Algorithmen und Datenstrukturen

Einführung:

Zielstellung ♦ Organisatorisches

Algorithmen ♦ Datentypen und -strukturen

Inhalte

- Typische, grundlegende Algorithmen
  - auf Sequenzen und Matrizen
  - auf Bäumen und Graphen
  - auf Punktmengen

  } **Datenstrukturen**

- Algorithmische Paradigmen
  - Teile und Herrsche (Divide and Conquer)
  - Dynamische Programmierung
  - Greedy-Algorithmen
  - Branch and Bound

Vorausgesetzte Kenntnisse

- Inhalte des Kurses *Grundlagen der Programmierung*
  - Programmierung in Python
    - Kontrollstrukturen
    - Funktionen/Prozeduren
  - Algorithmisches Denken
    - Entwurf einfacher Algorithmen
    - Korrektheit, Terminieren, Effizienz
    - Iteration und Rekursion
  - Graphen
    - Repräsentation von Graphen
    - Abstand von Knoten (Brute-Force, BFS, DFS)
  - Prinzipieller Aufbau einer Rechners (Speicher – Prozessor)

Vorlesung – Übung

- **Vorlesung**
  - Vermittlung der **Konzepte** und **Algorithmen**
  - Voraussetzung für erfolgreiche Teilnahme an den Übungen und der Prüfung
- **Übung**
  - Vertiefung von Teilen des Vorlesungsstoffs, „Training“
  - Finden von Lösungsansätzen
  - Implementieren von Algorithmen
- **Verwendete Sprachen:**
  - **Python**
  - Pseudocode

Termine

- Keine Vorlesungen: 30.04., 21.05.
- Übungen ab morgen!!!
- Keine Übungen:
  - Dienstag, 01.05.
  - Mittwoch, 09.05.
  - Freitag, 11.05.

Ablauf der Vorlesungen

- **Folien**
  - enthalten alle Begriffe, Definitionen, Aussagen und einige Beispiele und Erklärungen
  - aber bei weitem nicht alle relevanten Informationen

- **Tafel**

  Viele Beispiele, Überlegungen und Erklärungen werden schrittweise an der Tafel entwickelt.

  Schreiben Sie mit! Das wird Ihnen helfen!

Ablauf der Übungen

- Übungsaufgaben, die Sie vorher zu Hause lösen
- Vorstellung Ihrer Lösungen (in kleinen Schritten)

  Bonuspunkte *für die Klausur durch das Vorstellen von Lösungsschritten*

- *und das Wichtigste ...*
  Diskussion Ihrer **Fragen** zum Vorlesungsstoff

  Trauen Sie sich, Fragen zu stellen und Beiträge zu den Übungsaufgaben zu präsentieren!!!

„Rechnerübungen“

- keine Präsenzübungen
- *Implementieren Sie die behandelten Algorithmen zu Hause selbstständig!*
- Von Zeit zu Zeit fordern wir die Abgabe Ihrer Implementierung in **Python** via Moodle.
  - Arbeiten Sie in **Teams aus 2-3 StudentInnen**. (*Teambildung über Moodle*)
  - Die Abgaben sind „vorgezogene Klausuraufgaben“: Sie erhalten dafür Punkte für die Klausur (insgesamt **20 %**).

Leistungserfassung

- **Klausur**
  - Mittwoch, 1. August, 9:30 – 11:30 Uhr im H05
  - Damit erwerben Sie die übrigen **80%** der Klausurpunkte.

- **Bonuspunkte** aus den Übungen zählen für die Klausur. Die Klausur muss aber ohne diese bestanden werden!

- Zum Bestehen der Klausur müssen mindestens 50 Punkte (ohne Bonus, aber inklusive der Punkte aus den Implementierungen) erreicht werden.

Informationszugang

- **Webseite:**
  http://www.cs.uni-potsdam.de/bordihn/teaching/ss18/aud/announce.php

- **Moodle:**
  https://moodle2.uni-potsdam.de/
  - alle Lehrmaterialien (Folien, Übungsaufgaben und -blätter)
  - aktuelle Informationen
  - Abgaben
  - Forum

  Nutzen Sie die Kurssuche und schreiben Sie sich ein.

  Einschreibeschlüssel: aud18

PULS

1. Belegen

- Vorlesung
- Übung (genau eine Gruppe)
- Zulassungen erfolgen laufend, sofern Platz in den Gruppen ist

2. Prüfungsanmeldung

- mindestens acht Werktage vor dem Prüfungstermin (also vor dem Klausurtermin)
- Sie werden von uns zugelassen.
- Ohne Anmeldung keine Klausurteilnahme!!!

Fragen ?!

