Lena Kästner
Nico Möller

Informatik A: Algorithmen

Kapitel 1: Einführung

Behandelte Probleme

Beispiele für Algorithmen in umgangssprachlicher Form.

Terminierung, Korrektheit und Effizienz sind nicht algorithmisch zu bestimmen.

Begriffe

Algorithmus: endlich lange Vorschrift aus Einzelanweisungen (Anweisungsfolge)

Programm: Algorithmus für den Compiler formuliert

Prozess: Programm in der Ausführung

Ablaufprotokoll: genannt `Trace`, gibt an welche Werte *Variablen* zu bestimmten Zeitpunkten annehmen.

Kapitel 2: Java

Behandelte Probleme

Variablen, Konstanten, Kontrollstrukturen und einfache Datentypen

Begriffe

Variablen: Benannte Speicherstellen, deren Inhalt gemäß ihrer vereinbarten Typen interpretiert werden.

Konstanten: Sind unveränderlich, sie können zur besseren Lesbarkeit des Programms verwendet werden. Sie werden mit dem Attribut `final` deklariert.

Kontrollstrukturen: Regeln den dynamischen Ablauf der Anweisungsfolge eines Programms. Bespiele: (*Bedingungen, Verzweigungen* `[if..else, switch..case]`, *Schleifen* `[for, while]`)

Datentyp: legt fest: Wertebereich, Operationen, Konstantenbezeichner
→ Implementierung verlangt *Codierung*

Methode: einer Klasse von Objekten zugeordneter Algorithmus, operieren auf Datenfeldern um deren Zustand zu manipulieren

Datenfelder: den Objekten zugeordnete Daten

Klasse: abstrakter Oberbegriff für die Beschreibung der gemeinsamen Struktur und des gemeinsamen Verhaltens von Objekten (Klassifizierung)
→ Ansammlung von Objekten gleicher Bauart (Beschreibung)

Objekt: ein Exemplar eines bestimmten Datentypes oder einer bestimmten Klasse; auf Teilaufgaben spezialisiert

Attribute: Eigenschaften eines Objekts

Details

Kontrollstrukturen

`if` Anweisungen: Verzweigungen durch Bedingungen

```
If (Bedingung)
   {Anweisung/en}
   else
     {Anweisung/en}
```

Vergleichsoperatoren:

<	kleiner
<=	kleiner gleich
==	gleich
>	größer
>=	größer gleich
!=	ungleich

logische Operatoren:

&&	und
||	oder
^	exklusives oder
!	nicht

`switch/case` Anweisungen: Verzweigungen durch Fallunterscheidungen

```
switch (Variable){
      case Wert₁:                  Anweisung/en; break;
      case Wert₂: case Wert₃ :     Anweisung/en; break;
      default:                     Anweisung/en;
}
```

`while, do.. while` Schleifen mit Abbruchbedingung, Anzahl der Durchläufe entscheidet sich meist erst zur Laufzeit des Programms.

```
while(Bedigung/en){
      Anweisung/en;
}

do{
   Anweisung/en;
}while(Bedingung/en)
```

Mit `break` und `continue` können die beiden Kontrollstrukturen zusätzlich gesteuert werden

`for` Schleife, wird meist dann verwendet wenn eine Festen Anzahl von Schleifendurchläufen erwünscht ist

```
for (Variable; Bedingung/en; Inkrementation)
 { Anweisung/en; }
```

Einfache Datentypen:

`boolean`	8 Bit	true oder false
`char`	16 Bit	Zeichen Unicode
`byte`	8 Bit	Vorzeichenbehaftete Ganze Zahl
`short`	16 Bit	Vorzeichenbehaftete Ganze Zahl
`int`	32 Bit	Vorzeichenbehaftete Ganze Zahl
`long`	64 Bit	Vorzeichenbehaftete Ganze Zahl
`float`	32 Bit	Gleitkommazahl
`double`	64 Bit	Gleitkommazahl

Darstellungen von einfachen Datentypen

Bei Rechnung mit *ganzen Zahlen* in binär Darstellung Vorzeichenbit verdoppeln = Test auf Bereichsüberschreitung (Vorzeichenbits müssen gleich sein)
Bei negativen Zahlen in Binärdarstellung x --> -x: negiere bitweise, addiere 1

Binärdarstellung für Gleitkommazahlen:

$$(-1)^S \cdot 2^e \cdot 1,f$$

Vorzeichen (1 bit)	Exponent (8 bit)	reduzierte Mantisse (23 bit)	**float**
Vorzeichen (1 bit)	Exponent (11 bit)	reduzierte Mantisse (52 bit)	**double**

float

Die kleinste darstellbare positive Zahl im vereinfachten `float`-Format lautet $2^{-127} \approx 10^{-38}$

0 10000001 00000000000000000000000

Die größte darstellbare positive Zahl im vereinfachten `float`-Format lautet $2^{127} \approx 10^{38}$

0 01111110 11111111111111111111111

Wertebereich

± ∞	Falls e=	127
± 0	Falls e=	-128

double

Die kleinste darstellbare positive Zahl im vereinfachten `double`-Format lautet $2^{-1023} \approx 10^{-306}$

Die größte darstellbare positive Zahl im vereinfachten `double`-Format lautet $2^{1023} \approx 10^{306}$

Wertebereich

± ∞	Falls e=	1023
± 0	Falls e=	-1024

Typumwandlung

Verlustfreie Umwandlungen (*implizit*) sind von „kleineren" zu „größeren" Datentypen möglich. Beispielsweise von `byte` oder `short` nach `int`, Konvertierungen die ggf. verlustbehaftet sind (*explizit*) verlangen einen `Cast`-Operator.

