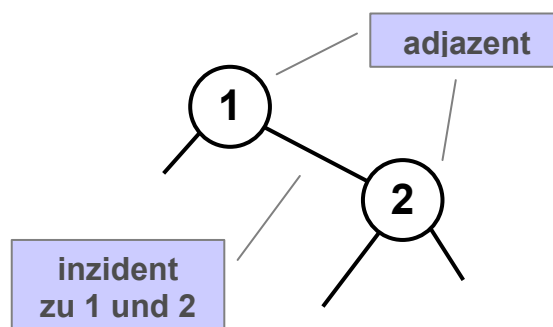


Abbildung 22: Adjazent und inzident

Grad eines Knotens

Grad :

Der Grad $\deg(v)$ eines Knoten v entspricht der Anzahl der Kanten, die diesen Knoten mit anderen Knoten verbindet.

Eingangsgrad:

Der Eingangsgrad eines Knoten v bei gerichteten Graphen entspricht der Anzahl der einmündenden Kanten

Ausgangsgrad:

Der Ausgangsgrad eines Knoten v bei gerichteten Graphen entspricht der Anzahl der wegführenden Kanten

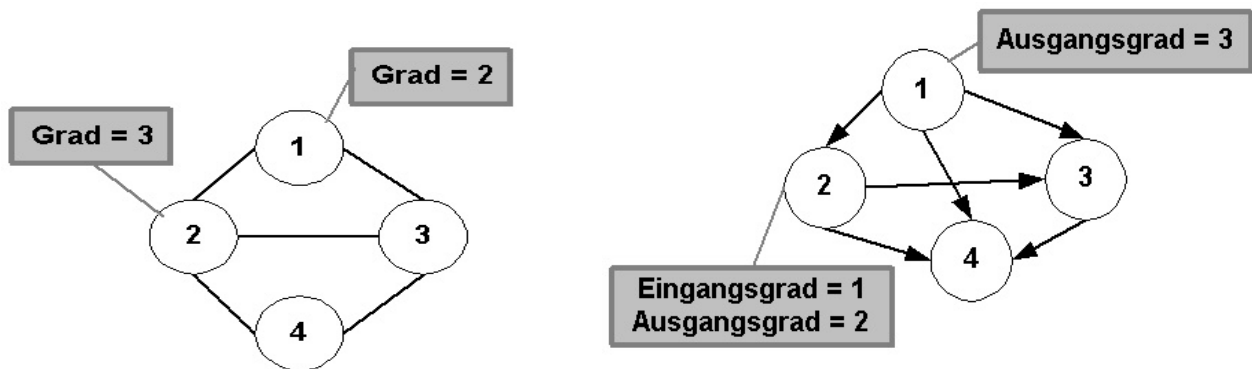


Abbildung 23: Grad eines Knotens

Pfade und Wege

Pfad oder Weg

Ein **Pfad** oder **Weg** ist eine Folge von Knoten v_1, v_2, \dots, v_k in der jeweils aufeinander folgende Knoten v_i und v_{i+1} adjazent zueinander sind.

Beispiele von Pfaden (vergleiche Abbildung 24):

v_1, v_2, v_4, v_5

v_3, v_4, v_2, v_5, v_2

kein Pfad: v_1, v_4, v_2, v_3

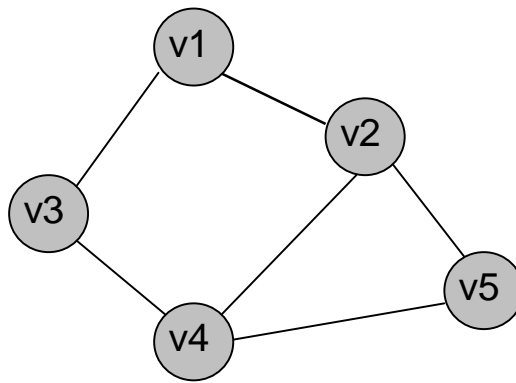


Abbildung 24: Pfade und Wege

Weglänge:

Mit **Weglänge** wird die Anzahl der Kanten auf einem Weg bezeichnet.

Beispiel Weglänge: Weg v1, v2, v4, v5 hat Länge = 3

Einfacher Weg:

In einem **einfachen Weg** kommt kein Knoten mehr als einmal vor.

Beispiel einfacher Weg: v1, v2, v4, v5

Zyklischer Weg:

Ein **zyklischer Weg** (kurz: **Zyklus**) ist ein einfacher Weg mit der Ausnahme, dass der erste Knoten im Pfad identisch mit dem letzten Knoten im Pfad ist.

Ein Graph ohne einen Zyklus bezeichnet man als zykliefrei.

Beispiel zyklischer Weg: v1, v2, v4, v3, v1

Kosten eines Weges

Die Summe der Gewichte aller Kanten in einem Weg stellen naturgemäß die Kosten für den Weg dar.

Weitere Begriffe

zusammenhängend:

Ein Graph heißt zusammenhängend, wenn von jedem Knoten zu jedem anderen Knoten ein Weg existiert

Komponente:

Ein nicht zusammenhängender Graph besteht aus mehreren Komponenten. Ein zusammenhängender Graph besteht genau aus einer Komponente.

stark zusammenhängend:

Ein Graph heißt stark zusammenhängend, wenn in einem gerichteten Graph zwischen 2 beliebigen Knoten in jede Richtung ein Weg führt.

vollständig:

Ein Graph, bei dem für alle Knoten $v, v' \in V$ eine Kante $(v, v') \in E$ existiert, heißt vollständig. Ein solcher Graph enthält $|V| \cdot (|V| - 1) / 2$ Kanten.

dicht:

Ein Graph, dem nur wenige Kanten zu einem vollständigen Graph fehlen, wird dicht genannt.

licht:

Ein Graph, der vergleichsweise wenige Kanten hat, wird licht genannt.

Subgraphen

Subgraph:

Ein Subgraph ist eine Teilmenge von Knoten und Kanten eines Graphen, die wiederum einen Graphen bilden.



Abbildung 25: Subgraph

Aufspannender Baum (Spannbaum, *Spanning Tree*):

Ein Subgraph eines Graphen, der alle Knoten enthält, aber nur so viele Kanten, dass er einen Baum bildet, nennt man Spannbaum oder aufspannenden Baum (englisch: *Spanning Tree*)

Für einen **aufspannender Baum** gilt: bei Ausfall einer einzigen Kante ist der Subgraph nicht mehr verbunden.

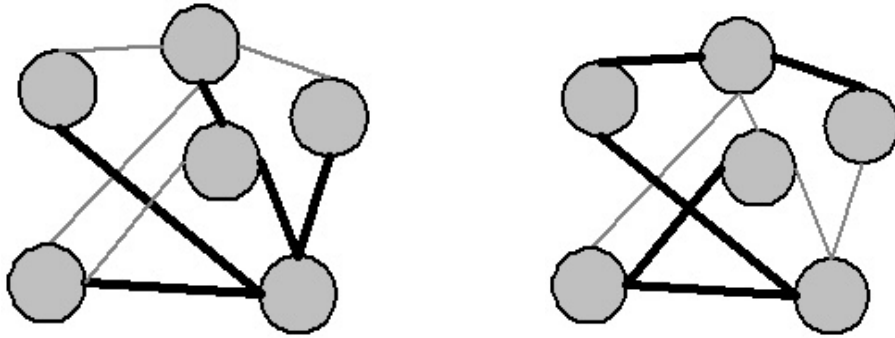


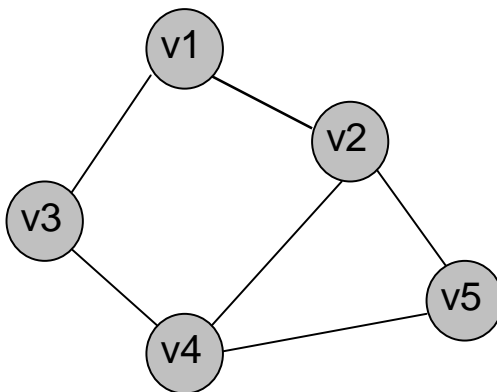
Abbildung 26: Spannbäume

7.4 Darstellungsformen für Graphen

Darstellungsform: Adjazenzmatrix

Adjazenzmatrix boolean $A[n, n]$ mit $A[v_i, v_j]$ ist

- true, wenn eine Kante (v_i, v_j) existiert,
- false sonst



$A[i,j]$	v1	v2	v3	v4	v5
v1	F	T	T	F	F
v2	T	F	F	T	T
v3	T	F	F	T	F
v4	F	T	T	F	T
v5	F	T	F	T	F

Abbildung 27: Adjazenzmatrix

! Adjazenzmatrix ist symmetrisch für ungerichtete Graphen

Datenstruktur Adjazenzmatrix

Folgende Datenstruktur stellt eine Adjazenzmatrix dar, wobei die Knoten in einem Array gespeichert werden, die Adjazenzmatrix selbst durch eine Matrix von bool'schen Werten repräsentiert ist.

Datenstruktur:

```
class Node {
    Object value;
    ...
}
```

```

class AdjMatrix {
    int maxNodes;
    Node[] = new Node[maxNodes];
    boolean[][] adjMatrix = new boolean[maxNodes][maxNodes];
    ...
}

```

Speicherbedarf:

Speicherbedarf für Adjazenzmatrix ist $O(|V|^2)$ und somit unabhängig von der Anzahl der Kanten. Daher ist Adjazenzmatrix für dichte Graphen gut geeignet, für lichte Graphen weniger geeignet.

Darstellungsform: Adjazenzmatrix mit beliebigen Werten

Adjazenzmatrix kann keine Kantengewichte oder andere Informationen bei Kanten speichern. Daher wird oft mit einer Erweiterung der Adjazenzmatrix mit Integer-Werten oder Kantenobjekten gearbeitet.

Statt bool'schen Werte wird in der Matrix gespeichert:

- **Integer:** für gewichtete Graphen verwendet, wobei der im Matrixelement gespeicherte Integerwert die Gewichte der Kante darstellt, und der Wert -1 für keine Verbindung steht.
- **Kantenobjekte:** Speicherung beliebiger Informationen mit Kanten (z.B. Wegbeschreibung, Gewichtsfunktion, Zugriff auf die Knoten, ...). Wert null bedeutet keine Kante.

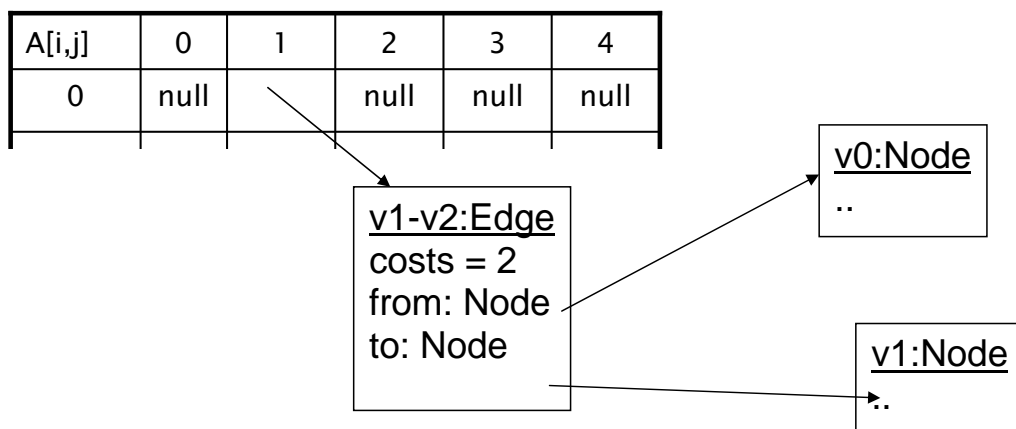


Abbildung 28: Adjazenzmatrix mit Kantenobjekten

Datenstruktur Adjazenzmatrix mit Kantenobjekten

- Knotenobjekte speichern Werte der Knoten
- Kantenobjekte speichern Verweise auf Knotenobjekte und weiteren Informationen bei den Kanten
- AdjazenzMatrix enthält

- Array der Knotenobjekte
- Matrix mit Verweisen auf die Kantenobjekte

```

class Node {
    Object value;
    ...
}

class Edge {
    Node from;
    Node to;
    ...
}

class AdjMatrix {
    int maxNodes;
    Node[] nodes = new Node[maxNodes];
    Edge[][] adjMatrix = new Edge[maxNodes, maxNodes]
    ...
}

```

Darstellungsform: Adjazenzliste

Bei der Adjazenzliste speichert man mit den Knoten eine linearen Liste der adjazenten Knoten.

