Software-Entwicklung 1

V13: Stack/Queue, Sortieren und Graphen

Prof. Maalej & Team **@maalejw**

Status der 12. Übungswoche

Zeit	Montag	Dienstag	Mittwoch	Donnerstag	Freitag
Vormittag	Gruppe 1 **Erfüllt: 21%**	Gruppe 3 **Erfüllt: 31%**	Gruppe 5 **Erfüllt: %**	Gruppe 6 **Erfüllt: 7%**	Gruppe 8 **Erfüllt: 46%**
Nachmittag	Gruppe 2 **Erfüllt: 35%**	Gruppe 4 **Erfüllt: 26%**	Vorlesung	Gruppe 7 **Erfüllt: 36%**	

Übungsbetrieb

- In der 13. Übungswoche gibt es **keine verlängerte Abnahmefrist**!

- Alle 13 Übungszettel müssen abgenommen werden (Außnahme jeweils letzte Aufgabe eines Zettels)

- Siehe Scheinkriterien in Moodle

Überblick

1 **Stacks und Queues**

2 Sortieren

3 Graphen

Der Stack (Stapel/Keller)

- **Datenstruktur** die eine Folge von Elementen vom gleichen Datentypen enthält
- Eingeschränkte Einfüge- und Ausfügeoperationen
- Einfügeoperation **push**
  - **Fügt ein** Element **an das Ende** der Folge ein
- Ausfügeoperation **pop**
  - **Entfernt** das **letzte** Element der Folge und liefert es als Ergebnis
- Beim **LIFO-Prinzip** muss immer das zuletzt eingefügte Element als erstes wieder entfernt werden (**Last In First Out**)

Ein Stack-Interface

```
interface Stack<E>
{
  void push(E element);
  E pop();
  boolean isEmpty();
}
```

gegeben:
Stack<String> s
mit zwei Elementen.

```
s.push("X");
```

```
s.push("Z");
```

```
String t = s.pop();
// t == "Z"
```

Klasse Stack im Java Collections Framework

Constructor Summary

Constructors

Constructor and Description
`Stack()` Creates an empty Stack.

Method Summary

Methods

Modifier and Type	Method and Description
`boolean`	`empty()` Tests if this stack is empty.
`E`	`peek()` Looks at the object at the top of this stack without removing it from the stack.
`E`	`pop()` Removes the object at the top of this stack and returns that object as the value of this function.
`E`	`push(E item)` Pushes an item onto the top of this stack.
`int`	`search(Object o)` Returns the 1-based position where an object is on this stack.

Stack implementiert als Liste

```
class StackImpl<E> implements Stack<E>
{
   private List<E> implList = new LinkedList<E>();
   public void push(E el)
  {
     implList.add(0,el); // vorn einfügen
  }

   /** pop is valid only if not isEmpty */
   public E pop()
  {
     if (!isEmpty())
     {
        return implList.remove(0);  // vorn entfernen
     }
     else
            // Fehlerbehandlung
  }
   public boolean isEmpty()
  {
     return implList.isEmpty();
  }
}
```

Die Queue (Schlange)

- Eine Folge von Elementen eines gegebenen Datentyps
- Ein Element wird **an das Ende** der Folge **eingefügt**
- Das **erste Element** der Folge wird **entfernt** und als Ergebnis geliefert
- Beim **FIFO-Prinzip** wird das zuerst eingefügte Element einer Schlange als erstes wieder entfernt (engl. **First In First Out**)

Ein Queue-Interface

```
interface Queue<E>
{
  void enqueue(E element);
  E dequeue();
  boolean isEmpty();
}
```

gegeben:
Queue<String> q
mit zwei Elementen

```
q.enqueue("X");
```

```
q.enqueue("Z");
```

```
String s = q.dequeue();
// s == "A"
```