Beispiel:

```
double d = 3.14159265358979; // ca. 15 Stellen Genauigkeit
int    i = 1000000000;       // ca.  9 Stellen
i = (int) d;                 // explizit mit Verlust
```

Wichtige Algorithmen

- `Collatz:` solange `x` gerade `x=x/2`, sonst `x=x*3+1`
- `Euklid (ggt):` solange `x!=y  x=x-y`, sonst `y=y-x`
- `Turbo-Euklid (ggt):` solange `y!=0  z=x%y`, `x=y`, `y=z`

Kapitel 3: Felder

Behandelte Probleme

Zusammenfassen mehrerer Daten zu einem Feld (`Array`).

Begriffe

Feld: Zusammenfassung mehrerer Daten des selben Typs in festgelegter Art und Weise

Allokieren: Besorgen von Speicherplatz für ein Array

endlicher Automat: ein endlicher Automat ist ein 5-Tupel

Details

Allokieren von Speicherplatz

Erfolgt über den Operator `new` folgend vom Typ der Daten die zusammengefasst werden sollen und eckigen Klammern mit Anzahl der Elemente die in diesem Array untergebracht werden sollen.
Beispiel:

```
double[][] m = new double[4][3]; // 4x3 Matrix vom Typ double
```

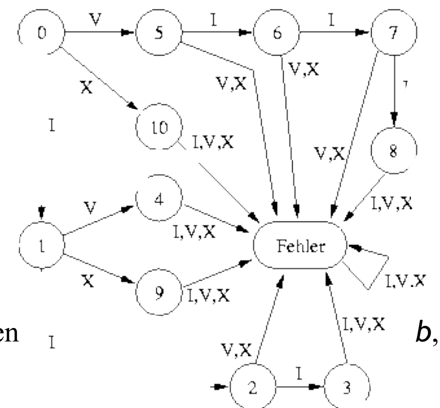
Der Abzählreim

n Kinder stehen in einem Kreis, jedes k-te Kind wird abgeschlagen. Die Kreisanordnung der Kinder wird durch Array realisiert, wobei jeder Index für ein Kind steht und als gespeicherten Wert den Index des aktuellen Nachfolgers enthält.

Der endliche Automat

Ein endlicher Automat ist ein 5- Tupel:

S	= Zustandsmenge,
Σ	= Eingabemenge,
$\delta : S \times \Sigma \rightarrow S$	= Übergangsfunktion,
$s_0 \in S$	= Anfangszustand,
$F \subseteq S$	= Endzustände



Die Wirkungsweise der Überführungsfunktion ± kann durch einen Zustandsüberführungsgraphen beschrieben werden. In diesem Graphen führt eine mit *x* beschriftete Kante von Knoten *a* zu Knoten *b*, falls gilt: $\delta(a, x) = b$.

Beispiel: Ziffernfolge in römische Zahlen umwandeln (s. Bild).

Lineare und binäre Suche

Lineare Suche, Suche einer Zahl in einem unsortierten Array.

→ Laufzeit	`best_case`	1
	`worst_case`	n
	`Durchschnitt`	$\frac{n}{2}$

Binäre Suche, Suche in einem sortierten Array, Lege ein Intervall fest und betrachte da mittlere Element, falls gesuchte Zahl größer, nehme rechtes Teilintervall sonst linkes Teilintervall:

→ Laufzeit	`worst_case`	$\log_2 n$

Wichtige Algorithmen

- `Sieb des Eratosthenes:` streicht (beginnend bei 2) das Vielfache einer jeden Zahl aus gegebener Zahlenfolge, rückt weiter zur nächsten verbleibenden Zahl → Primzahlen bleiben übrig
- `Abzählreim:` Kreisstruktur durch Array, mit Indizes der Nachfolger als gespeicherten Werten
- `Lineare Suche:` durchlaufen und Wert mit gesuchtem vergleichen
- `Binäre Suche:` Sprung in die Mitte, weiter suchen links oder rechts, dort in die Mitte

Kapitel 4: Klassenmethoden

Behandelte Probleme

Zusammenfassung häufig genutzter Algorithmen zu Methoden.
Klassen bezogene Methoden

Begriffe

Klassenvariable: Datenfelder, die mit dem Schlüsselwort static deklariert werden, existieren pro Klasse genau einmal (unabhängig von der Zahl der kreierten Instanzen) und alle Objekte dieser Klasse können auf sie zugreifen.

Modifier: bestimmt ob auf eine Methode auch von einer beliebigen anderen Klasse zugegriffen werden darf.

Rückgabewert: Datentyp, falls ein Wert zurückgeliefert wird, sonst `void`.

formale Parameter: Parameterliste die in die im Methodenrumpf festgelegt werden

aktuelle Parameter: Parameter die beim Aufruf der Methode im Programm übergeben werden

Überladen: mehrere Methoden gleichen Namens, die durch ihre Parameter identifiziert werden, Parameterlisten müssen jedoch unterschiedlich sein (*Signatur*) (Bsp: *Default-Konstruktor* = leere Parameterliste)

Signatur: Methodenname + Parameterliste

Sichtbarkeit von Variablen: nach Beenden einer Methode „verschwinden“ ihre Parameter wieder, abgesehen vom *Rückgabewert*. Zum Zeitpunkt der Ausführung der Methode überdecken die temporären Parameter die der Klasse.