Beispiel:

$$V = \{1,2,3,4,5\}$$

$$E = \{<1,3>, <2,3>, <2,5>, <3,1>, <5,4>\}$$

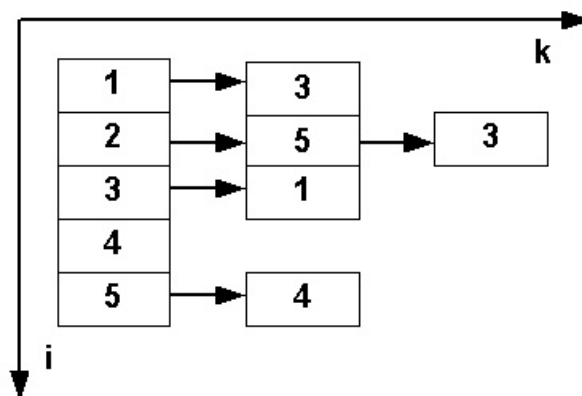


Abbildung 29: Adjazenzliste

! Bei ungerichteter Graph muss zu jeder "gerichteten" Kante $\langle v,w \rangle$ auch die entgegengesetzte Kante $\langle w,v \rangle$ eingetragen werden.

Datenstruktur Adjazenzliste

Folgend sieht man eine Implementierung für eine Adjazenzliste.

- Die Knotenobjekte Node speichern dabei einen Wert (in dem Fall ein int) und Verweise auf einen nächsten Knoten (wie bei einer linearen Liste).
- Die Adjazenzliste hält ein Array mit allen Knoten, wobei bei diesen Knoten die Verkettung jeweils der adjazenten Knoten erfolgt.

```
class Node = {
    int number;
    Node next;
}
class AdjList {
    int maxNodes;
    Node[] adjList = new Node[maxNodes];
    ...
}
```

Speicherbedarf

Speicherbedarf für die Adjazenzliste ist

$$O(|V| + |E|)$$

d.h. Speicherbedarf ist proportional zur Anzahl der Knoten + Anzahl der Kanten. Damit ist die Adjazenzliste gut geeignet für lichte Graphen.

Darstellungsform: Kantenliste

Bei Datenstruktur Kantenliste geht man folgend vor:

- Kanten sind Objekte mit Verweisen auf inzidente Knoten
- Knoten und Kanten werden in Liste verwaltet

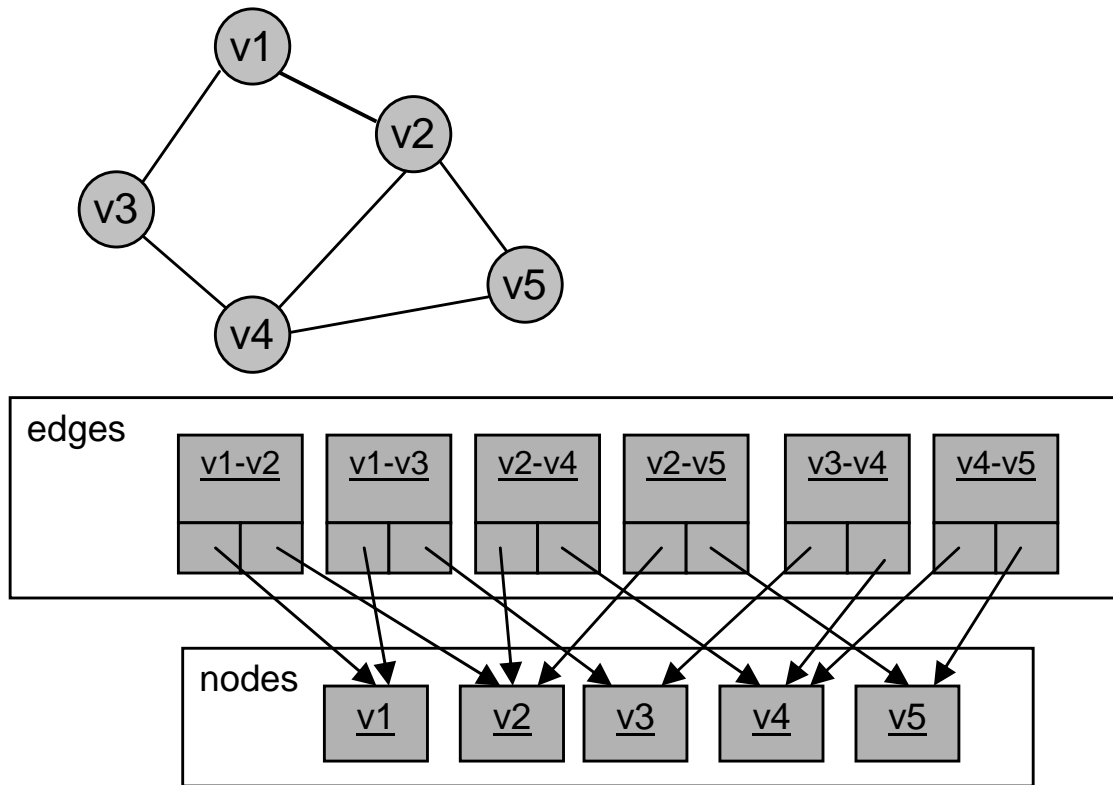


Abbildung 30: Kantenliste

Datenstrukturen für Kantenliste:

- Knoten enthalten Werte
- Kanten haben Verweise auf 2 Knoten from und to
- Der Graph führt eine Liste aller Knoten und eine Liste aller Kanten mit.

```
// Knotenklasse
class Node {
    Object value;
    ...
}

// Kantenklasse mit Verweise auf inzidente Knotenobjekte
class Edge {
    Node from;
    Node to;
    ...
}

// Graph mit Liste der Knoten und Kanten
class Graph {
    int maxNodes;
    Node[] nodes = new Node [maxNodes];
    Edge[] edges = new Edges[(maxNodes * (maxNodes - 1)) / 2];
    ...
}
```

Darstellungsform: Voll verkettete Knoten und Kanten

Die Speicherung von Graphen als verkettete Knoten- und Kantenobjekte erfolgt folgend:

- Knoten haben eine Liste der inzidenten Kanten
- Kanten haben Verweise auf die zwei adjazenten Knoten

Vorteil dieser Speicherung ist, dass man den Graphen direkt objektorientiert durchwandern kann, d.h. von Knoten über Kanten zu weiteren Knoten und wieder zurück.

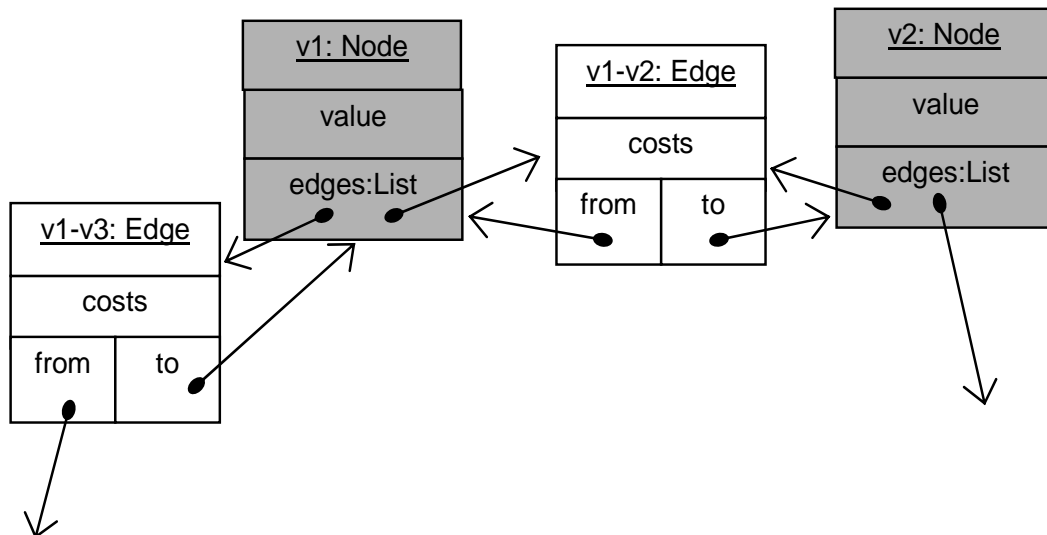


Abbildung 31: verkettete Knoten und Kanten

Datenstrukturen für voll verkettete Knoten und Kanten

- Knoten enthalten Werte plus eine Liste der inzidenten Kanten
- Kanten haben Verweise auf 2 Knoten from und to
- Der Graph führt eine Liste aller Knoten und eine Liste aller Kanten mit.

```
// Knotenklasse mit Liste der Kanten
class Node {
    Object value;
    List edges;
    ...
}

// Kantenklasse mit Verweise auf inzidenten Knoten
class Edge {
    Node from;
    Node to;
    int weight;
    ...
}

// Graph mit Liste der Knoten und Kanten
type Graph = {
    int maxNodes;
    Node[] nodes = new Node [maxNodes];
    Edge[] edges = new Edges [(maxNodes * (maxNodes -1))/2];
    ...
}
```

ADT Graph, Node and Edge (ungerichtet)

ADT Node:

Object value() - liefert den Wert des Knotens

Node[] adjacentNodes() - liefert die adjazenten Knoten des Knotens

Edge[] incidentEdges() - liefert die inzidenten Kanten des Knotens

Node next(Edge e) - liefert den durch Kante e adjazenten Knoten

Edge edgeToNode(Node next) – liefert die Kante, die zu Knoten next führt

int degree() - liefert den Grad des Knotens

boolean isAdjacent(Node v) - true, falls dieser Knoten adjazent zu v ist

ADT Edge :

double weight() - Gewicht (Kosten) der Kante

Node[] nodes() - liefert die Knoten der Kante (Array der Länge 2)

Node first() - liefert den ersten Knoten der Kante

Node second() - liefert den zweiten Knoten der Kante

boolean isIncident(Node n) – true, falls die Kante inzident zu Knoten n ist

ADT Graph

int nrNodes() - Anzahl der Knoten des Graphen

int nrEdges() - Anzahl der Kanten des Graphen

Node[] nodes() - liefert die Knoten des Graphen

Edge[] edges() - liefert die Kanten des Graphen

boolean insertNode(Object o) - fügt einen neuen Knoten mit Wert o ein

boolean insertEdge(Node from, Node to, double weight) - fügt eine neue Kante von from nach to mit Gewicht weight ein

Edge findEdge(Node from, Node to) – findet die Kante, die von from nach to führt

boolean deleteNode(Node n) - löscht den Knoten n

boolean deleteEdge(Edge e) - löscht die Kante e

boolean deleteEdge(Node from, Node to) - löscht die Kante von from nach to

Interface-Definitionen für Graphen

Interface Node

```
public interface Node {
    /** liefert den Wert des Knotens */
    Object value();

    /** liefert die adjazenten Knoten des Knotens*/
    Node[] adjacentNodes();

    /** liefert die inzidenten Kanten des Knotens */
    Edge[] incidentEdges();

    /** liefert den durch Kante e adjazenten Knoten*/
    Node nextNode(Edge e);

    /** liefert die Kante die zu Knoten next führt */
    Edge edgeToNode(Node next);

    /** liefert den Grad des Knotens */
    int degree();

    /** prüft, ob dieser Knoten adjazent zu n ist*/
    boolean isAdjacent(Node n);

    /** liefert den Graph zu dem Knoten gehört*/
    Graph graph();
}
```

Interface Edge

```
public interface Edge {

    /** liefert das Gewicht der Kante */
    double weight();

    /** liefert die Knoten der Kante (Array der Länge 2) */
    Node[] nodes();

    /** liefert den ersten Knoten der Kante */
    Node first();

    /** liefert den zweiten Knoten der Kante */
    Node second();

    /** prüft, ob diese Kante e inzident zum Knoten n ist*/
    boolean isIncident(Node n);

    /** liefert den Graph zu dem Knoten gehört*/
    Graph graph();
}
```

Interface Graph

```
public interface Graph {

    /** liefert die Anzahl der Knoten */
    int numNodes();

    /** liefert die Anzahl der Kanten */
    int numEdges();

    /** liefert die Knotenobjekte in einem Array */
    Node[] nodes();

    /** liefert die Kantenobjekte in einem Array */
    Edge[] edges();

    /** fügt einen Knoten mit neuem Wert o ein */
    boolean insertNode(Object o, Point pos);

    /** fügt einen Kante von from zu to und Gewicht weight ein */
    boolean insertEdge(Node from, Node to, double weight);

    /** liefert die Kante von Knoten from to Knoten to */
    Edge findEdge(Node from, Node to);

    /** löscht einen Knoten n */
    boolean deleteNode(Node n);

    /** löscht eine Kante e */
    boolean deleteEdge(Edge e);

    /** löscht die Kante von Knoten from zu Knoten to */
    boolean deleteEdge(Node from, Node to);

}
```

7.5 Durchwandern von Graphen (Traversierung)

Algorithmen auf Graphen durchwandern den Graphen, indem sie von Knoten über Kanten zu nächsten Knoten gehen und entsprechende Operationen durchführen. Durchwandern von Graphen arbeitet ganz ähnlich dem Durchwandern von Bäumen (vergleiche Abschnitt 4.4). Das Durchwandern erfolgt durch

Expandieren von Knoten

womit das Generieren der adjazenten Knoten eines Knotens verstanden wird. Das heißt, Durchwandern erfolgt, indem ausgehend von einem Startknoten dieser expandiert wird und damit alle adjazenten Knoten behandelt werden.