Breitendurchlauf mithilfe einer Queue

- Gegeben: Ein binärer Baum
- Gewünscht: Ein Breitendurchlauf durch die Knoten

- Lösungsskizze
  - Initialisierung: füge das Wurzelelement in eine Queue ein
  - Schleife – solange die Queue noch Elemente enthält:
    - Entnimm das erste Element (verarbeitete es) und füge seine Nachfolger (falls vorhanden) in die Queue ein

Implementationsskizze für eine Queue als zyklisches Array

```
Intern:
String[] _implArray =
          new String[6];
```

```
q.enqueue("X");
```

```
q.enqueue("Z");
```

```
String s = q.dequeue();
```

Dieser Entwurf als sogen. **Ringpuffer** ist speichereffizient für begrenzte Schlangen

Überblick

1 Stacks und Queues

2 **Sortieren**

3 Graphen

Sortieren

- **Klassische Informatikaufgabe** für Sammlungen
- Voraussetzung zum Sortieren ist eine **Ordnungsrelation**
- Die Algorithmen, nach denen Sammlungen sortiert werden, heißen **Sortierverfahren**
- Beispiele
  - Telefonverzeichnisse (lexikographischer sortiert)
  - Fahrpläne (sortiert nach der Abfahrtszeit)

Sortierverfahren

- **Komplexität**
  Anzahl der Schritte um n Elemente zu ordnen
  (im besten / durchschnittlichen / schlechtesten Fall, abhängig von der Vorsortierung der Elemente

- **Speicherbedarf beim Sortieren**
  Wird kein zusätzlicher Speicherbedarf benötigt ist ein Sortierverfahren *in place*

- **Stabilität**
  Ein Sortierverfahren ist stabil, wenn es die ursprüngliche Reihenfolge zweier gleicher Elemente beim Sortieren nicht verändert

- **Art des Algorithmus**
  entweder vergleichsbasiert oder die Position in der Sortierung kann errechnet werden (z.B. Bucketsort, Radixsort)

Komplexität von Sortierverfahren

- Gesucht wird eine **obere Schranke** für die Anzahl von **Schlüsselvergleichen**, die im **schlechtesten Fall** notwendig sind, um *n* Objekte zu sortieren

Typische Komplexitätseigenschaften von Sortierverfahren:

O (n log n)	gute Verfahren
O (n^2)	einfache Verfahren

Zwei Aufwandsfunktionen…

Aufwand(n): f(n) …

n

$f(n) = n^2$

→ **$O(n^2)$**

… und g(n)

n

$g(n) = n (\log n) + 2$

→ **$O(n \cdot \log n)$**

… von denen eine die andere dominiert

Ab einem bestimmten Punkt liegen die Werte der Funktion g **immer** unter denen von f

Bis zu diesem Punkt kann der Aufwand der Funktion f geringer sein, obwohl sie die Funktion **g** für große ***n*** dominiert

Prinzipien von vergleichsbasierten Sortierverfahren

- **Vertauschen**
  Vertausche Einträge, wenn sie wechselseitig falsch angeordnet sind (Bubble-Sort, Shaker-Sort)

- **Einfügen**
  Stelle eine sortierte Teilfolge her und füge die verbleibenden Elemente an der richtigen Stelle ein (Insertion-Sort, Shell-Sort)

- **Auswahl**
  Suche das Element, das an Platz 1 gehört, an Platz 2 usw (Selection-Sort, Heap-Sort)

- In allen Fällen sind die Grundoperationen **Vergleich zweier Elemente** und **Bewegen eines Elementes** erforderlich

Beispiel Bubble-Sort

- Vergleich zwei benachbarte Elemente und vertauscht sie, falls sie in der falschen Reihenfolge vorliegen
- Wird solange wiederholt, bis keine Vertauschungen mehr nötig sind
- Hierzu sind in der Regel mehrere Durchläufe erforderlich
- Elemente steigen wie Blasen im Wasser auf (daher der Name)