Details

Methodenaufrufe

Beispiele:
ohne Rückgabewert, mit formalen Parametern

```
public static void sterne(int k){
      for(int i = 0; i<k;i++)
            Io.print('*');
      IO.println();
}
```

mit Rückgabewert, mit formalen Parametern

```
public static int zweihoch (int n){
      int h=1;
      for(int i = 0; i<n;i++)
            h=h*2;
      return h;
}
```

Parameterübergabe an Arrays

Bei der Übergabe von Array als Parameter an Methoden, wird nur eine Referenz, also ein Verweis auf dieses Objekt erzeugt. Änderungen die innerhalb der Methode mit dieser Referenz durchgeführt werden, wirken sich auch auf das ursprüngliche Array aus, da beide Referenzen auf die gleiche Speicherfläche verweisen.
Bei einfachen Datentypen (`int, float`) ist dies nicht der Fall. Änderungen die hier in der Methode durchgeführt werden haben keine Auswirkung auf die ursprünglichen Variable.

Sichtbarkeit

Innerhalb einer Methode verdecken formale Parameter und lokale Variablen gleich lautende *Identifier* aus der umschließenden Klasse. So kann ein *Variablenname* mehrmals innerhalb einer Klasse erscheinen, z.B. als *Klassenvariable*, *lokale Variable* oder als *formaler Parameter*.
Soll innerhalb einer Methode auf eine Variable zugegriffen werden, wird diese erst innerhalb der Methode gesucht (lokale Variable oder formaler Parameter). Ist dort die entsprechende Variable nicht vorhanden wird nach einer entsprechenden *Klassenvariable* gesucht.

Fehlerbehandlung

`RuntimeExceptions` werfen, wenn Fehler auftreten. Beispielsweise beim *Integer-Overflow.*

```
If (Bedingung/en) throw new RuntimeException („Fehlermeldung");
```

Kapitel 5: Rekursion

Behandelte Probleme

Eine Methode darf in der Deklarationen ihres Rumpfes sich selbst aufrufen. Üblicherweise wird sie der aktuelle Parameter dabei so modifiziert, dass die Problemgröße schrumpft.

Begriffe

Rekursion : kurz, braucht aber mehr Speicher; Verankerung, Anweisung mit Selbstaufruf, Bremse, noch nicht abgearbeitete Aufrufe werden dabei in einem *Laufzeitkeller* abgelegt

Laufzeitkeller (Stack): enthält für jeden noch nicht beendeten Aufruf einen Speicherblock (Schachtel, activation record) mit Speicher für Parametervariable und lokale Variable, bei Aufruf wird eine Schachtel gekellert (push), bei Beenden des Aufrufes entkellert (pop).

Details

Rekursion

Wesentlich kompakterer Programme als iterative Version. Sind in der Lösung meist eleganter, Beispiele dafür sind mathematische Probleme wie Fakultät, Potenzieren oder GGT. Der Speicherbedarf der Methode wächst allerdings mit jedem Selbstaufruf und es sollte auf eine geeignete Abbruchbedingung geachtet werden, da es sonst zu einer Endlosschleife kommt (*Halteproblem*).

Wichtige Algorithmen

- **Fibonacci-Zahlen:** 1 1 2 3 5 8 13 21 34 55 ...
- **Türme von Hanoi:** n-1 von Start nach Zwischen, erste Scheibe von Start nach Ziel, n-1 Scheiben von Zwischen nach Ziel

Kapitel 6: Komplexität, Verifikation und Terminierung

Behandelte Probleme

Komplexitätsklassen, Methoden und deren Laufzeiten. Partielle und totale Korrektheit.

Begriffe

Komplexität: Kompliziertheit von Problemen, Ressourcenverbrauch

Lena Kästner
Nico Möller

Laufzeiten (Komplexitätsklassen): Best-case – Average-case – Worst-case

$\log n$	n	$n \cdot \log n$	n^2	n^3	n^4	$\dots$	2^n	3^n	$n!$
↑	↑	↑	↑	↑			↑		↑
binäre Suche	lineare Suche	schlaues Sortieren	dummes Sortieren	Gleichungs-system lösen			alle Teil-mengen		alle Permu-tationen

Obere Schranke für einen Algorithmus: Laufzeit im worst-case

Untere Schranke für ein Problem: Beispiel Sortieren durch Vergleichen. Laufzeit O(n) trivial, da jedes Element einmal betrachtet werden muss, Idealfall bei *Entscheidungssequenz*: O(n*logn)

Verifikation: partielle Korrektheit (Schleifeninvariante) + Terminierung
→ totale Korrektheit + Laufzeitanalyse in O-Notation

O-Notation: Laufzeit eines Programms, nicht absolute CPU-Zeit, sondern Wachstum der Laufzeit in Schritten, in Abhängigkeit der Größe der Eingabe. *Alle Laufzeitangaben* beziehen sich jeweils auf einen *konkreten Algorithmus A* (*für ein Problem P*).

Details

Komplexitätsklassen

Dienen als Aussagen über Laufzeit und Speicherplatzbedarf. Betrachtet werden dabei allerdings nicht absolute Zahlen, sondern das Wachstum der Zeit/ des Platzbedarfs in Abhängigkeit von der Größe der eingegebenen Daten.

Auf die Eingabegröße achten: Bei verschachtelten `for`-Schleifen kann dies einen Unterschied zwischen O (n) und O (n^2) ausmachen.