Im Gegensatz zu Bäumen muss bei Graphen erkannt werden,

ob ein Knoten schon besucht wurde.

Würde man das nicht prüfen, würde man Zyklen gehen und der Algorithmus würde nicht terminieren. Man benötigt daher eine Möglichkeit, bei Knoten zu vermerken, dass man diesen Knoten bereits besucht hat. Trifft man daher auf einen Knoten

- prüft man zuerst, ob dieser bereits besucht wurde
- nur wenn er noch nicht besucht wurde, behandelt man den Knoten und kennzeichnet ihn danach als besucht.

Das heißt, für das Durchwandern von Graphen benötigt man eine Operation

```
boolean visited(Node node)
```

die für jeden Knoten angibt, ob er bereits besucht wurde. Möglichkeiten diese Operation zu realisieren sind:

- bei jedem Knoten ein Flag `visited` einfügen, das initial `false` ist und dann nach der Behandlung des Knotens auf `true` gesetzt wird
- eine Liste `visitedNodes`, die am Anfang leer ist und in die jeder besuchte Knoten eingefügt wird.

Beim vollständigen Durchwandern von Graphen wird immer ein

Spannbaum des Graphen

erzeugt.

Wesentlich beim Durchwandern ist die Reihenfolge, in der die Knoten besucht werden. Wie bei Bäumen unterscheidet man unterschiedliche Arten, Graphen zu durchwandern. Die zwei grundlegenden Arten sind:

- Tiefentraversierung (*Depth-First-Traversal*), wobei es eine
 - rekursive und
 - eine nicht-rekursive Art gibt
- Breitentraversierung (*Breadth-First-Traversal*).

Weitere Arten, wie z.B. die so genannte Best-First-Traversal, ergeben sich als Erweiterung dieser Algorithmen.

Tiefentraversierung rekursiv (recursive Depth-First-Traversal)

Die Tiefentraversierung erfolgt analog der rekursiven Traversierung bei Bäumen, mit dem Zusatz, dass bereits besuchte Knoten nicht mehr behandelt werden.

Ausgehend von einem beliebigen Knoten wird in einer rekursiven Methode `visit`

- zuerst geprüft, ob der Knoten nicht bereits besucht wurde
- die Operation auf dem Knoten ausgeführt (*preorder* Version)
- der Knoten als besucht gekennzeichnet
- der Knoten expandiert, indem für alle adjazenten Knoten rekursiv die Methode `visit` aufgerufen wird.

Bei dieser Art werden immer sofort die Nachfolger des aktuellen Knotens besucht, bevor die Nachbarknoten gleicher Ebene betrachtet werden, was der Tiefentraversierung entspricht.

```
void visit(Node node) {
    node.doOperation();
    mark node as visited
    for each next  $\in$  node.adjacentNodes() {
        if (next not visited yet ) {
            visit (next);
        }
    }
}
```

Aufgabe:

Zeichnen Sie in folgender Abbildung den Ablauf der Tiefentraversierung ausgehend von Knoten 1 ein. Tragen Sie dazu jeweils die Knoten ein, für die rekursiven Aufrufe von `visit` erfolgt sind.

Überzeugen Sie sich, dass ein Spannbaum erzeugt wird. Beachten Sie weiters, dass der entferntere Knoten 4 und 5 vor näher liegenden Knoten behandelt werden.

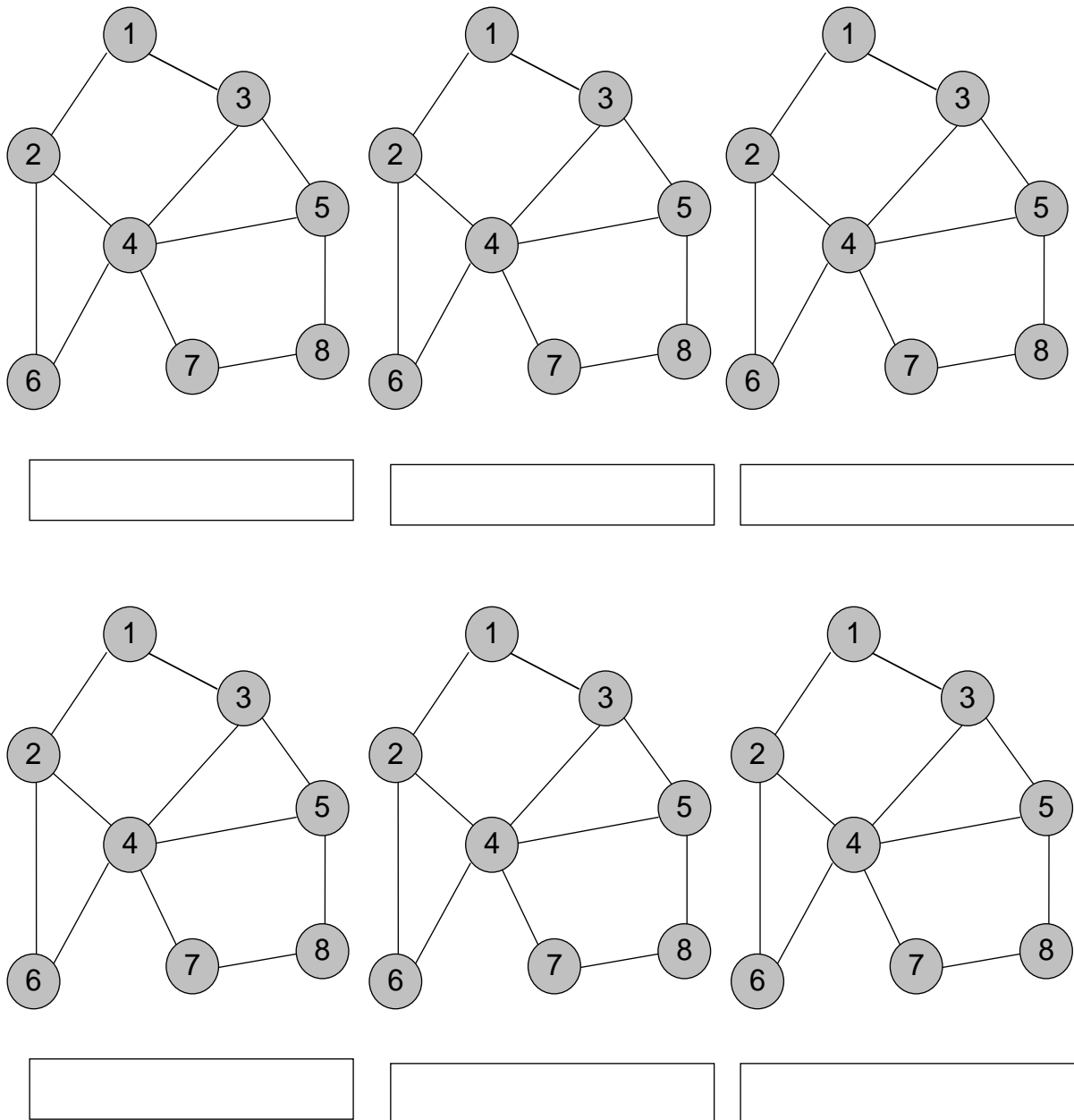


Abbildung 32: Arbeitsblatt Rekursive Tiefentraversierung

Tiefentraversierung nicht-rekursiv (non-recursive Depth-First-Traversal)

Die nicht-rekursive Variante der Tiefentraversierung von Graphen verwendet einen

Stack (!)

auf den die Nachfolgeknoten eines besuchten Knotens gelegt werden. In einer Schleife wird immer der oberste Knoten am Stack als nächster behandelt, indem

- die Operation auf dem Knoten ausgeführt wird (*preorder* Version)
- der Knoten als besucht gekennzeichnet wird
- der Knoten expandiert wird, indem alle adjazenten Knoten, die noch nicht besucht wurden und nicht schon am Stack liegen, auf den Stack gelegt werden.

```

visit(Node start) {
    stack.push(start)
    while (!stack.isEmpty()) { //solange Knoten am Stapel
        node = stack.pop()
        node.doOperation()
        mark node as visited
        // alle Nachbarknoten
        for each next ∈ node.adjacentNodes() {
            if ( next not visited yet && ! stack.contains(next)) {
                stack.push(next);
            }
        } //end for
    } //end while
}

```

Aufgabe:

Zeichnen Sie in folgender Abbildung den Ablauf der Tiefentraversierung ausgehend von Knoten 1 ein. Tragen Sie jeweils den aktuellen Inhalt des Stacks ein.

Überzeugen Sie sich, dass der gleiche Weg wie bei der rekursiven Lösung beschriftet wird.

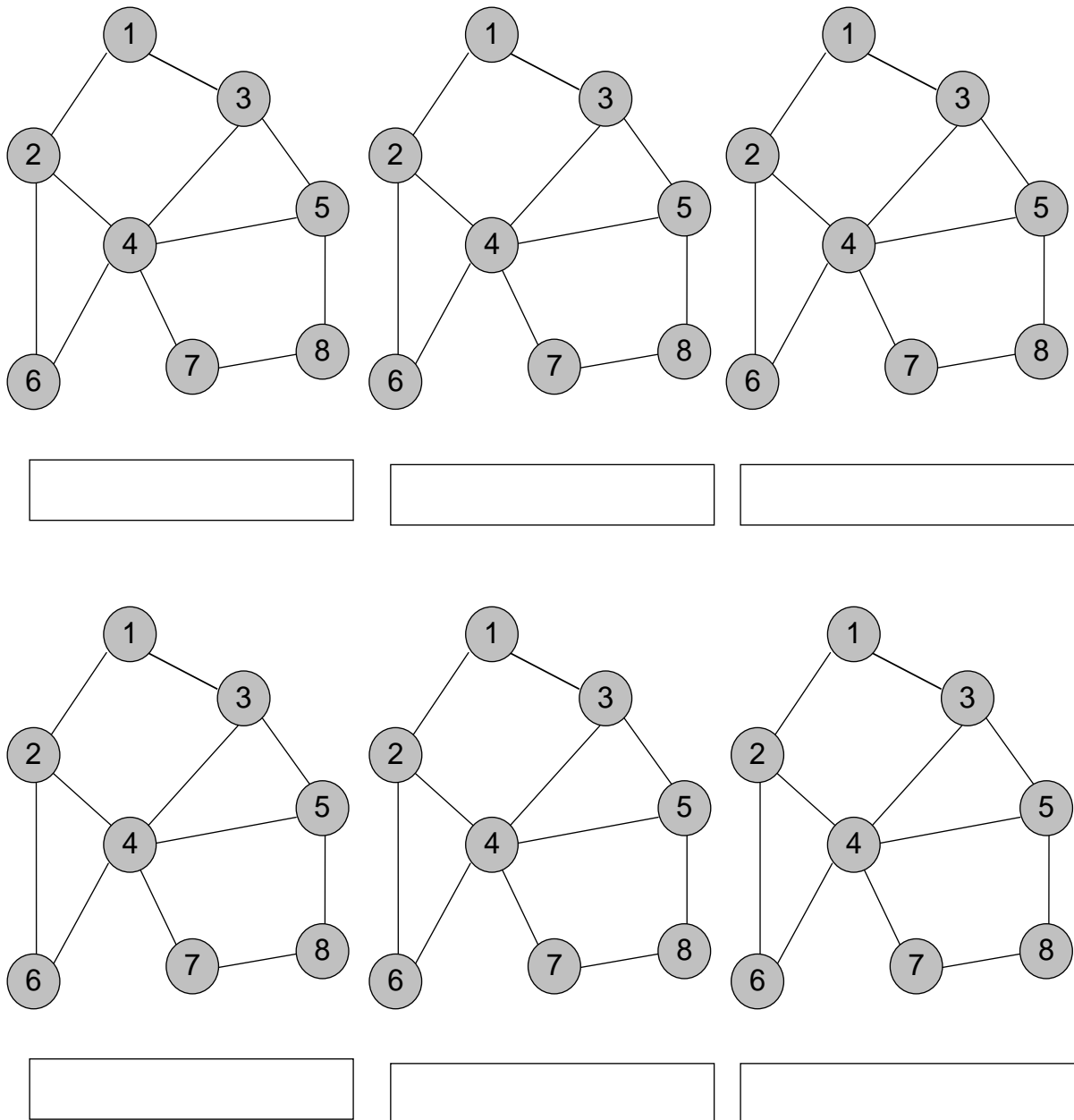


Abbildung 33: Arbeitsblatt nicht-rekursive Tiefentraversierung

Breitentraversierung (Breadth-First-Traversal)

Bei der Breitentraversierung werden immer sofort **alle** Nachfolger eines Knotens besucht, bevor weiter in die Tiefe gegangen wird. Das heißt ausgehend von einem Startknoten werden

- alle direkten Nachfolger des Startknotens besucht
- dann werden die Knoten, die 2 Kanten vom Startknoten entfernt sind, besucht
- dann die mit Entfernung 3, Entfernung 4, usw.