55	07	78	12	42	1. Durchlauf
07	55	78	12	42	
07	55	78	12	42	
07	55	12	78	42	
07	55	12	42	78	2. Durchlauf
07	55	12	42	78	
07	12	55	42	78	
07	12	42	55	78	3. Durchlauf
07	12	42	55	78	
07	12	42	55	78	4. Durchlauf
07	**12**	**42**	**55**	**78**	Ergebnis

nach © wikipedia.de

Implementationsskizze Bubble-Sort

```
Für eine Liste L mit n Elementen:

Schleife über durchlauf = 1 .. n-1
 {
    Schleife über position = 1 .. n - durchlauf
    {
        Falls L[position] > L[position+1] ...
        {
           ... vertausche L[position] und L[position+1]
        }
    }
 }
```

Optimierungs-
möglichkeit?

Komplexität:

- Der Algorithmus besitzt eine **quadratische Komplexität** ($O(n^2)$) und ist daher langsamer als viele andere Sortieralgorithmen

Vorteile:

- Geringer Speicherbedarf (in-place-Verfahren), (potenziell) stabil
- Einfach und für kleine Elementmengen gut geeignet

Beispiel Quicksort

- Entwickelt 1962 von Hoare
- Ein schneller, rekursiver, nicht-stabiler Sortieralgorithmus nach dem Prinzip **Teile und Herrsche** (engl. *divide and conquer*)

- Prinzip:
  1. Ein beliebiger Schlüssel **x** wird aus den Elementen der zu sortierenden Reihe ausgewählt (**Pivot-Element**)
  2. Die Reihe wird anschließend in eine Reihe mit Schlüsseln **> x** und eine mit Schlüsseln **< x** zerlegt (Divide-Schritt)
  3. Beide resultierenden Teilreihen müssen um **mindestens ein Element kleiner** sein als die zu sortierende Reihe und werden rekursiv bis auf Elementebene in gleicher Weise behandelt (Conquer)
  4. Die miteinander verketteten Teilreihen bilden insgesamt eine sortierte Gesamtreihe

Funktionale Definition von Quicksort

```
ALGORITHM Quicksort(S)
   IF |S| = 1
         THEN RETURN S
   ELSE
         Divide: Wähle irgendeinen Schlüsselwert x = s_j.key aus S aus.
               Berechne eine nichtleere Teilfolge S1 aus S mit den
               Elementen, deren Schlüsselwert ≤ x ist und eine
               nichtleere Teilfolge S2 mit Elementen ≥ x.
         Conquer: S1′ = Quicksort(S1); S2′ = Quicksort(S2);
         Merge: RETURN Concat(S1′, S2′)
   END;
```

Erläuterung „funktional“: Quicksort wird hier als eine mathematische Funktion angesehen, die jeweils eine Sequenz von Werten als Argument erhält und eine (andere) Sequenz als Ergebnis liefert. Die gelieferten Sequenzen müssen abschließend noch verkettet werden (Concat).

Veranschaulichung Quicksort

wähle Schlüsselelement

Teile (Aufspalten in zwei Reihen)

Herrsche (rekursive Anwendung)

In-Place Quicksort

- wähle ein Element der Folge als Pivotelement x;
- setze den Index i auf das erste und j auf das letzte Element der Folge;

- wiederhole solange i <= j

  erhöhe Index i bis zum nächsten Element a_i mit a_i >= x;

  erniedrige Index j bis zum nächsten Element a_j mit a_j <= x;

  falls i <= j

  vertausche a_i und a_j;

  erhöhe i und erniedrige j um eine Position;

- Wende InPlaceQuicksort auf resultierende Teilfolgen mit Länge > 1 an

Veranschaulichung In-Place Quicksort

9	5	1	12	8	10	6	3

wähle Pivotelement: x = 9

i j

setze Indizes

i j

Indizes werden nicht verschoben: 9>=9 und 3<=9

3	5	1	12	8	10	6	9

weil i <= j: vertausche Elemente…

1 → i j ← 1

… und versetze i und j jeweils um eine Position

i j

verschiebe Indizes (nur i wird verschoben)