Algorithmen und Datenstrukturen

Einführung:

Algorithmen ♦ Datentypen und -strukturen

Algorithmen

- Kern zur Lösung von Problemen und Aufgaben mit den Mitteln der Informatik
- **Programme** realisieren Algorithmen (in einer bestimmten Programmiersprache).
- Algorithmen beschreiben die Problemlösung *unabhängig* von Programmiersprachen.
- Dabei werden im Allgemeinen **Daten** verarbeitet.
  - **Eingabedaten** $\xrightarrow{\textit{Abbildung}}$ **Ausgabedaten**
  - **Anweisungsfolge**

Probleme und Algorithmen

- **Spezifikation des Problems**
  - **Eingabe:** Daten, die dem Algorithmus als Eingabedaten gegeben werden
  - **Ausgabe:** Daten, die der Algorithmus aus den Eingabedaten berechnet und damit die Problemstellung beantwortet
- **Algorithmus** beschreibt, *wie* die Eingabedaten in zugehörige Ausgabedaten transformiert werden

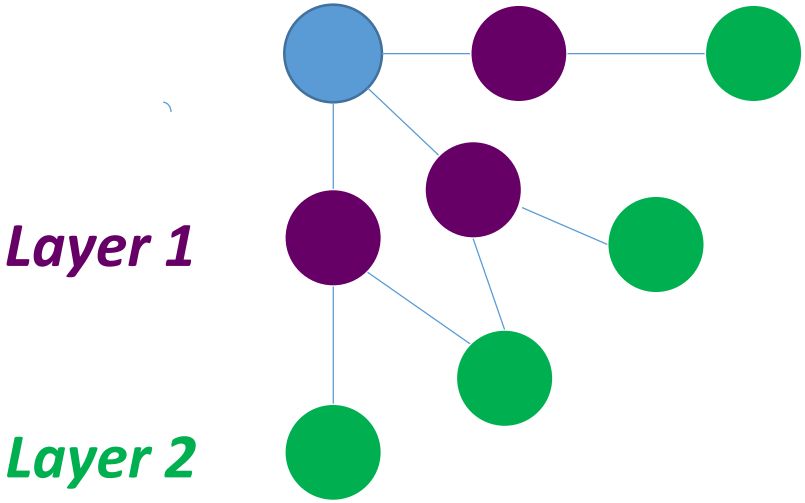
Beispiele

- schriftliche Addition, Multiplikation, Division, ...
- Euklidischer Algorithmus
- Abstand von Knoten in Graphen
- ...

- Algorithmen *unabhängig* von Quelle der Daten
  → **Annahme:** Daten liegen im Speicher vor.
- Algorithmen *abhängig* von Repräsentation der Daten

Wdh.: Breitensuche in Graphen

- zuerst alle Nachbarn eines Knotens bestimmen (*Layer 1*)
- dann für alle Knoten aus *Layer 1* alle (neuen) Nachbarn bestimmen (*Layer 2*)
- usw.



Wdh.: Breitensuche (Markierung)

Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$
in Adjazenzlisten-Repräsentation, $u \in V$

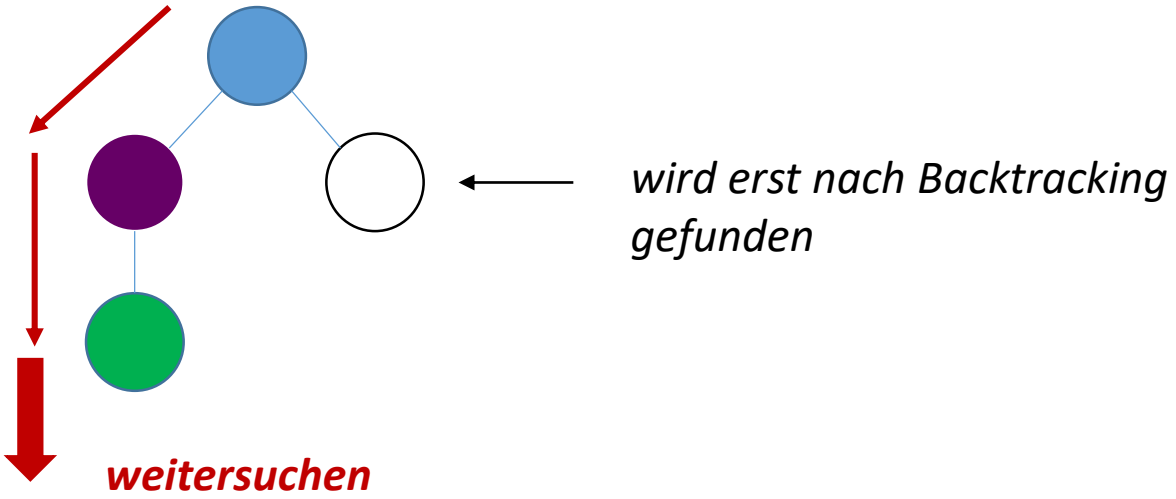
```
Q ⟵ leere Warteschlange          # für Knoten, die noch unmarkierte
                                 # Nachbarn haben könnten
für alle  i ∈ V
          mark[i] ⟵ 0
mark[u] ⟵ 1
enqueue(Q,u)
solange Q nicht leer ist
          j ⟵ dequeue(Q)         # j wird untersucht
          für alle k in adj[j]
                    falls mark[k] = 0
                              mark[k] ⟵ 1
                              enqueue (Q,k)   # neuer Knoten gefunden
```