Der Algorithmus für die Breitensuche ergibt sich direkt aus der nicht-rekursiven Tiefensuche, indem der Stack durch eine

Queue (!)

ersetzt wird. Das heißt,

- immer erster in Queue wird untersucht
- der aktuell besuchte Knoten wird expandiert, indem neue Nachfolgeknoten des Knotens hinten angefügt und daher nach allen bereits eingefügten Knoten gereiht werden.

```
visit(Node start) {
    queue.enqueue(start)
    while (! queue.isEmpty()) { //solange Knoten am Stapel
        queue.dequeue(start);
        node.doOperation()
        mark node as visited
        // alle Nachbarknoten
        for each next  $\in$  node.adjacentNodes() {
            if ( next not visited yet && ! queue.contains(next)) {
                queue.enqueue(next); // hinten anfügen
            }
        } //end for
    } //end while
}
```

Aufgabe:

Zeichnen Sie in folgender Abbildung den Ablauf der Breitensuche ausgehend von Knoten v1 ein. Tragen Sie jeweils den aktuellen Inhalt der Queue ein.

Überzeugen Sie sich dass alle Knoten mit Entfernung 1 vor Knoten mit Entfernung 2 vom Startknoten besucht werden. Beachten Sie weiters dass auch hier ein Spannbaum erzeugt wird. Vergleichen Sie dann die Tiefe des Spannbaums bei der Tiefensuche und der Breitensuche.

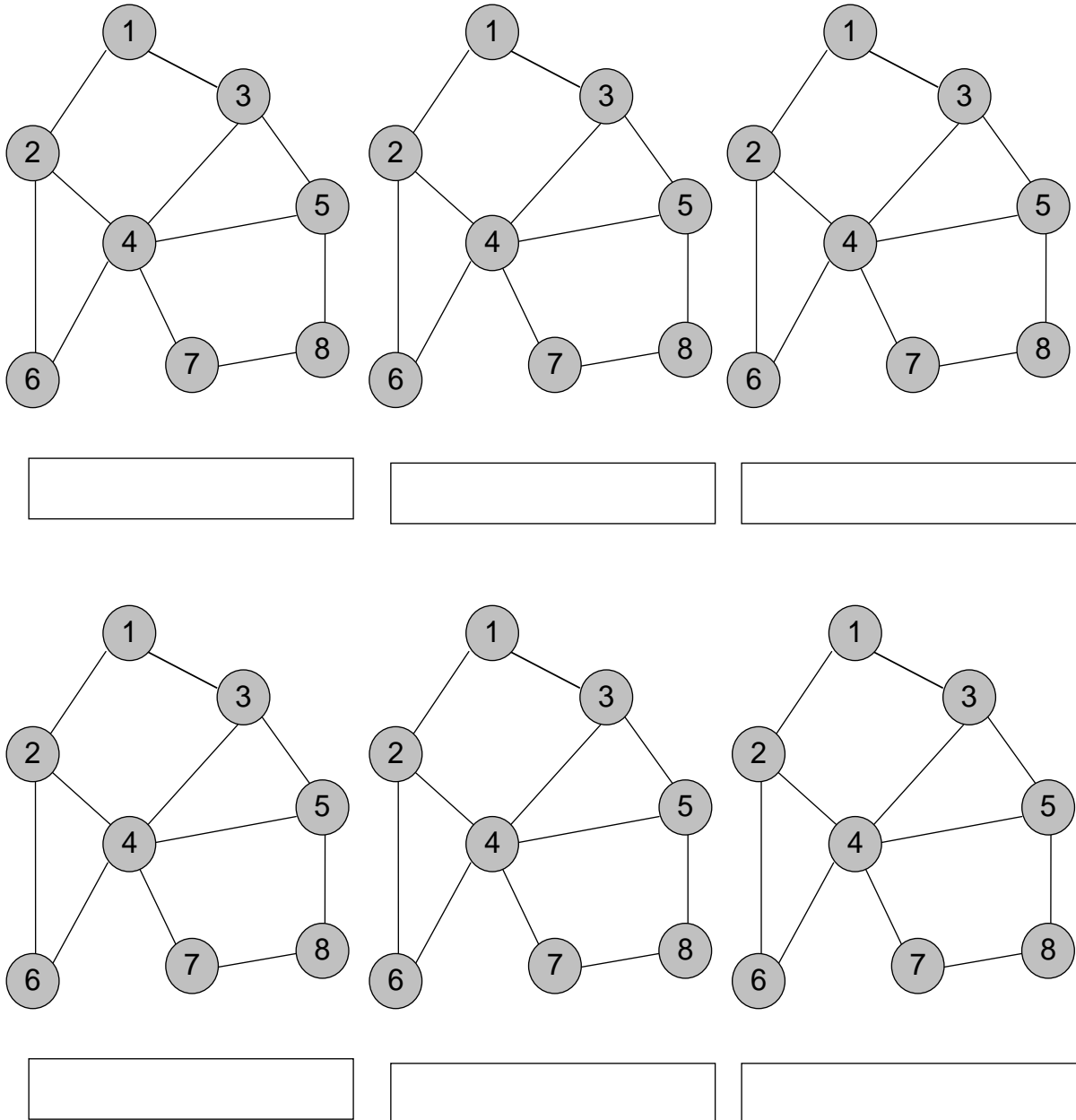


Abbildung 34: Arbeitsblatt Breitentraversierung

7.6 Zusammenfassung

- Graphen werden bei unterschiedlichen Problemen der Informatik als allgemeines Konzept eingesetzt
- Graphen sind ein mathematisches Konzept
- unterschiedliche Arten von Graphen
 - ungerichtete Graphen
 - gerichtete Graphen
 - gewichtete Graphen
 - Netzwerke

- unterschiedliche Datenstrukturen zur Speicherung
 - Adjazenzmatrix
 - Adjazenzliste
 - Kantenliste
 - verkettete Knoten- und Kantenobjekte
- Allgemeine Algorithmen
 - Tiefentraversierung
 - Breitentraversierung
- Spezielle Algorithmen, wie Suche nach kürzesten Wegen, bauen auf diesen grundlegenden Verfahren auf.

8 Algorithmen auf Graphen

Für Graphen gibt es eine Reihe von wichtigen Algorithmen, die breit einsetzbar sind:

- Minimaler Spannbaum
- Kürzeste Wege
- Flussoptimierung
- uw.

Im Folgenden wollen wir Algorithmen für minimalen Spannbaum, kürzeste Wege und Flussoptimierung besprechen.

8.1 Minimaler Spannbaum

Einführende Betrachtungen

Berechnung eines (nicht-minimalen) Spannbaums durch Traversierung

Erinnern wir uns, dass Spannäume, Teilgraphen von Graphen sind, die alle Knoten enthalten und Bäume sind. Sie enthalten minimale Anzahl von Kanten, wobei der Graph noch zusammenhängend ist.

Depth-First-Traversal und Breadth-First-Traversal berechnen Spannäume eines Graphen. Diese sind aber nicht minimal d.h., die Gewichte der Kanten sind nicht minimal.

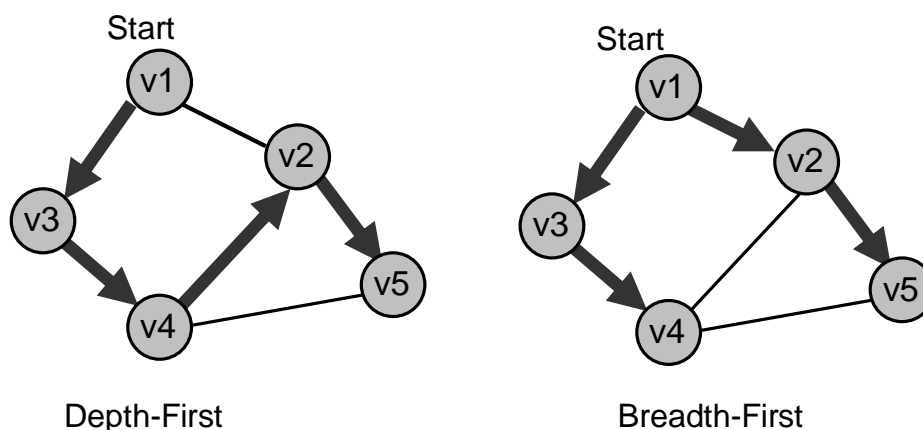


Abbildung 35: Berechnung von Spannäumen durch Depth-First-Traversal und Breadth-First-Traversal

Definition minimaler Spannbaum

Gegeben:

Ungerichteter zusammenhängender und gewichteter Graph $G = \{V, E\}$; V = Menge der Knoten, $E \subseteq V \times V$ (Menge der Kanten), $n = |V|$

Gesucht:

Minimaler Spannbaum (minimal spanning tree, shortest spanning tree, minimum genetic tree) $MST = (V, E_{MST})$ des ungerichteten Graphen $G = (V, E)$ ist Teilbaum von G mit folgenden Eigenschaften

- MST ist ein ungerichteter Baum (zusammenhängend, zyklensfrei)
- $E_{MST} \subseteq E$ (der minimale Spannbaum ist Teilgraph von G , $|E_{MST}| = n-1$)
- Die Summe der Kantenlängen von E_{MST} hat den kleinstmöglichen Wert aller Spannbäume von G

Eigenschaft eines minimaler Spannbaums

Die Algorithmen zur Berechnung von minimalen Spannbäumen beruht auf folgender Eigenschaft:

Für jede Zerlegung eines Graphen in zwei Subgraphen enthält der minimale Spannbaum die kürzeste der Kanten, die Knoten aus der einen Menge mit denen der anderen Menge verbinden.

Beispiel:

Betrachtet man die Zerlegung des Graphen in zwei disjunkte Subgraphen wie in Abbildung 36 gezeigt, so ist Kante (2, 3) mit Gewicht 1 Kante des minimalen Spannbaums.

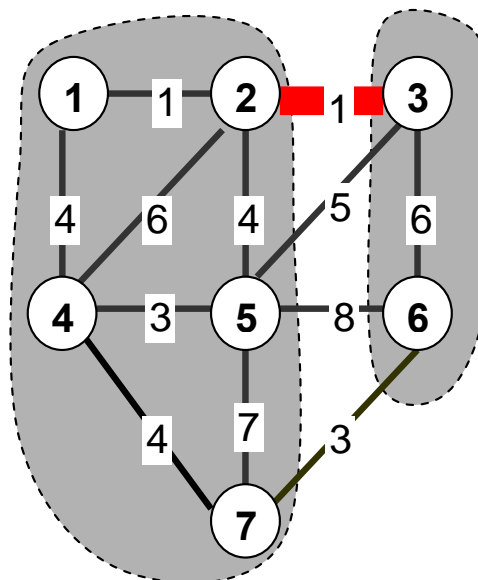


Abbildung 36: Eigenschaft eines minimalen Spannbaums

Algorithmus von Kruskal

Lösungsidee

Der minimale Spannbaum wird sequentiell aus Teilbäumen aufgebaut, indem Teilbäume über minimale Kanten vereinigt werden.