3	5	1	6	8	10	12	9

weil i <= j: vertausche Elemente…

1 → i j ← 1

… und versetze i und j jeweils um eine Position

j i

verschiebe Indizes (beide werden verschoben)

3	5	1	6	8	10	12	9

weil i > j: zwei Teilfolgen gebildet, rekursiv weiter…

Zur Wahl des Pivotelements

- Idealerweise sollte der Median gewählt werden

- Wählt man bei einer bereits sortierten Reihe immer das linke Element als Pivot, wird in jedem Rekursionsschritt die Reihe nur um ein Element verkürzt

- Komplexität
  Durschnitt: $O(n \bullet \log n)$
  Worst case: $O(n^2)$

- Wenn die Implementierung es zulässt (schneller Indexzugriff), sollte deshalb das räumliche mittlere Element gewählt werden, um eine Degeneration auszuschließen

- Quicksort ist eines schnellsten Sortierverfahren und leicht zu implementieren

$O(n^2)$ VS $O(n \log n)$

- Bei einer Sortierung von 10 Millionen Zahlenwerte
- Auf einem Rechner der 10^8 elementare Schritte pro Sekunde ausführen kann

- **$O(n^2)$**: 10 Millionen^2 = $10^7 * 10^7 = 10^{14} / 10^8 = 10^6$ Sekunden ≈ **11 Tage**
- **$O(n \log n)$**: $10^7 * \mathrm{ld}(10^7) = 10^7 * 24 / 10^8$ ≈ **2,4 Sekunden**

- Ergo: Die Mühe lohnt sich für große Datenmengen sehr!

Sortieren mit linearer Ordnung

Voraussetzungen:

- Die Menge aller Schlüssel ist hinreichend klein
- Es muss nicht ‚in situ' sortiert werden

Sortieren mit **Bucket-Sort**:

- Lege für jeden Schlüssel eine Liste an
- Mache die Listen zu Elementen eines Arrays, das mit dem Schlüssel indiziert werden kann
- Gehe die zu sortierenden Elemente einzeln durch und hänge jedes an die zu seinem Schlüssel gehörige Liste an
- Verbinde die Listen zu einer Ergebnisliste

- Das Verfahren ist **linear in n**, da der Aufwand für das Durchlaufen der Ausgangsliste und des Schlüssel-Arrays linear von der Zahl der Listenelemente abhängt

Parade der Sortierverfahren

Durchschnittskomplexitäten:

Bubble-Sort	$O(n^2)$
Shaker-Sort	$O(n^2)$
Selection-Sort	$O(n^2)$
Insertion-Sort	$O(n^2)$
Shell-Sort	$O(n^{1.2})$
Merge-Sort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heap-Sort	$O(n \log n)$
Bucket-Sort	$O(n)$

Literatur

N. Wirth: Algorithmen und Datenstrukturen, Teubner

G. Saake, K.-U. Sattler: Algorithmen und Daten-strukturen: Eine Einführung mit Java, dpunkt

R.H. Güting, S. Dieker: Datenstrukturen und Algorithmen, Teubner

Demo-Beispiele:

https://visualgo.net/sorting
http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

Zwischenfazit

1

Sortieren ist eine Standardaufgabe in der Informatik.