Bei der Laufzeitanalyse von rekursiven Programmen müssen mehrere Formel für die Laufzeit aufgestellt werden. Zum einen die Formel für die Abbruchbedingung (meistens konstant) zum anderen eine Formel für die übrigen Fälle. Durch eine Grenzwertbestimmung lässt sich dann die Laufzeit des Algorithmus bestimmen.

Korrektheit und Terminierung

Durch Testen kann nachgewiesen werden, dass sich ein Programm für endlich viele Eingaben korrekt verhält. Durch eine Verifikation kann nachgewiesen werden, dass sich ein Programm für alle Eingaben korrekt verhält.

Schleifeninvariante
Dient zur Überprüfung von Schleifen (Abbruchbedingung), nur wenn ein Abbruch gewährt ist kann das Programm terminieren.

Halteproblem

Es gibt prinzipiell kein Programm, welches entscheidet, ob ein gegebenes Programm, angesetzt auf einen gegebenen Input, anhält. → „Klasse *Fee*“

Lena Kästner
Nico Möller

Kapitel 7: Sortieren

Behandelte Probleme

Verschiedene Sortieralgorithmen, *Greedy* und *Divide & Conquer* Strategien, Motivationen fürs Sortieren: - Häufiges Suchen, Suche nach doppelten Elementen.

Begriffe

Greedy-Strategie: verfolgt nächstliegendes Ziel

Divide & Conquer: schnellere Suchstrategie, die auf Teilsuchen setzt

Binärer Baum: ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind. Dieser heißt dann Vater des linken bzw. rechten Teilbaums. Ein Knoten ohne Vater heißt Wurzel. Die Knoten, die x zum Vater haben, sind seine Söhne. Knoten ohne Söhne heißen Blätter.(Ebene 0 = Wurzel. Ebene i + 1 = Söhne von Ebene i; Anz(Blätter) = Anz(Knoten/2), max. Blattzahl bei n Knoten n!, Höhe log n)

Heap: binärer Baum mit n+1 Ebenen, in dem die Ebenen 0, 1, ... , n−1 vollständig besetzt sind; Ebene n ist von links beginnend bis zum so genannten letzten Knoten vollständig besetzt. Die Knoten enthalten Schlüssel. Der Schlüssel eines Knotens ist kleiner oder gleich den Schlüsseln seiner Söhne. Der kleinste Schlüssel eines Heaps steht in der Wurzel.

Entscheidungssequenz: Sortierverfahren, das auf dem Vergleich von immer zwei Elementen beruht

Details

Greedy Strategien

Beispiele sind:

- Selectionsort
- Bubblesort / Shakersort

Divide & Conquer

Beispiele

- Mergesort
- Quicksort

Andere Ansätze

Beispiele

- Heapsort
- Bucketsort

Zusammenfassung Laufzeiten und Platzbedarf

	Best	Average	Worst	Zusätzlicher Platz
Selectionsort	O (n^2)	O (n^2)	O (n^2)	O (1)
Bubblesort	O (n)	O (n^2)	O (n^2)	O (1)
Mergesort	O (n · log n)	O (n · log n)	O (n · log n)	O (n)
Quicksort	O (n · log n)	O (n · log n)	O (n^2)	O (log n)
Heapsort	O (n · log n)	O (n · log n)	O (n · log n)	O (1)

Wichtige Algorithmen

- **`SelectionSort (Array):`** suche kleinste Zahl, vertausche mit erstem Element, beginne ab zweiter Stelle neu → $O(n^2)$, Platzbedarf O(1)=konst.,
- **`BubbleSort (Array):`** läuft von unten los und vertauscht zwei Elemente, wenn das untere kleiner ist, beginnt dann neu → $O(n^2)$, selten O(n) (best); Platzbedarf O(1)=kosnt.
- **`SkakerSort:`** Bubbelsort mit Richtungswechsel nach jedem Tausch
- **`MergeSort (rekursiv):`** zersplitte Array in Einzelelemente (iterativ: sofort Einzelelemente), setze immer zwei zusammen, sodass jeweils das kleinere der aktuelle Elemente an die aktuelle neue Stelle rückt → O(n*logn) = Mischen *Durchläufe, Platzbedarf zusätzlich O(n)
- **`QuickSort (Array):`** springt in die Mitte (Pivotelement x), wandert von oben (j) und unten (i) durchs Array und prüft ob aktuelle Elemente auf der richtigen Seite von x stehen, falls nicht: Tausch, Laufindizes haben sich irgendwann überholt: Quicksort für die Teile (unten, j; oben, i)
  → best, average O(n*logn), worst $O(n^2)$,
  Platzbedarf O(logn)
  → Idealisierung: Median als Pivotelement, dann immer optimale Aufteilung, aber Implementation für realistische Problemgrößen unwirtschaftlich
- **`HeapSort (Array als Baum aufgefasst):`** Einfügen oder Entfernen des letzten Elementes, dann siften (Heap reparieren); Bei Sortierung Wurzelelement ausgeben, letzten Schlüssel in Wurzel kopieren und letztes Blatt löschen, siften: überprüfe jeden Knoten von unten beginnend, ob er größer ist als sein Vater, falls nicht: tauschen, die Blätter der vom Tausch betroffenen Knoten werden erneut überprüft (zusätzliche sift-Aufrufe werden als Vertauschungen gezählt)
  → all cases O(n*logn) Platzbedarf O(1)
- **`BucketSort:`** sortiert Strings, indem es Treffer auf Buckets verteilt → keine Vergleiche, da endlicher, zuvor bekannter Schlüsselbereich
  → Aufwand O(n)+O(N), N = Raum der Schlüssel