- gegeben gewichteter Graph $G = (V, E)$
- Bilde initial n Teilmengen von Knoten mit jeweils einem Knoten als Element (Partition der Knoten in n Blöcke $P = \{\{v_1\}, \{v_2\}, \dots, \{v_n\}\}$)
- solange nicht alle Teilmengen vereinigt sind
 - suche jene Kante $(v_1, v_2) \in E$ mit $v_1 \in V_1$ und $v_2 \in V_2$ und V_1 und V_2 sind unterschiedliche Teilmengen in P , welche minimales Gewicht hat
 - füge (v_1, v_2) zu E_{MST} hinzu
 - vereinige V_1 und V_2 zu einer Menge V_3 in P
- $MST = (V, E_{MST})$ ist ein minimaler Spannbaum von G

Algorithmus von Kruskal: Arbeitsblatt

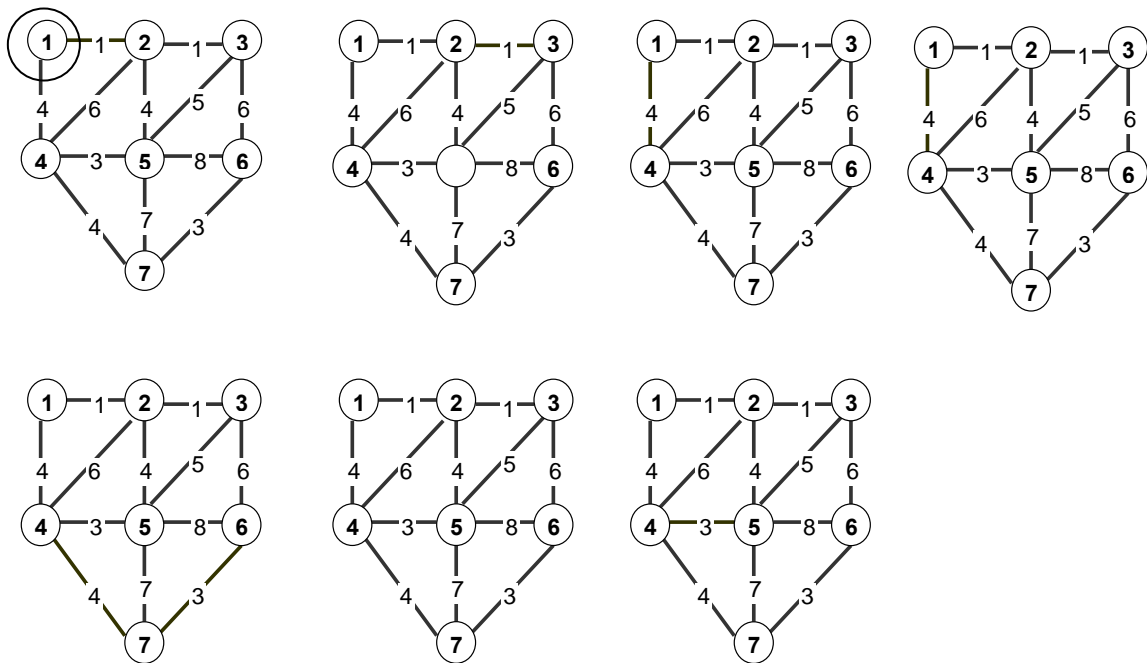


Abbildung 37: Arbeitsblatt Algorithmus von Kruskal

Wiederholung Prioritätswarteschlangen

In Kapitel 5 haben wir den ADT der Prioritätswarteschlangen besprochen. Diesen werden wir bei den folgenden Algorithmen einsetzen. Zur Wiederholung, die Idee von PriorityQueue ist die Reihung der Elemente nach einer Priorität (Ordnung, Sortierkriterium). Zu speichernde Elemente haben somit 2 Aspekte:

- Inhalt (Objekte)
- Ordnung (\leq)

PriorityQueue wurde als ADT, mit folgenden Operationen definiert:

- **insert(Object obj, Object key)** - Fügt ein Objekt nach dem Sortierkriterium **key** ein
- **Object findMin()** - Zugriff auf das kleinste Element
- **deleteMin()** - Löschen des kleinsten Elements
- **boolean isEmpty()** - Prüfen, ob Queue leer ist:

Zusätzlich verwenden wir hier folgende Operationen:

- **delete(Object obj)** - Löscht ein beliebiges Element aus der Prioritätswarteschlange
 - **boolean contains(Object obj)** - Prüft, ob ein Element enthalten ist
-

Algorithmus von Kruskal: Pseudocode

Der Algorithmus arbeitet mit folgenden Datenstrukturen:

- Datenstruktur **Parti ti on** als Menge von Mengen über Knoten $v \in V$; **Parti ti on** definiert Operationen
 - **uni fy**(Set s_1 , Set s_2) zum Vereinigen von zwei Teilmengen der Partition
 - **Set subset**(Node v) für das Finden der Teilmenge, die v enthält

Initial enthält die Partition n Teilmengen mit jeweils einen einzigen Knoten v_i :

$$p = \{ \{v_1\}, \{v_2\}, \dots, \{v_n\} \}$$

- Kanten werden in einer Prioritätswarteschlange verwaltet, wobei das Gewicht **wei ght** der Kante als Sortierkriterium verwendet wird.

```
Graph MST_Kruskal( Graph g=(V, E) )
    EMST = { }
    initialize Partition p = { {v1}, {v2}, ..., {vn} }, vi ∈ V
    for all e ∈ E do priorQueue.insert(e, weight(e));
    while (!priorQueue.isEmpty() ) {
        (v1, v2) = priorQueue.findMin();
        priorQueue.deleteMin();
        if (p.subset(v1) != p.subset(v2)) {
            // v1 and v2 belong to different subsets in p
            add edge (v1, v2) to EMST
            p.unify(p.subset(v1), p.subset(v2)) //unify subsets
        } //end if
    } //end while
    return MST = (V, EMST)
} //end MST_Kruskal
```

Greedy-Algorithmen

Der Prim-Jarnik-Algorithmus ist ein so genannter Greedy-Algorithmus. Greedy-Algorithmen sind spezielle Algorithmen mit folgenden Merkmalen:

- Ist eine Technik, die vorwiegend zur Lösung von Optimierungsproblemen (Maximum oder Minimum) verwendet wird.
- Ein Algorithmus nach dieser Strategie arbeitet sich sequentiell bezüglich einer gegebenen Zielfunktion an die optimale Lösung des vorgelegten Problems heran.
- Lösung ähnlich Depth-First-Traversal, aber es wird immer der aktuell "beste" Zweig verfolgt
- Realisierung mit Prioritätswarteschlangen

Algorithmus von Prim-Jarnik

Lösungsidee

Der Algorithmus von Prim-Jarnik zur Berechnung des minimalen Spannbaums arbeitet wie folgt:

- gegeben gewichteter Graph $G = (V, E)$
- gesucht minimaler Spannbaum $MST = (V, EMST)$
- beginne mit einem einzelnen Knoten $v_1 \in V$ und bilde eine Teilmenge $V' = \{v_1\}$ und eine Teilmenge $V'' = V \setminus \{v_1\}$
- solange $V'' \neq \{ \}$
 - suche jene Kante $(v_1, v_2) \in E$ mit $v_1 \in V'$ und $v_2 \in V''$, welche minimales Gewicht hat
 - füge (v_1, v_2) zu E_{MST} hinzu (durch die Eigenschaft für minimale Spannäume ist garantiert, dass (v_1, v_2) zum MST gehört)
 - nehme v_2 aus V'' heraus und füge v_2 in V' ein (v_2 ist bereits besucht)
- $MST = (V, EMST)$ ist ein minimaler Spannbaum von G

Algorithmus von Prim-Jarnik: Arbeitsblatt

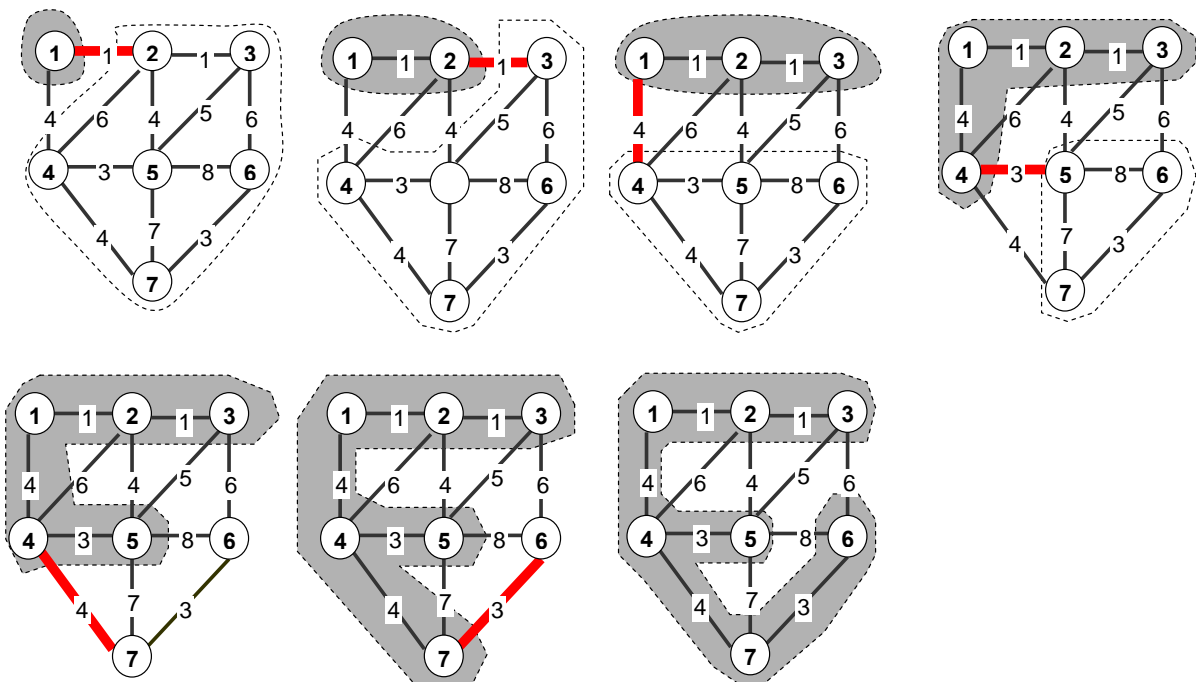


Abbildung 38: Arbeitsblatt Algorithmus von Prim-Jarnik

Algorithmus von Prim-Jarnik: Pseudocode

Der Algorithmus arbeitet mit folgenden Datenstrukturen:

- Zu noch nicht besuchten Knoten $v_2 \in V''$ wird in der Zuordnung $E[v_2]$ jene Kante $e = (v_1, v_2)$ gespeichert, die unter allen Kanten zu besuchten Knoten $v_1 \in V'$ minimales Gewicht hat¹.
- noch nicht besuchte Knoten $v_2 \in V''$ werden in einer Prioritätswarteschlange (`priorityQueue`) verwaltet, wobei das Gewicht `weight` von $E[v_2]$ als Ordnungskriterium verwendet wird (Anmerkung: ist $E[v_2] == null$, so sei `weight(E[v_2]) == ∞`).
- Es wird immer der erste Knoten v_1 aus der Prioritätswarteschlange besucht; dieser hat minimale Kante $E[v_1]$ aller noch nicht besuchten Knoten; die bei v_1 gespeicherte Kante $E[v_1]$ gehört zum minimalen Spannbaum.
- Für den neuen besuchten Knoten v_1 werden für alle adjazenten Knoten v_2 , die noch nicht besucht und somit noch in der Prioritätswarteschlange sind, die gespeicherte Kante $E[v_2]$ durch die Kante (v_1, v_2) dann ersetzt, wenn diese Kante geringeres Gewicht hat.
- **EMST** speichert schließlich alle Kanten, die zum minimalen Spannbaum gehören

! ¹ Wir werden im Folgenden oftmals für Abbildungen oder Zuordnungen von Werten zu Objekten, die Indexschreibweise `[]` bekannt von den Arrays verwenden. Das heißt, in diesem Fall bezeichnet $E[v]$ den Wert der Funktion $E: V \rightarrow E$ für den Wert v .

```

Graph MST_PrimJarnik ( Graph g=(V, E) )
  EMST = { }
  initialize E[v] = null for each v in V;
  for each v in V priorQueue.insert(v, weight(E[v]));
  while (!priorQueue.isEmpty) {
    v1 = priorQueue.findMin();
    priorQueue.deleteMin();
    mark v1 as visited
    if (E[v1] != null) EMST = EMST  $\cup$  {E[v1]}
    for all e=(v1, v2)  $\in$  v1.incidentEdges() {
      if ( v2 not visited yet)
        if (weight(e) < weight(E[v2]) ) {
          E[v2] = e;
          // Achtung: bewirkt Umsortieren in priorQueue
        } //end if
    } //end if
  } //end for
} //end while
return MST = (V, EMST);
} //end MST

```

- ! Achtung: Durch das Setzen von $E[v_2]$ ändert sich die Reihenfolge in der Prioritätswarteschlange. Das dadurch notwendige ständige Umsortieren der Elemente ist durch eine geeignete Implementierung der Prioritätswarteschlange zu unterstützen.

8.2 Kürzester Weg (Single source shortest path problem)

Problembeschreibung kürzeste Wege

Problembeschreibung

- Gegeben ist ein gewichteter Graph $G=(V, E)$
- Gesucht sind die kürzesten Wege ausgehend von einem Startknoten s zu den anderen Knoten

Definition Distanz:

Die Distanz $d(x, y)$ von Knoten x zu Knoten y ist definiert

$$d(x,y) = \begin{cases} \min\{distance(p) \mid p \text{ ist Weg von } x \text{ nach } y\} & \text{falls ein Weg existiert} \\ \infty & \text{sonst} \end{cases}$$

mit $distance(p)$ sind die Summe der Gewichte der Kanten am Weg p .

Kürzester Weg bei gleichen Kantengewichten

Sind alle Kantengewichte gleich, lässt sich mit einfachen **Breadth-First-Traversal** ein Shortest-Path-Suchalgorithmus realisieren. Der Algorithmus verwendet folgende Datenstrukturen:

- einen Datentyp **Path** als eine Liste von Knoten, der einen Weg definiert, mit folgenden Methoden:
 - **Path(Node s)** Konstruktor, der einen neuen Pfad mit s als Startknoten erzeugt
 - **Path extend(Node n)** erweitert den Pfad um Knoten n und liefert den erweiterten Pfad als Ergebnis
 - **int distance()** liefert die Länge des Pfades
 - **Node getLast()** liefert den Endknoten im Pfad
 - **Path getRest()** liefert den Rest des Pfades nach dem Endknoten
- eine Zuordnung **path: V → Path** (geschrieben **path[v]**), welche jedem Knoten v einen Weg zuordnet, der den zur Zeit kürzesten gefundenen Weg vom Startknoten s zum Knoten v darstellt.