2

Es gibt verschiedene Sortierverfahren mit speziellen Stärken und Schwächen hinsichtlich Komplexität, Speicheraufwand und Stabilität

3

Bubble-Sort ist ein sehr einfach zu implementierender Algorithmus mit schlechter Effizienz, höchstens für kleine Sammlung einsetzbar

4

Quicksort ist einer der schnellsten In-Place-Sortieralgorithmen, er kann jedoch bei bereits sortierten Sammlungen degenerieren

Überblick

1 Stacks und Queues

2 Sortieren

3 **Graphen**

Anwendungsfälle Graphen

- Graphen spielen in vielen Anwendungen eine Rolle, z.B.
  - Streckennetze öffentlicher Verkehrsmittel
  - Vernetzte Rechner
  - Logistik
  - Social Media

- Beispiel Fluggesellschaft:
  - Gibt es zwischen zwei Flughäfen ausschließlich Flügen der eigenen Gesellschaft?
  - Flughäfen sind Knoten in einem Graphen
    Eigene Flugverbindungen sind Kanten in einem Graphen

TRANSPORTATION
CLUSTERS
3.200 airports
60.000 routes
EUROPE
GEOGRAPHICAL LAYOUT
Color = Longitude
Size = Number of routes
Alaska
Canada
Russia
NORTH AMERICA
ASIA
AFRICA
India
MIDDLE EAST
LATIN AMERICA
Pacific islands
OCEANIA
CC-BY martingrandjean.ch 2016
Data: openflights.org

Was sind Graphen?

- Ein Graph stellt eine Menge von Objekten mit einer Relation auf diesen Objekten dar.
- Formal: Ein (gerichteter) Graph G ist ein Paar **G = (V, E),** wobei gilt:
  - **V** ist eine endliche Menge von Knoten (engl.: **vertex**, daher **V**)
  - **E** ist eine endliche Menge von Kanten (engl.: **edge**, daher **E**).
    - formal: **E** ist eine zweistellige Relation auf **V, E** $\subseteq$ **V x V**.

- Bildliche Darstellung
  - eines Knotens durch einen kleinen Kreis
  - einer gerichteten Kante durch einen Pfeil
  - V = {1, 2, 3}
  - E = {(1,2), (2,1), (1,3), (2,3)}

Typen von Graphen

Gerichteter Graph

Ungerichteter Graph (mit symmetrischen Beziehungen)

nicht zusammenhängend

Vollständiger Graph

Bewerteter (gewichteter) Graph

Alle vier Graphen sind **zyklisch**: Es gibt mindestens einen Pfad, auf dem ein Knoten zweimal vorkommt

Implementierung von Graphen

- Zwei beliebte Methoden:
  - Adjazenzmatrix
  - Adjazenzlisten

- Zwei Knoten u,v sind adjazent (=benachbart), wenn sie durch eine Kante verbunden sind

Implementierung Adjazenzmatrix

- Für ungewichtete Graphen:
  A_{ij} = true (1), falls Kante von i nach j

	j = 1	2	3	4	5
i = 1	0	1	0	1	0
2	1	0	0	1	0
3	0	0	1	0	1
4	0	0	0	0	1
5	0	0	1	0	0

	j = 1	2	3	4	5
i = 1	∞	6	∞	4	∞
2	5	∞	∞	11	∞
3	∞	∞	3	∞	1
4	∞	∞	∞	∞	9
5	∞	∞	13	∞	∞

- Bei gewichtete werden die „Kosten“ für Zustandsübergang von i nach j eingetragen
- „unendlich“ für „keine Kante“

Implementierung Adjazenzlisten

- Jeder Knoten verfügt über eine Liste seiner Nachfolgerknoten
- Zusätzlich möglich: Alle Knoten sind über ein Array direkt zugreifbar

- Geringer Platzbedarf, Zugriff auf alle Nachfolgerknoten in Linearzeit
- Bei Bedarf können auch **inverse Adjazenzlisten** (Listen von Vorgängerknoten) vorgesehen werden

Implementierung Adjazenzlisten

(objektorientiert)

- Jeder Knoten verfügt über eine ~~Liste~~ Menge seiner Nachfolgerknoten
- Zusätzlich möglich: Alle Knoten sind über ~~ein Array~~ eine Menge direkt zugreifbar