Algorithmen *versus* Datenstrukturen

- Algorithmen manipulieren Daten
- Komplexe Daten müssen in geeigneten **Datenstrukturen** vorliegen und verwaltet werden
  - *am Bsp.:* Adjazenzlisten; Warteschlange
  - bedingen Korrektheit und Effizienz der Algorithmen
- Manipulation der Daten durch für die gewählten Datenstrukturen spezifischen **Operationen**
  - *am Bsp. Liste:* Zugriffsoperation adj[*j*]
  - *am Bsp. Schlange:* enqueue, dequeue

Wdh.: Tiefensuche in Graphen

- von jedem gefundenen Knoten sofort einen neuen Nachbarn suchen
- erst, wenn so kein neuer Knoten gefunden werden kann, zurückgehen zum zuletzt gefundenen Knoten, der noch weitere Nachbarn haben kann: **Backtracking**



Wdh.: Tiefensuche (Markierung)

Eingabe: ungerichteter, schlingenfreier Graph $G = (V,E)$
in Adjazenzlisten-Repräsentation, $u \in V$

für alle $i \in V$
    $mark[i] \longleftarrow 0$
$S \longleftarrow$ leerer Stack
$mark[u] \longleftarrow 1$
push(S,u)
solange S nicht leer ist
    $akt \longleftarrow$ **top**(S)
    falls $k \in$ adj$[akt]$ existiert UND $mark[k] = 0$
        $mark[k] \longleftarrow 1$
        push(S,k)
    sonst
        pop(S)

Abstrakte Datentypen (ADT)

- **Datentypen** sind definiert durch
  - die Menge der darstellbaren **Werte** und
  - die ausführbaren **Operationen**
- **ADT**: Abstraktion von der Art, wie die Werte gespeichert und die Operationen ausgeführt werden
  - nur, welche Operationen erlaubt sind
  - *unabhängig von Realisierung in Programmiersprachen*
- Bereitstellung der Information, die für den Anwendungsprogrammierer relevant ist

Datenstrukturen

- Implementierungen eines ADT
  - feste Darstellung der Werte
  - Realisierung der Operationen in einer Programmiersprache
- verschiedene Implementierungen eines ADT
- Datentypen mit den gleichen Operationen können in Algorithmen gegeneinander ausgetauscht werden.
- Implementierung kann die Laufzeit eines Algorithmus beeinflussen, der den ADT verwendet

Queue (Warteschlange) – informal

- **Beschreibung:** FIFO-Liste
- **Wertebereich:** Menge aller endlichen Folgen von Elementen des Grundtyps
- **Operationen:**

Operation	gibt zurück	Verhalten
empty()	Queue	erzeugt leere Warteschlange
isEmpty(*Q*)	bool	entscheidet, ob *Q* leer ist
enqueue(*Q*,*x*)	Queue	reiht *x* in *Q* ein
dequeue(*Q*)	**???**	löscht Frontelement aus *Q* und gibt es zurück

Queue (Warteschlange) – informal

Schnittstelle/Interface

Operation	gibt zurück	Verhalten
empty()	Queue	erzeugt leere Warteschlange
isEmpty(*Q*)	bool	entscheidet, ob *Q* leer ist
enqueue(*Q*,*x*)	Queue	reiht *x* in *Q* ein
delete(*Q*)	Queue	löscht Frontelement aus *Q*
top(*Q*)	Grundtyp	Wert des Frontelements

dequeue(*Q*) ≙ top(*Q*); delete(*Q*)

Formale Spezifikation eines ADT-Interface

type Σ =

sorts *verwendete Typen*

functions *Namen, Definitions- und Wertebereiche in der Form*

$f : A \rightarrow B$

end.

Die Bedeutung der Symbole ergibt sich erst durch eine Interpretation.