Folgender Algorithmus liefert für gleiche Kantengewichte den Pfad mit minimaler Distanz von einem Startknoten s zu einem Zielknoten z :

```

Path ShortestPath(Node s, Node z, Graph g) {
    path[s] = new Path(s); //initial Path contains start
    queue.enqueue(s);
    while ( ! queue.isEmpty() ) {
        node = queue.dequeue();
        mark node as visited;
        if node == z return path[node];
        // für alle Nachbarknoten
        for all next  $\in$  g.adjacentNodes(node) {
            if ( next not visited yet && next  $\notin$  queue) {
                path[next] = path[node].extend(next)
                queue.enqueue(next); // hinten anfügen
            } // end if
        } // end for
    } // end while
    return null;
} // end ShortestPath

```

Arbeitsblatt: Kürzester Weg bei gleichen Kantengewichten

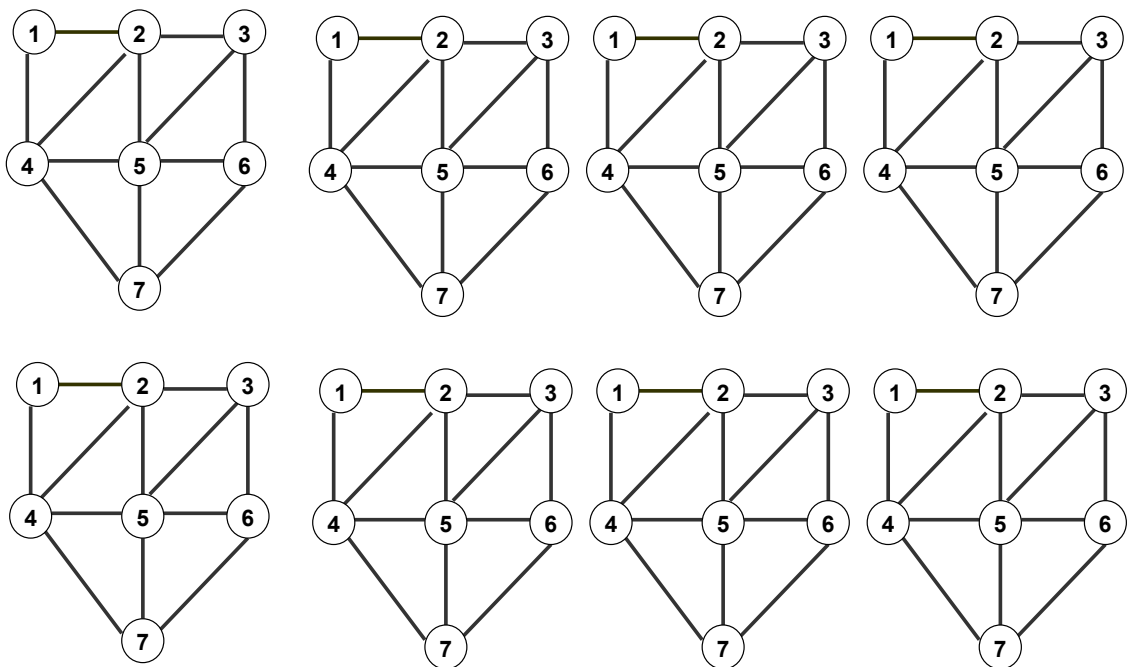


Abbildung 39: Arbeitsblatt Kürzester Weg bei gleichen Kantengewichten

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra ist der wichtigste Algorithmus für die Suche nach minimalen Wegen. Er berücksichtigt – im Gegensatz zum obigen Algorithmus – Kantengewichte.

Lösungsidee

Der Algorithmus funktioniert ähnlich dem Algorithmus von Prim-Jarnik für minimale Spannbäume, indem er die Elemente nach der aktuell besten Bewertung in einer Prioritätswarteschlange reiht und immer den aktuell besten weiter behandelt. Im Falle des Algorithmus von Dijkstra sind die Bewertungen die minimalen Distanzen dieser Knoten zum Startknoten.

Der Algorithmus verwendet folgende Zuordnung:

- Zuordnung $d: V \rightarrow \text{int}$ (geschrieben wieder $d[v]$) speichert für jeden Knoten v die bis zu diesem Zeitpunkt gefundene minimale Distanz des Knotens v zum Startknoten s ;
 $d[v]$ wird mit ∞ initialisiert für alle Knoten $v \neq s$, $d[s]$ wird auf 0 gesetzt.

Der Algorithmus arbeitet nach folgenden Prinzipien:

- Es wird immer der Knoten v als nächster besucht, der minimale Distanz $d[v]$ hat (Greedy-Algorithmus).
- Wird ein neuer Knoten v besucht, wird überprüft, ob über v ein Weg zu einem noch nicht besuchten adjazenten Knoten v_2 führt, der geringere Distanz hat als die aktuell gespeicherte minimale Distanz $d[v_2]$:

$$d[v] + \text{weight}(v, v_2) < d[v_2]$$

Ist dies der Fall wird der Wert von $d[v_2]$ mit $d[v] + \text{weight}(v, v_2)$ aktualisiert.

Algorithmus von Dijkstra: Pseudocode

Folgender Algorithmus liefert die minimale Distanzen $d[v]$ aller Knoten v zum gegebenen Startknoten s (Achtung: kein Pfad wird geliefert, siehe weiter unten!!)

Lösung unter Verwendung einer Prioritätswarteschlange `priorQueue` mit Knoten v als Elemente und $d[v]$ als Ordnungskriterium

```
d[] ShortestPath_Dijkstra( g=(V, E), s ∈ V )
  d[s] = 0
  for all v ∈ V if (v != s) d[v] = ∞
  priorQueue.insert(s, d[s]);
  while ( ! priorQueue.isEmpty() ) {
    v = priorQueue.findMin();
    priorQueue.deleteMin();
    mark v as visited;
    for all v2 ∈ v.adjacentNodes() {
      if ( v2 not visited yet )
        e = findEdge(v, v2)
        if (!priorQueue.contains(v2)) {
          d[v2] = d[v] + weight(v, v2);
          priorQueue.insert(v2, d[v2]);
        } else if (d[v] + weight(e) < d[v2]) {
          d[v2] = d[v] + weight(v, v2);
          //Achtung: bewirkt Umsortieren von v2 in priorQueue
        } //end if
      } //end if
    } //end for
  } //end while
  return d[v] for all v ∈ V
} //end ShortestPath_Dijkstra
```

Algorithmus von Dijkstra: Arbeitsblatt

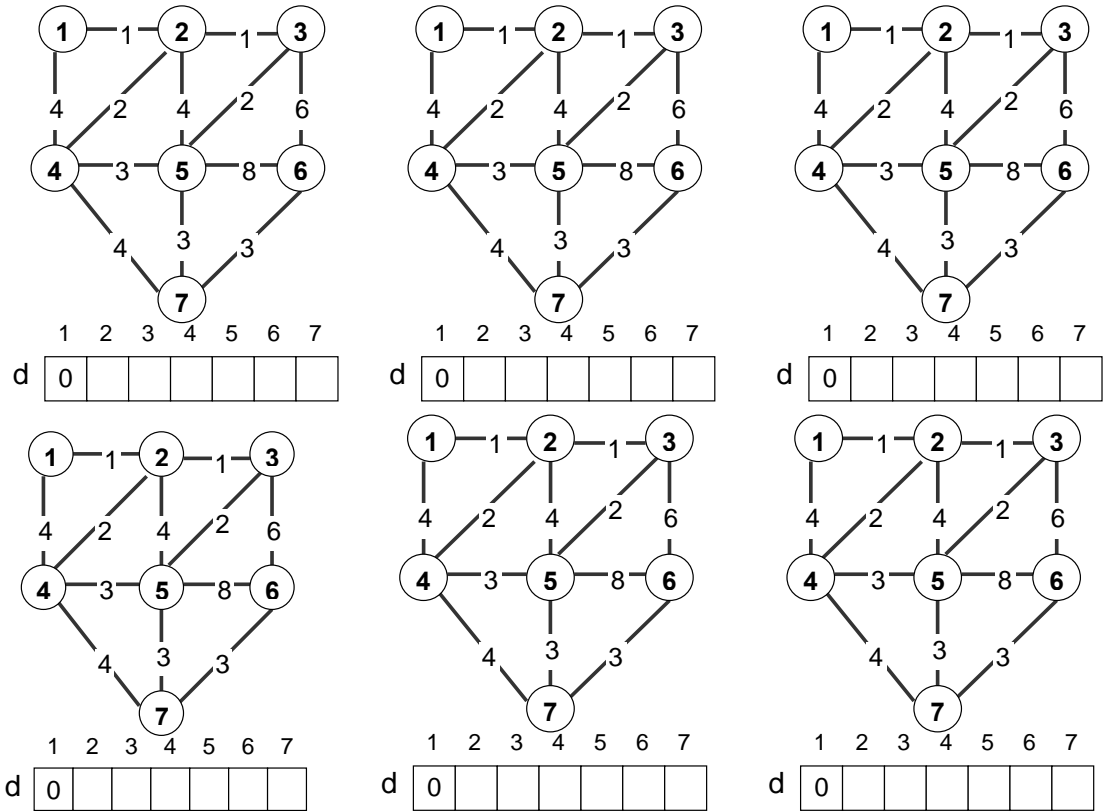


Abbildung 40: Arbeitsblatt Algorithmus von Dijkstra

Algorithmus von Dijkstra: Version kürzester Weg von Start zum Ziel

Obige Version des Dijkstra-Algorithmus berechnet die kürzesten Distanzen für alle Knoten. Im Folgenden wollen wir eine Version darstellen, die für einen Zielknoten den kürzesten Weg (= Folge von Kanten) bestimmt.

Dazu werden wieder folgende Datenstrukturen verwendet:

- Datenstruktur **Path** wie bei Algorithmus für kürzesten Weg bei gleichen Kantengewichten (wie in Abschnitt über kürzeste Wege bei gleichen Kantengewichten)
- **priorQueue**, wobei die Knoten **v** in **priorQueue** nach Distanz des Pfades des Knotens (**path[v].distance**) gereiht sind.

```
Path ShortestPathToDes (Node s, Node z, Graph g) {  
  
    path[s] = new Path(s);  
    priorQueue.insert(s, path[s].distance);  
  
    while (!priorQueue.isEmpty()) {  
        v = priorQueue.findMin();  
        priorQueue.deleteMin();  
        mark v as visited;  
        if (v == z) return path[v];  
        // alle Nachbarknoten  
        for all v2 ∈ v.adjacentNodes() {  
            if ( v2 not visited yet )  
                e = findEdge(v, v2)  
                if ((!priorQueue.contains(v2)) {  
                    path[v2] = path[v].extend(v2);  
                    priorQueue.insert(v2, path[v2].distance);  
                } else if (path[v].distance + weight(v, v2)  
                        < path[v2].distance) {  
                    path[v2] = path[v].extend(v2);  
                    //Achtung: bewirkt Umsortieren von v2 in priorQueue  
                } //end if  
            } //end if  
        } //end for  
    } //end while  
    return null; // not found  
} // end ShortestPathToDes
```

Best-First-Search nach kürzesten Wegen

Probleme beim Algorithmus von Dijkstra:

Der Dijkstra-Algorithmus nützt nur Wissen über "zurückgelegte Wege" und nicht Wissen über die "bevorstehenden Wege". Er kann daher keine „Richtung“ bestimmen, in die er bevorzugt gehen würde. Das bedeutet auch, dass der Algorithmus zuerst in die ganz falsche Richtung laufen kann (siehe Abbildung 41). Wie Abbildung 41 zeigt, ist dieses Vorgehen bei z.B. geometrischen oder geographischen Problemen nicht optimal.

Eine Lösung dazu bieten die *Best-First-Search-Algorithmen mit Heuristiken*, die wir im Folgenden besprechen wollen.