Objektorientierte Modellierung

- Eine Klasse modelliert die **Knoten**
- Mehrere Möglichkeiten für **Kanten**:
  - Implizit durch Referenzen auf Nachbarn
  - explizit als eigene Klasse modelliert
- Eine Klasse modelliert **Graphen**:
  - Hält eine Menge von Knoten

- Geringer Platzbedarf, Zugriff auf alle Nachfolgerknoten in Linearzeit
- Bei Bedarf können auch **inverse Adjazenzlisten** (Listen von Vorgängerknoten) vorgesehen werden

Realisierung in Java

- Wenn jeder Knoten eine Menge seiner Nachbarknoten hält, ist ein gerichteter Graph automatisch realisierbar

- Die Kanten werden direkt mit Java-Referenzen realisiert

```
public class Knoten
{
    private Set<Knoten> _nachbarn;

    public Knoten() {
        _nachbarn = new HashSet<Knoten>();
    }

    public boolean hatVerbindungZu(Knoten k)
    {
        return _nachbarn.contains(k);
    }

    public void verbindeMit(Knoten k)
    {
        _nachbarn.put(k);
    }

    public void trenneVerbindungZu(Knoten k)
    {
        _nachbarn.remove(k);
    }
}
```

Graph mit einer Menge von Knoten

```
class Graph
{
    private Set<Knoten> _knotenmenge;

    public Graph()
    {
        _knotenmenge = new HashSet<Knoten>();
    }

    public void neuerKnoten(Knoten k)
    {
        _knotenmenge.add(k);
    }

    public void verbindeDirekt(Knoten k1, Knoten k2)
    {
        k1.verbindeMit(k2);
        k2.verbindeMit(k1);
    }

    public boolean enthaelt(Knoten k)
    {
        return _knotenmenge.contains(k);
    }
}
```

Bei einem ungerichteten Graphen muss eine Verbindung zweier Knoten wechselseitig eingetragenen sein

Abbildung von gewichteten Kanten mit Maps

- Anforderung:
  Für jede Kante soll ein Gewicht (bspw. ein int-Wert) gehalten werden

- Eine Abbildung (engl.: Map) ist eine Menge von Schlüssel-Wert-Paaren

- Der **Schlüssel** ist (wie die Elemente in einer Menge) **eindeutig**

- Der **Wert** kann ein **beliebiges Objekt** sein (muss nicht eindeutig sein)

Kantengewichte: Map statt Set

- Ein Knoten enthält eine Abbildung von Nachbarn auf Werte
- Die **Nachbarknoten** sind die **Schlüssel** für die Abbildung
- Für das **Kantengewicht** **Integer-Objekt** verwendet

```
class Knoten
{
    private Map<Knoten,Integer> _nachbarn;

    public Knoten() {
        _nachbarn = new HashMap<Knoten,Integer>();
    }

    public boolean hatVerbindungZu(Knoten k)
    {
        return _nachbarn.keySet().contains(k);
    }

    public void verbindeMit(Knoten k, int gewicht)
    {
        _nachbarn.put(k,new Integer(gewicht));
    }

    public int gibVerbindungsgewicht(Knoten k)
    {
        Integer gewicht = _nachbarn.get(k);
        if (gewicht == null)
            return 0;
        else
            return gewicht.intValue();
    }
}
```

Durchläufe in Graphen

- Oft soll jeder Knoten nur einmal bearbeitet werden
- Wiederholtes Erreichen von Knoten möglich:
  - mehrere Pfade
  - Zyklen

Buchführung erforderlich

Durchläufe in Graphen

- Prinzip für Durchlauf:
  - **Breitendurchlauf** oder **Tiefendurchlauf** auf die erreichbaren Knoten
  - **Markiere** dabei **besuchte Knoten** und vermeide Mehrfachbesuche

- Tiefendurchlauf ist rekursiv einfach zu implementieren
- „Schon besucht“-Markierungen müssen vor neuem Durchlauf zurückgesetzt werden
- Alternative: Markierungen mit jedem Durchlauf umkehren
- Die Buchhaltung kann auch beim Besucher erfolgen