Lena Kästner
Nico Möller

Kapitel 8: Objektorientiertes Programmieren

Behandelte Probleme

Sichtbarkeit von Datenfeldern, Eigene Datentypen, Binden, Referenz

Begriffe

Wrapper-Klassen: für primitive Datentypen gibt es in Java entsprechende Klassen

```
Integer x = new Integer (42);
IO.println(x.intValue()+1);)
```

Konstruktor: Instantiierung, Kreieren der Objekte → erzeugt Instanzen der Klasse

Default-Konstruktor: Erzeugt „Schachteln“

Vererbung: neues Objekt = altes Objekt + neue Eigenschaften → stützt sich auf Erweiterung funktionierender Klassen, bestimmt durch *Zugreifbarkeit*

Überladen: mehrere Methoden gleichen Namens, die durch ihre Parameter identifiziert werden (Bsp: *Default-Konstruktor* = leere Parameterliste)

Überschreiben: Methoden, die vererbt werden, können in dem neuem Objekt überschrieben werden. Bei einem Zugriff auf diese Methode wird dann auf die Methode in dem Aktuellen Objekt und nicht auf die der Superklasse zugegriffen.

Dynamisches Binden: welche Methode ausgeführt wird, entscheidet sich erst zur Laufzeit (Programmcode wird Objekt zur Laufzeit zugeordnet – gilt in Java für alle *Instanzmethoden*)

Statisches Binden: Einem Objekt wird zur Kompilezeit der Programmcode zugeordnet (in Java: *Klassenmethoden*, *Klassenvariablen*, *Instanzvariablen*)
→ Auswahl wird fest verdrahtet, Compiler wählt also die Klassenmethode oder Variable aus, die dem Typ der Referenz entspricht

Klassenvariable: Datenfelder die mit dem Schlüsselwort `static` deklariert werden.

Instanz: ein Objekt einer bestimmten Klasse

Referenz: Verweis, Zeiger

Instanzvariable: Datenfeld, welches ohne Schlüsselwort `static` deklariert wird, existiert je Instanz (Objekt) genau einmal und kann entsprechend für jede Instanz einen anderen Wert annehmen, stellt sozusagen eine Eigenschaft eines Objektes dar.

Instanzmethode: kann nicht einzeln aufgerufen werden, an Objekt gebunden, kann innerhalb der Vererbungshierarchie überschrieben werden

lokale Variable: Variable welche innerhalb einer Methode deklariert wird, sie existiert nur zur Zeit des Aufrufs.

Details

Sichtbarkeit von Datenfeldern:

geregelt von *Modifiern*, die bestimmen, wer welche Datenfelder lesen (und ändern) darf

Erreichbar für:	`public`	`protected`	paketsichtbar	`private`
Dieselbe Klasse	ja	ja	ja	ja
andere Klasse im selben Paket	ja	ja	ja	nein
Subklasse in anderem Paket	ja	ja	nein	nein
Keine Subklasse, anderes Paket	ja	nein	nein	nein

ohne Vorgabe: `friendly` (dh. `protected` mit der Einschränkung, dass sie in Unterklassen anderer Pakete unsichtbar sind)

Binden

`this`.thing = thing → ändert Sichtbarkeit; in Konstruktor: übernimmt Parameter für das Objekt, das gerade erzeugt wird; in Methode: bezieht sich auf Parameter des Objektes, an das die Methode geschickt wird. `this`-Referenz löst das Problem, wenn lokale Variablen Objektvariablen verdecken

`super` (Parameterliste) = Schlüsselwort für die Verbindung zur Oberklasse (Student ist Person mit MatNr, Student `extends` Person → neue Methoden könne jetzt die der Oberklasse überschreiben);

Achtung beim Casten: nicht jede Person ist ein Student, beide haben Namen, aber nicht MatNr
→ Fehler vermeiden/ werfen:

```
if (p instanceof Student)
        IO.println (((Student)p).MatNr)
        else throw new PersonException (”…”);
```

zusätzlich wird die Klasse `PersonException` (`extends Exception`) benötigt.

Exceptions

Neben dem auslösen einer `RuntimeException`, gibt es in Java die Möglichkeit eigene Fehlerklassen zu definieren.
`try-catch`: Eine Fehlermeldung in Java ist nichts weiter als ein Objekt, welches an der Stelle erzeugt wird, wo der Fehler aufgetreten ist und von dort aus zum Aufrufer der Methode geworfen wird. Der Programmierer kann sie in einer eigenen Fehlerbehandlung abfangen, ohne, dass das Programm terminiert

Lena Kästner
Nico Möller

Kapitel 9: Abstrakte Datentypen

Behandelte Probleme

Abstrakte Datentypen und deren Erstellung, `Interfaces`

Begriffe

ADT: Datenstruktur zusammen mit darauf definierten Operationen

Interface: enthält nur Methodenköpfe und Konstanten → Schnittstelle, die Funktionen festlegt, ohne Methoden zu implementieren (erfolgt durch beliebige Klasse, die durch `implements` gekennzeichnet ist)

Details

Liste

(ggf. leere) Folge von Elementen zusammen mit so genannten (ggf. undefinierten) aktuellen Element

Schnittstellen:

`empty:` Liste → `Boolean`; Liefert `true`, falls Liste leer ist
`endpos:` Liste → `Boolean`; Liefert `true`, wenn Liste abgearbeitet
`reset:` Liste → Liste; Das erste Listenelement wird zum aktuellen
`advance:` Liste → Liste; Der Nachfolger des aktuellen wird zum aktuellen
`elem:` Liste → Objekt; Liefert das aktuelle Element
`insert:` Liste x Objekt → Liste; Fügt vor das aktuelle Element ein Element ein; das neu eingefügte wird zum aktuellen
`delete:` Liste → Liste; Löscht das aktuelle Element; der Nachfolger wird zum aktuellen
`anf` zeigt auf den ersten Listen-Eintrag (leerer Inhalt)
`pos` zeigt auf den Listen-Eintrag vor dem Listen-Eintrag mit dem aktuellen Element

Implementation
mittels Verweisen (Array macht zu viel Kopieren nötig)

Keller

(ggf. leere) Folge von Elementen zusammen mit einem so genannten (ggf. undefinierten) Top-Element

Schnittstellen: (LIFO-Prinzip)

`empty:` Keller → `boolean`; liefert `true`, falls Keller leer; ansonsten `false`
`push:` Keller x Objekt → Keller; legt Element auf Keller
`top:` Keller → Objekt; liefert oberstes Element
`pop:` Keller → Objekt; entfernt oberstes Element

Semantik:

1. Ein neu konstruierter Keller ist leer.
2. Nach einer `push`-Operation ist ein Keller nicht leer.
3. Nach einer `push`-`pop`-Operation ist der Keller unverändert.
4. Nach der `push`-Operation mit dem Element x liefert die `top`-Operation das Element x.

Implementation mittels Verweisen:
Boxing: Primitive Datentypen müssen zunächst in Wrapper-Klassen verpackt werden!
Vorteil: kein ständiges Umkopieren, Array auch möglich (schneller Zugriff auf bestimmte Position)

Beispiele: Klammerung überprüfen, Infix- in Postfixnotation umwandeln, Quicksort iterativ

Lena Kästner
Nico Möller

Schlange

(ggf. leere) Folge von Elementen zusammen mit einem (ggf. undefinierten) Front-Element (Front-Element = aktuelles Element)

Schnittstellen: (FIFO-Prinzip)

`enq:` Schlange x Objekt → Schlange; Fügt Element hinten an
`deq:` Schlange → Schlange; Entfernt vorderstes Element
`front:` Schlange → Objekt; Liefert vorderstes Element
`empty:` Schlange → `boolean`; liefert `true`, falls Schlange leer ist, `false` sonst

Implementaion

mittels Array: nur endliche Länge, Platzersparnis (falls Array a [N] gefüllt bis a [N-1] füge folgende Elemente ab a [0] ein ← Zählvariablen `head` und `count`)

Baum (binär)

ist entweder leer oder besteht aus einem Knoten, dem zwei binäre Bäume zugeordnet sind.

Schnittstellen:

`empty:` Baum → `boolean`; `true`, falls Baum leer
`value:` Baum → Objekt (Wurzelelement)
`left:` Baum → Baum; linken Teilbaum
`right:` Baum → Baum; rechten Teilbaum

Implementaion

mittels Verweisen: Baum besteht aus Menge von Knoten; VerweisBaum enthält Verweis auf Wurzelknoten oder null-Verweis; jeder Knoten enthält Verweise auf Inhalt, linken und rechten Sohn (sofern vorhanden)

Traversieren (durch Rekursion)

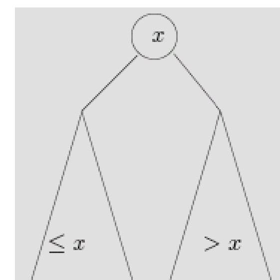
`preorder`: Vater – links – rechts
`inorder`: links – Vater – rechts
`postorder`: links – rechts – Vater

`Tiefensuche`: iterativ; VerweisKeller, der Objekte des Typs Baum enthält
→ Baum in `preorder`-Reihenfolge betrachten

`Breitensuche`: iterativ; Schlange, die Teilbäume enthält → Baum ebenenweise betrachten
Konstruktion einen binären Baums aus Operanden und Operatoren (Eingabe in `Postfix`-Reihenfolge): Elemente werden auf Keller gestapelt, falls Operator `top`- Element: verknüpfe ihn mit zwei darunter liegenden Objekten zu Baum, lege auf Keller

Interface für Mengen

```
public interface Menge{
   public Comparble lookup (Comparable x);
   // Rückgabe des Verweises auf das gesuchte Objekt
      (oder null)
   public boolean insert (Comparable x);
   public boolean delete (Comparable x);
   // melden, ob Operation gelungen
}
```



→ Suchbaumoperationen sind typisch für Mengen
Elemente müssen vergleichbar sein (`s.insert(apfel));` ~~`(s.insert(birne))`~~

Suchbaum (binär)

binärer Baum, bei dem alle Einträge im linken Teilbaum eines Knotens `x` kleiner sind als der Eintrag im Knoten `x` und bei dem alle Einträge im rechten
Teilbaum eines Knotens `x` größer sind als der Eintrag im Knoten `x`.