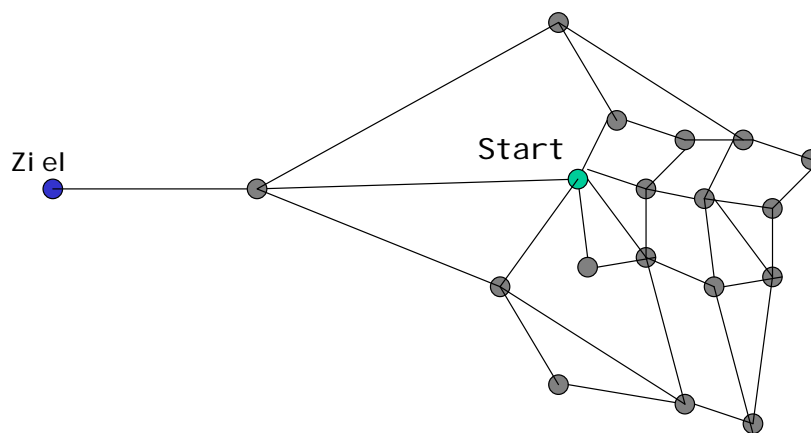


Abbildung 41: Beispiel für Problem beim Dijkstra-Algorithmus

Best-First-Search-Algorithmen mit Heuristiken

Idee ist, man verwendet bei der Suche eine Heuristik (= ungenaues Wissen), welche erlaubt, viel versprechende Richtungen der Suche zuerst einzuschlagen

- Man bewertet die Knoten auf Basis
 - Distanz des bisher zurückgelegten Weges (wie bei Dijkstra-Algorithmus)
 - plus Schätzung der Distanz des restlichen Weges zum Ziel
- Knoten werden nach der Bewertung gereiht und der Knoten mit minimaler Bewertung wird als nächster verfolgt

Beispiel Wege zwischen Orten: Schätzung erfolgt auf Basis

- Distanz des Pfades
- plus Luftliniendistanz des aktuellen Endknotens zum Zielknoten

Schätzwerte bei Best-First-Search-Algorithmen mit Heuristiken

Folgende wichtige Bedingungen werden an Schätzwerte gestellt:

- Schätzwerte müssen optimistisch sein, d.h. tatsächliche Distanz muss größer als Schätzwert sein !!!!! Erfüllen Schätzwerte diese Bedingung nicht, funktioniert der A*-Algorithmus nicht.
- Schätzwerte sollen naturgemäß möglichst gut sein
 - bei Schätzwert immer gleich 0 ergibt sich eine blinde Suche nach dem Dijkstra-Verfahren
 - bei Schätzwert gleich der exakten Distanz ergibt sich ein direktes Zugehen auf das Ziel ohne Irrwege
- Der Best-First-Search-Algorithmen arbeitet umso besser, je besser die Heuristik ist. Es gilt:
 - ist eine Schätzfunktion **h1** besser als eine Schätzfunktion **h2**, so betritt der Algorithmus mit **h1** keinen Knoten, der nicht auch vom Algorithmus mit **h2** betreten wird,
d.h. Ein Algorithmus mit einer besseren Schätzfunktion geht weniger Irrwege und findet immer schneller zum Ziel

A*-Algorithmus

A*-Algorithmus ist der klassische Best-First-Search-Algorithmus mit heuristischer Bewertung der Knoten. Er arbeitet ähnlich dem Dijkstra-Algorithmus aber mit anderer Sortierung der Knoten

- Knoten v geht in **priorQueue** nach Distanz des Pfads $\text{path}[v].\text{distance}$ plus heuristische Bewertung h

$$\text{path}[v].\text{distance} + h(v, z)$$

mit $h(v, z)$ Heuristik für Knoten v in Bezug auf z (z.B. Luftliniendistanz von Knoten v zu Ziel z)

Daraus ergibt sich folgende wesentliche Erweiterung:

- Beim Dijkstra-Algorithmus kann garantiert werden, dass für den nicht-besuchten Knoten v , welcher unter allen nicht-besuchten Knoten minimale Pfaddistanz $\text{path}[v].\text{distance}$ hat, der aktuelle Pfad der minimale Pfad ist.

Daraus folgt, dass wenn ein Knoten in **priorQueue** minimal ist und als nächster besucht wird, er später nicht mehr betrachtet werden muss.

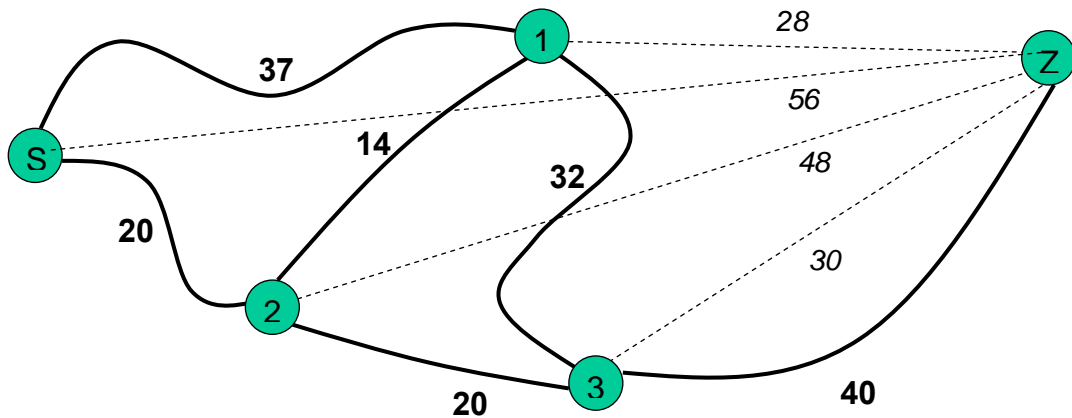
- Dadurch dass bei der Reihung auch der Schätzwert eingeht, ist diese Bedingung beim A*-Algorithmus nicht mehr gegeben und für bereits besuchte Knoten können neue minimale Pfade gefunden werden.

Solche Knoten müssen dann wiederum in der Suche berücksichtigt und erneut in **priorQueue** eingefügt werden.

```
Path ShortestPath_A*(Node s, Node z, Graph g) {
    path[s] = new Path(s);
    priorQueue.insert(s, path[s].distance + h(s, z));

    while (!priorQueue.isEmpty()) {
        v = priorQueue.findMin();
        priorQueue.deleteMin();
        if (v == z) return path[v];
        // alle Nachbarknoten
        for all v2 ∈ v.adjacentNodes() {
            e = findEdge(v, v2)
            if (path[v2] == null) {
                path[v2] = path[v].extend(v2);
                priorQueue.insert(v2, path[v2].distance + h(v2, z));
            } else if (path[v].distance + weight(v, v2)
                < path[v2].distance) {
                path[v2] = path[v].extend(v2);
                // neu einsortieren
                if (priorQueue.contains(v2)) {
                    priorQueue.delete(v2);
                }
                priorQueue.insert(v2, path[v2].distance + h(v2, z));
            } //end if
        } //end for
    } //end while
    return null; // not found
} // end ShortestPath_A*
```

A*-Algorithmus: Arbeitsblatt



		1. Schritt		2. Schritt		3. Schritt	
v							
priorQueue + Bewertung							
s	56						
1	28						
2	48						
3	30						
n	h	path	d	path	d	path	d

Abbildung 42: Best-First-Search Arbeitsblatt

v		s		1		2	
priorQueue + Bewertung		1/65 2/68		2 /68 3/99		1/62 3/70	
s	56	S	0	S	0	S	0
1	28	s-1	37	s-1	37	s-2-1	34
2	48	s-2	20	s-2	20	s-2	20
3	30	-	-	2-1-3	69	s-2-3	40
n	h	path	d	path	d	path	d

Abbildung 43: Best-First-Search Arbeitsblatt

8.3 Flussoptimierung

Gerichtete, gewichtete Graphen (Netzwerke) bilden die Basis für die Flussoptimierung. Dabei stellt der Graph mit den Knoten und gewichteten Kanten Möglichkeiten des Flusses (= gewichtete Kanten) zwischen Verbindungsstellen (= Knoten) dar. Die Flussoptimierung berechnet nun eine Verteilung des Flusses, sodass der Durchsatz insgesamt optimal ist.

Flussoptimierung kann z.B. verwendet werden für:

- Verteilung eines Verkehrstromes in einem Verkehrsnetz mit Verkehrsstrecken unterschiedlicher Kapazität
- Durchflussmaximierung über ein komplexes Pipelinesystem
- Stromverteilung in einem Hochspannungsverteilungssystem
- Transport von Rohstoffe auf vorgegebenen Transportwegen von Rohstofflieferanten zur Verarbeitungszentrale
- Transport von Abwasser durch ein vorgegebenes Entsorgungsnetz zur Abwasserreinigung

Problemstellung (eingeschränkt auf eine Quelle und eine Senke)

Die Flussoptimierung stellt sich auf der Ebene der Graphen als folgendes abstrakte Problem dar (Einschränkung auf Netzwerke mit einer Quelle und einer Senke).

- Gegeben:
 - Netzwerk (gerichteter, azyklischer Graph) mit einer Quelle (q) und einer Senke (s); Quelle hat keine eingehenden Kanten, Senke keine ausgehenden Kanten
 - Gewichte der Kanten entsprechen der maximalen Kapazität (max. Durchfluss, Transportvolumen...) der Kante
- Gesucht:
 - Belegung der Kanten e mit Fluss $F(e)$, sodaß optimaler Fluss durch das Netzwerk, wobei $0 \leq F(e) \leq \text{weight}(e)$ mit weight ist das Kantengewicht

Einfacher Optimierungsalgorithmus (1. Versuch)

Lösungsidee:

- Suche Pfad von der Quelle zur Senke und erhöhe den Durchfluss auf allen Kanten entlang dieses Pfads.
- Ausgehend von gegebenen Graph G erzeuge 2 Graphen
 - G_f (Flow): Kantengewichte entsprechen dem aktuellen "Durchfluss". Alle Durchflüsse in G_f werden zu Beginn auf 0 gesetzt.

- G_R (Residual): Kantengewichte entsprechen den verbleibenden Kapazitäten entlang den Kanten. Alle Gewichte in G_R werden zu Beginn auf die Gewichte von G gesetzt.
- Optimierungsschritte
 - Solange in G_R ein Pfad von q nach s existiert, wobei das kleinste Gewicht der Kanten im Pfad > 0 ist
 - bestimme kleinstes Gewicht g aller Kanten e im Pfad
 - für alle Kanten e im Pfad do
 - in G_F kleinstes Kantengewicht g bei Kante e addieren
 - in G_R kleinstes Kantengewicht g bei Kante e subtrahieren
- G_F enthält schließlich die gefundene Lösung

! Vorsicht ! Ergebnis ist von den gewählten Pfaden abhängig und ist möglicherweise nicht optimal.

Einfacher Optimierungsalgorithmus: Beispiel

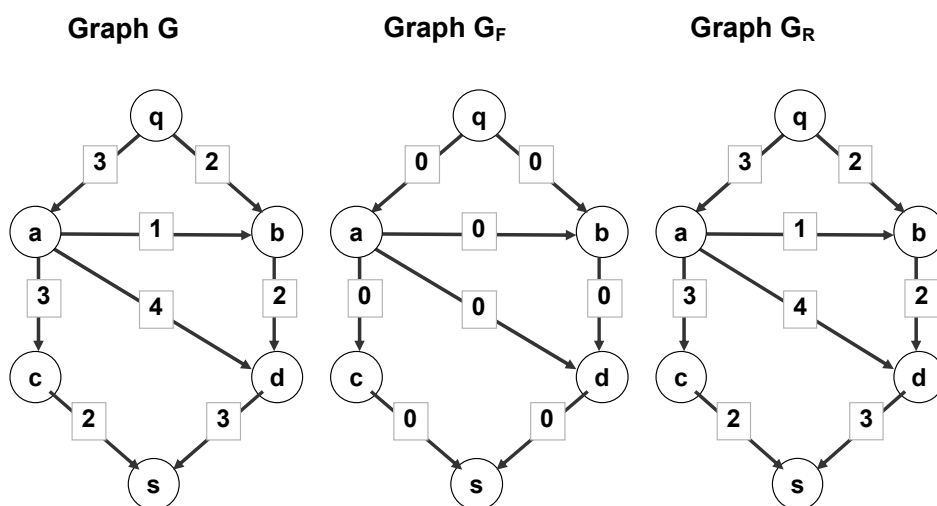


Abbildung 44: Beispiel einfacher Optimierungsalgorithmus

Einfacher Optimierungsalgorithmus: Arbeitsblatt 1

Es werden nacheinander die Pfade

- $q - a - c - s$ mit Flussmenge 2
- $q - b - d - s$ mit Flussmenge 2
- $q - a - d - s$ mit Flussmenge 1

gewählt und es ergibt sich ein optimaler Fluss.