Kürzester Pfad durch einen Graphen

- Frage nach dem kürzesten Weg zwischen zwei Knoten in gewichteten Graphen

- Beispiele:
  - Suche nach dem kürzesten Weg von S nach Z in einem Straßennetz
  - in einem Netz von Versorgungsleitungen oder
  - zum Routing im Internet

Suche nach dem kürzesten Pfad

- Viele Algorithmen beruhen auf dem Optimalitätsprinzip:
  - Ist $\mathbf{p = (u_0, u_1, \ldots, u_z)}$ ein kürzester Pfad von Knoten $\mathbf{u_0}$ nach Knoten $\mathbf{u_z}$,
  - so ist $\mathbf{p' = (u_i, \ldots, u_j), 0 \leq i < j \leq z}$, auch immer ein kürzester Pfad von $\mathbf{u_i}$ nach $\mathbf{u_j}$

Um einen kürzesten Pfad zu finden,
kann man von kürzesten Teilpfaden ausgehen

Dijkstras „Wellen“-Algorithmus

- Dijkstras Idee (1959):
  - Vom Startknoten eine „äquidistante Welle“ aussenden, die sukzessiv weitere Knoten erfasst, bis Zielknoten erreicht ist
- Es werden **drei Knotenmengen** unterschieden:
  - für „gewählte Knoten“ **G** ist kürzester Weg vom Startknoten s bekannt
  - für „Randknoten“ **R** ist ein Weg von s bekannt
  - für „unerreichte Knoten“ **U** kennt man noch keinen Weg

Dijkstras Algorithmus

1. **Initialisierung**

Pfadlänge aller Knoten außer s ist ∞, G = {s}, R = {Nachfolger von s}

2**. Berechne Wege ab s**

- Falls R leer ist, ist Zielknoten nicht von s erreichbar, brich ab
- Entferne **nächstgelegenen** Randknoten u aus R und füge ihn in G ein
- **Falls u Zielknoten** ist, gib **Vorgängerliste** von u bis s aus und brich ab
- Füge alle **Nachfolger** von u in R ein, **die nicht in G** sind
- Vermerke **Pfadlänge und Vorgänger** für kürzesten Pfad zu allen Nachfolgern von u in R
- Wiederhole 2

Beispiel für Dijkstras Algorithmus

Notation: (Name, Pfadlänge, Vorgänger)

Gewählt	Randknotenliste (sortiert)
(s 0 s)	(c 13 s) (a 14 s)
(c 13 s)	(a 14 s) (f 21 c) (g 23 c) (b 24 c)
(a 14 s)	(b 20 a) (f 21 c) (g 23 c) (d 31 a)
(b 20 a)	(f 21 c) (g 23 c) (d 31 a)
(f 21 c)	(g 23 c) (e 30 f) (d 31 a)
(g 23 c)	(e 30 f) (d 31 a) (i 45 g)
(e 30 f)	(d 31 a) (h 37 e) (i 45 g)
(d 31 a)	(h 37 e) (i 45 g)
(h 37 e)	(i 45 g) (z 46 h)
(i 45 g)	(z 46 h)
(z 46 h)	

Kürzerer Teilpfad nach b wird entdeckt, Vorgänger und Pfadlänge werden gemerkt.

Ergebnis: s, c, f, e, h, z

Zusammenfassung

1. Ein Graph besteht aus **Knoten** und **Kanten**. Kanten können **gerichtet** oder **ungerichtet** sein.

2. In einem **gewichteten Graphen** haben die Kanten Gewichte. **Graphendurchläufe** besuchen jeden erreichbaren Knoten (mindestens) einmal.

3. Ein häufig zu lösendes Problem ist, den **kürzesten Weg** in einem Graphen von einem Knoten zu einem anderen zu finden.