Operationen:
`lookup (x)`: stiege in Wurzel ein und gehe tiefer in zuständige Richtung
`insert (x)`: steige bei Wurzel ein, gehe tiefer bis zu der Stelle, an der x vermutet wird; füge `x` ein, falls nicht schon vorhanden (Menge!)
`delete (x)`: Blätter → `null`-Verweis; Knoten mit Sohn: Vater → `null`-Verweis, Großvater verweist auf Sohn; Knoten mit zwei Söhnen: wird ersetzt durch größten Knoten des Teilbaumes (dieser wird dann entsprechend gelöscht)
→ Aufwand im O-Kalkül:
günstige Struktur (lookup & insert): ausgeglichener, blattreicher Baum
→ best $O(2^{\log(n)}-1)$, average $O(\log(n))$
ungünstige Struktur (lookup & insert): unausgeglichener, zu Liste entartet
→ worst $O(n)$

Implementaion
Suchbaum `extends` Versweisbaum

Multimenge
lässt doppelte Elemente zu (entweder doppelt einfügen, oder Zähler im Knoten mitführen)

AVL-Baum

binärer Baum, der in jedem Knoten ausgeglichen ist → günstiger Suchaufwand, Höhe bei n Knoten ≤ 1,45 $\log_2(n)$

Balance
Ein Knoten eines binären Baumes heißt ausgeglichen oder balanciert, wenn die Höhen deiner beiden Söhne um höchstens 1 unterscheiden
Balance eines Knotens = Höhe des rechten Teilbaums – Höhe des linken Teilbaums (Zählung beginnt von unten; bei |Balance|>1 erfolgt Reparatur)
Ausgeglichenheit (wieder)herstellen: Reparatur mittels Rotation
nach Einfügen eines Schlüssels: höchstens eine Rotation; nach Löschen eines Schlüssels: kann Rotation für jeden Knoten bis zur Wurzel erfordern

Mögliche Rotationen
`LL-, LR-` (linksseitiges Übergewicht), `RL-, RR-` Rotationen (rechtsseitiges Übergewicht)
Durchgeführt wird Bei Übergewicht mindestens eine der Rotationen in bestimmten Fällen auch zwei.

Zwei Beispiele für linksseitiges Übergewicht
`single LL:`
→vom Problemknoten eine Ebene nach unten
→ dieser Knoten wird an die Stelle des Problemknoten „verlinkt", kleinerer Teilbaum wird zum Sohn des Problemknotens)
`double LR:`
→ vom Problemknoten zwei Ebenen nach unten
→ dieser Knoten wird an die Stelle des Problemknoten „verlinkt", übrige Verweise werden umgebogen)

Fibonacci-Bäume
minimal ausgeglichene Bäume, 45% länger als komplett ausgeglichene Bäume

Lena Kästner
Nico Möller

Mehrwege-Baum

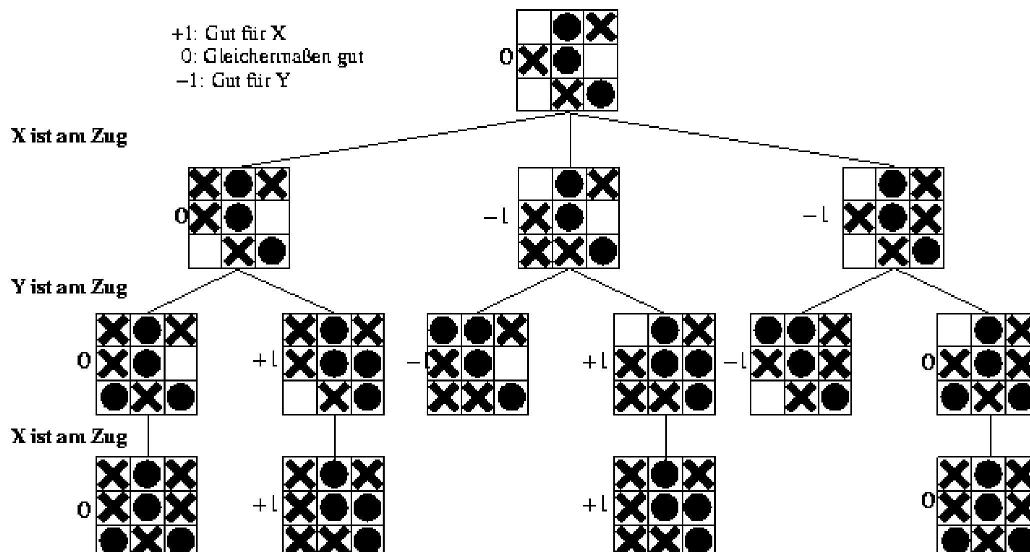
variable Anzahl von Söhnen

Implementation
jeder Knoten enthält Verweis auf seinen ältesten Sohn und seinen nächstjüngeren Bruder

Methoden
`empty, first, next, value`

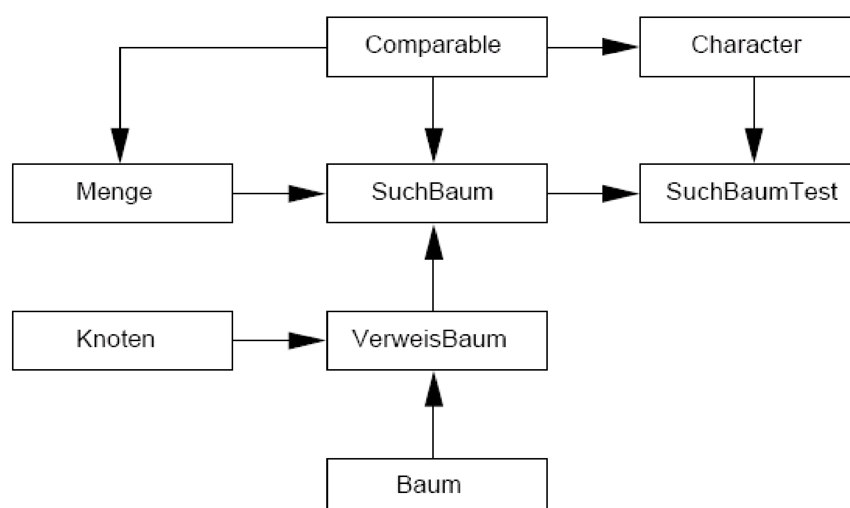
Spielbaum
Mehrwege-Baum mit `min`-Knoten (Wert = min. seiner Söhne) und `max`-Knoten (Wert = max. seiner Söhne), in dem die Knoten Spielstellungen, die Kanten Spielzüge repräsentieren, Wert eines Blattes bestimmt durch Stellungsbewertung (statisch)
→ viele Alternativen, ökonomisch: dynamische Erzeugung während des Spielverlaufs
→ Baum nie vollständig vorhanden