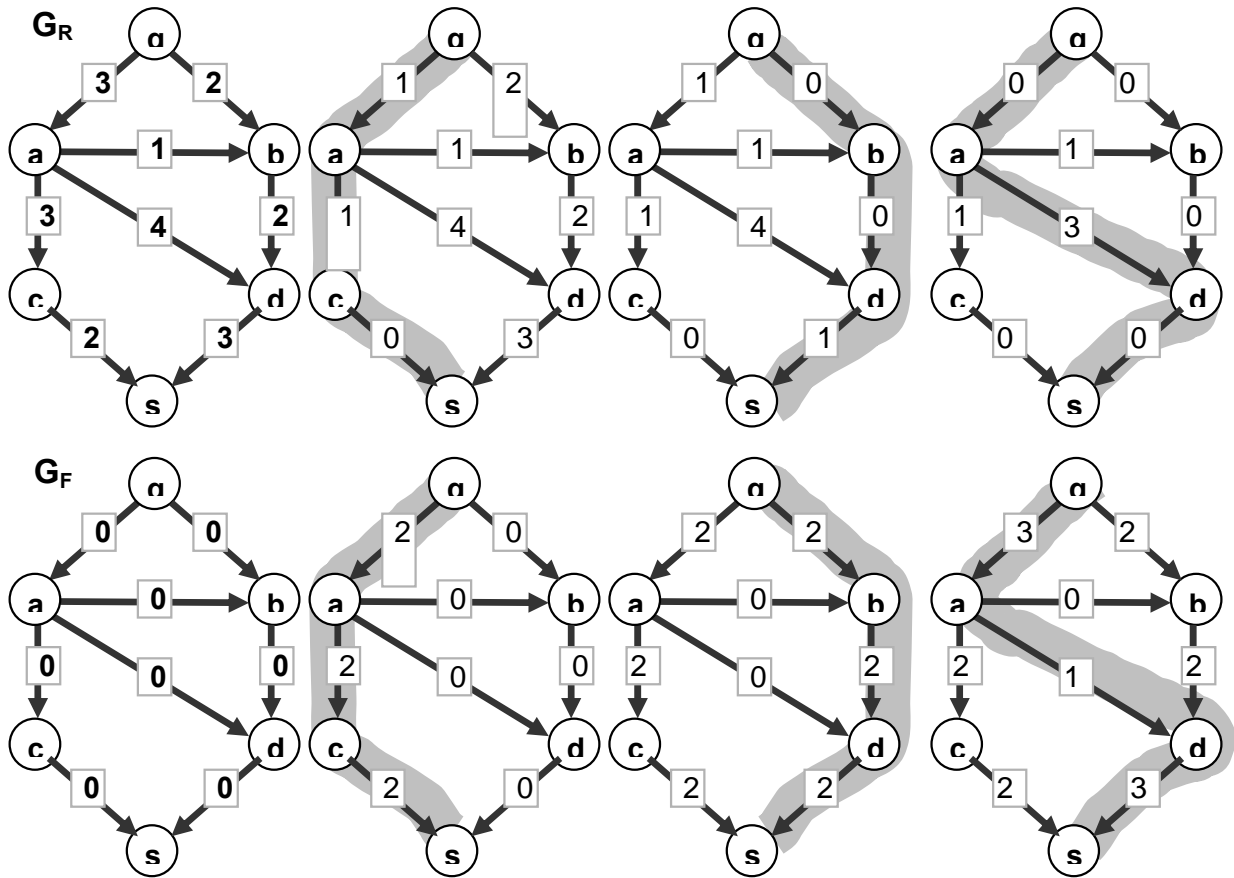


Abbildung 45: Einfacher Optimierungsalgorithmus: Arbeitsblatt 1

Einfacher Optimierungsalgorithmus: Arbeitsblatt 2

Wählt man aber zuerst den Pfad

- $q - a - d - s$ mit Fluss 3

kann man keine weiteren Pfade finden und man endet ohne das Optimum gefunden zu haben.

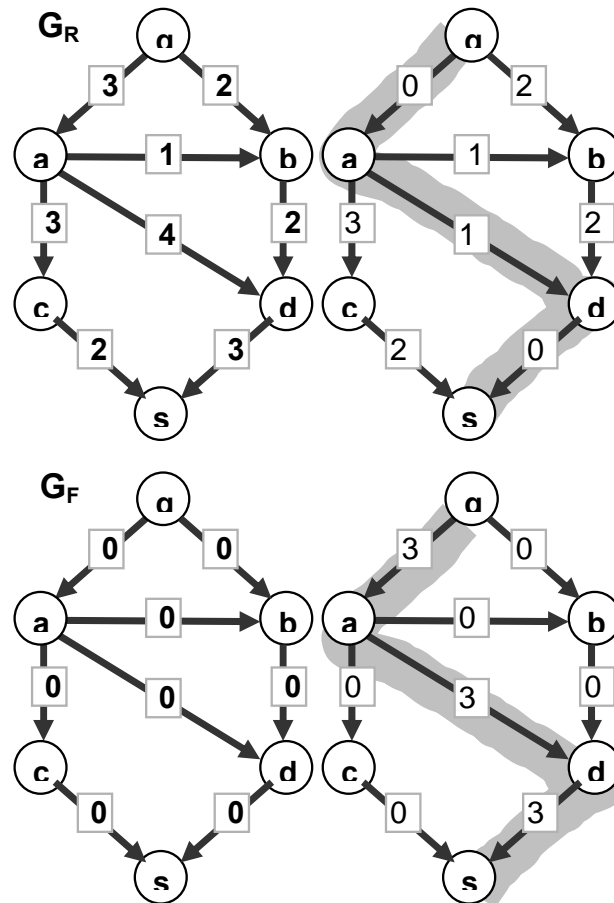


Abbildung 46: Einfacher Optimierungsalgorithmus: Arbeitsblatt 2

Optimierung nach Ford-Fulkerson

Der obige Algorithmus hat wie gezeigt die Schwäche, dass bei bestimmten Konstellationen durch unglückliche Auswahl von Pfaden, das Optimum nicht gefunden wird. Der folgende Algorithmus von Ford-Fulkerson löst dieses Problem.

Die Idee ist, dass einmal ausgewählte Flüsse, die zu nicht-optimalen Lösungen führen würden, auch wieder zurückgenommen werden können.

Vorgehen:

- Pfade können auch Rückwärtskanten beinhalten, welche Flüsse darstellen, die in G_F bereits definiert sind
- Flüsse in G_F werden in G_R als Rückwärtskanten betrachtet

= dies erlaubt die Zurücknahme von Flüssen

Beispiel:

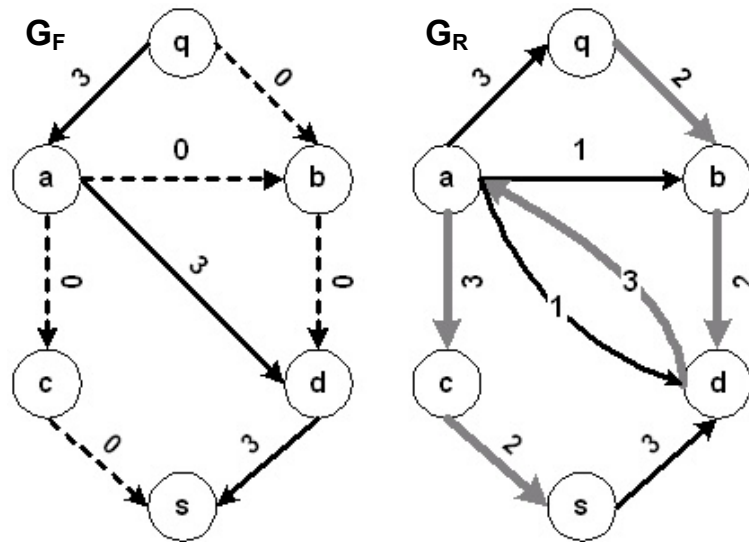


Abbildung 47: Beispiel Optimierung nach Ford-Fulkerson

Algorithmus nach Ford-Fulkerson

Initialisierung wie oben

die Kanten in G_F werden in G_R als Rückwärtskanten betrachtet!!

Optimierungsschritte:

- Solange in G_R ein Pfad von q nach s existiert, wobei das kleinste Gewicht der Kanten im Pfad > 0 ist
 - bestimme kleinstes Gewicht g aller Kanten im Pfad
 - für alle Kanten e im Pfad do
 - if Kante e ist Vorwärtskante
 - in G_F kleinstes Kantengewicht g bei Kante e addieren
 - in G_R kleinstes Kantengewicht g bei Kante e subtrahieren
 - if Kante e ist Rückwärtskante
 - in G_F kleinstes Kantengewicht g bei Kante e subtrahieren
 - in G_R kleinstes Kantengewicht g bei Kante e addieren

Algorithmus nach Ford-Fulkerson: Arbeitsblatt

Bei dem Beispiel aus folgender Abbildung wird wie folgt vorgegangen:

- Auswahl des Pfads q, a, d, s von der Quelle zur Senke und erhöhe Durchfluss um 3 Einheiten entlang des Pfads q, a, d, s .
- Nächster betrachteter Pfad: q, b, d, a, c, s . Durchfluss um 2 Einheiten entlang des Pfads q, b, d, a, c, s erhöht.

Beachten sie dass der Fluss von Knoten a zu Knoten d in diesem Schritt um 2 Einheiten zurückgenommen wurde.

- Kein weiterer Pfad von der Quelle zur Senke vorhanden

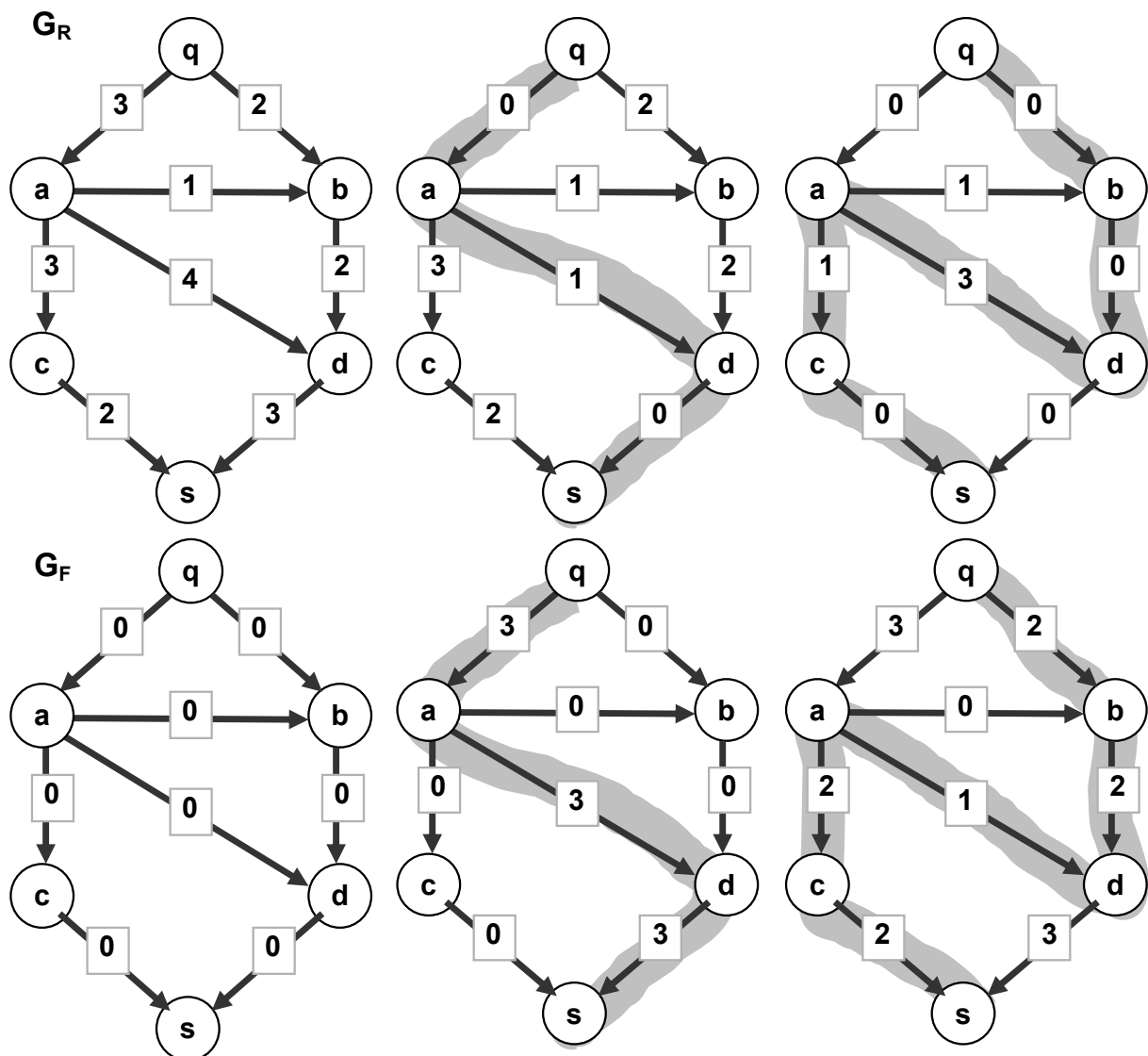


Abbildung 48: Algorithmus nach Ford-Fulkerson: Arbeitsblatt

8.4 Algorithmen auf Graphen: Zusammenfassung

- Algorithmen auf Graphen, die wichtige allgemeine Probleme lösen
 - Zusammenhang
 - kürzeste Wege
 - Flussoptimierung,
 - ...
- Algorithmus für die Suche nach kürzesten Wegen
 - Dijkstra-Algorithmus ohne Heuristiken als Basisalgorithmus für die Wegsuche
 - A*-Algorithmus kann auch Schätzungen für den restlichen Wege berücksichtigen und findet in dieser Weise schneller optimale Wege

9 Geometrische Algorithmen

- siehe Folienskriptum

10 Backtracking und NP-Vollständigkeit

- siehe Folienskriptum