Interface ADT Queue

type Queue =

sorts T, bool, q

functions

empty: $\rightarrow$ q

isEmpty : q $\rightarrow$ bool

enqueue: q $\times$ T $\rightarrow$ q

delete: q $\rightarrow$ q

top: q $\rightarrow$ T

end.

Operation	gibt zurück
empty()	Queue
isEmpty(*Q*)	bool
enqueue(*Q*,*x*)	Queue
delete(*Q*)	Queue
top(*Q*)	Grundtyp

Interface ADT Boolean

type Boolean =

sorts bool

functions

t : $\rightarrow$ bool

f : $\rightarrow$ bool

not: bool $\rightarrow$ bool

and: bool $\times$ bool $\rightarrow$ bool

or: bool $\times$ bool $\rightarrow$ bool

end.

Interpretation

- Das Verhalten eines ADT wird festgelegt, indem
  - den Sorten konkrete Wertemengen und
  - den Funktionssignaturen konkrete Abbildungsvorschriften

  zugeordnet werden.

- Es gibt verschiedene formale und semi-formale Methoden, eine Interpretation anzugeben.

Interpretation für den ADT Boolean

- Zuordnung von Wertemengen zu den Sorten

bool $\rightarrow$ {true, false}

- Definition der Funktionen

t() = true

f() = false

not(true) = false, not(false)=true,

and(false, false) = and(false, true) = and(true, false) = false,

and(true, true) = true,

or(false, false) = false,

or(false, true) = or(true, false) = or(true, true) = true

Ausblick: Formale Spezifikation von Datentypen

- Zuordnung von Wertemengen zu den Sorten
- Festlegung des Verhaltens durch Gesetze
- Beispiel Boolean:

$$\text{bool} \rightarrow \{\text{true}, \text{false}\}$$

$$\text{not}(\text{true}) = \text{false}, \quad \text{not}(\text{false}) = \text{true},$$

$$\forall x.\ \text{and}(\text{false}, x) = \text{false},$$

$$\forall x.\ \text{and}(\text{true}, x) = x,$$

$$\forall x\ \forall y.\ \text{or}(x, y) = \text{not}(\text{and}(\text{not}(x), \text{not}(y)))$$

Man kann nun beweisen, dass bis auf Isomorphie nur ein Datentyp existiert, der die Spezifikation erfüllt.

Ausblick: Gesetze Beispiel Queue

isEmpty(empty)

isEmpty(delete(empty))

$\forall q\ \forall x.$ not(isEmpty(enqueue(q, x)))

$\forall q\ \forall x.$ isEmpty(q) $\Rightarrow$ isEmpty(delete(enqueue(q, x)))

$\forall q\ \forall x.$ isEmpty(q) $\Rightarrow$ top(enqueue(q, x)) = x

$\forall q\ \forall x.$ not(isEmpty(q)) $\Rightarrow$ top(enqueue(q, x)) = top(q)

Semi-formale Interpretation

Operation

pre: Bedingungen, die vor Ausführung der Operation erfüllt sein müssen (**Vorbedingungen**)

post: Bedingungen, die nach Ausführung der Operation garantiert sind (**Nachbedingungen**)

Semi-formale Interpretation: Queue

T: Wertemenge des Grundtyps
q: Menge aller endlichen Folgen von Elementen von **T**
bool: ADT Boolean

empty()
post: eine neue leere Queue ist erzeugt

isEmpty(*Q*)
post: true, falls *Q* keine Elemente enthält, sonst false

type Queue =

sorts T, bool, q

functions

empty: $\rightarrow$ q

isEmpty : q $\rightarrow$ bool

enqueue: q $\times$ T $\rightarrow$ q

delete: q $\rightarrow$ q

top: q $\rightarrow$ T

end.

Semi-formale Interpretation: Queue

enqueue(*Q*,*x*)

post: Der Schlange ***Q*** ist das Element ***x*** vom Typ **T** als neues letztes Element hinzugefügt.

delete(*Q*)

post: Das erste Element ist aus *Q* entfernt.

top(*Q*)

pre: isEmpty(*Q*) = false

post: Das erste Element aus *Q* ist zurückgegeben. *Q* ist unverändert.

type Queue =

sorts T, bool, q

functions

empty: $\rightarrow$ q

isEmpty : q $\rightarrow$ bool

enqueue: q $\times$ T $\rightarrow$ q

delete: q $\rightarrow$ q

top: q $\rightarrow$ T

end.

ADT – Datenstruktur

Schnittstelle des Datentyps
(und ggf. Gesetze)

Interpretation der Sorten- und Funktionssymbole

ADT

Verhalten des Datentyps

Implementierung

verwendbarer Datentyp

Datenstruktur