Anz. Blätter = α^{h-1} α ist Verzweigungsgrad, h ist Höhe des Baumes
Beispiel: 2 Spieler: `min`- und `max`-Knoten wechseln sich ab



Zusatz Kapitel 9:

Abhängigkeiten



Java Collection Framework

Collection (interface)

Interface, das eine Sammlung von (nicht notwendigerweise verschiedenen) Objektenverwaltet; Test auf inhaltliche Gleichheit: `Object`-Methode `equals` (geeignet vom Anwender überschrieben)
Folgende Methoden:

```
boolean add(Object o);       // fuegt Objekt o hinzu
boolean contains(Object o);  // testet, ob es Objekt x gibt mit x.equals(o)
boolean remove(Object o);    // entfernt ein Objekt x mit x.equals(o)
boolean isEmpty();           // testet, ob Collection leer
Iterator iterator();         // liefert einen Iterator
```

Iterator (object)

Objekt, welches das Durchlaufen aller Objekte einer `Collection` erlaubt
Methoden des `Iterator`:

```
boolean hasNext();      // liefert true, falls noch Objekte da
Object next();          // liefert naechstes Objekt
void remove();          // entfernt das Objekt, welches als letztes mit
                        // next() geliefert wurde
```

Set (interface)

Von `Collection` abgeleitet, verwaltet Menge ohne Duplikate, verfügt nur über geerbte Methoden.

List (interface)

von `Collection` abgeleitet, verwaltet sequentiell geordnete Sammlung von Objekten, verfügt über zusätzliche Methoden:

```
boolean add(int i, Object o);     // fuegt Objekt o bei Position i ein
Object get(int i );               // liefert Objekt an Position i
boolean int indexOf(Object o);    // liefert Position von Objekt o
```

LinkedList (interface)

von `List` abgeleitet, doppelt verzeigert; verfügt zusätzlich über **`ListIterator`** - kann Liste auch rückwärts durchlaufen, Methoden:

```
boolean hasPrevious()
Object previous()
```

SortedSet (interface)

von `Set` abgeleitet, verwaltete Menge muss geordnet sein (Objekte müssen `Comparable` implementieren oder durch die `compare`-Methode eines `Comparator`-Objektes vergleichbar sein)

Comparable (interface)

vergleichbare Objekte müssen dieses Interface implementieren
liefert = 0 falls this = x
liefert < 0 falls this < x
liefert > 0 falls this > x

Beispiel:

```
public class StudentComparable extends Student implements Comparable{
   int CompareTo (Student s){
   }
}
```

Aufruf: `if (c.CompareTo (d) = 0)…`

TreeSet `(interface)`

von `SortedSet` abgeleitet
Befehl:

```
Set <Character> = new TreeSet <Character> ();
```

instantiiert `TreeSet`, das ausschließlich Objekte vom Typ `Character` aufnimmt (generische Klasse), eine solche Instantiierung wird als *typsicher* bezeichnet.

Enumeration `(Interface)`

Beispiel Traversierung von Bäumen.
Methoden:

```
boolean hasMoreElements()

Object nextElement()
```

Statische Sortiermethoden für Java Collection Framework

Methode aus der Klasse `java.util.collections`.
`sort(List l)`
Objekte der Liste müssen `Comparable`-Interface implementieren
`sort(List l, Comparator c)`
Der Anwender muss das Interface `Comparator` durch Bereitstellung der Methode
`int compare (Object o1, Object o2)` implementieren

Kapitel 10: Hashing

Details

Problem:	einem zu speichernden Element möglichst zielsicher eine Adresse zuordnen.
Lösungsversuch:	Hashfunktion (gute Funktionen führen zu wenigen Kollisionen)
Kollision:	$f(u) = f(v) bei\, v\, != u \rightarrow$ unterschiedliche Behandlung (Fall: $y = f(x)$ sei schon belegt)

offenes Hashing
Listenbildung bei Kollision (Knubbel)

geschlossenes Hashing

a. lineares Sondieren: probiere `y, y+1, y+2, …`
b. quadratisches Sondieren: probiere `y, y+1, y+4, y+9, …`
c. double Hashing: probiere `y+g(x), y+2*g(x), …`

→ Problem: durch Löschen früh eingefügter Objekte, sind neuere mit gleichem Hashwert nicht mehr auffindbar

Lösung: Hilfsarray, das speichert, ob der Platz schon einmal besetzt war

Laufzeitanalyse für geschlossenes Hashing

N = Anz. möglicher Positionen (sog. Buckets)
n = Anz. aktuell gespeicherter Objekte
α = n/N ≤ 1 = Auslastungsfaktor

	AVL-Baum	Geschlossenes Hashing
Laufzeit	logarithmisch	konstant
Speicherbedarf	dynamisch wachsend	in Sprüngen wachsend
Sortierung	möglich durch Traversierung	nicht möglich

Anz. der Schritte mit `double Hashing` bei erfolgloser Suche: ≈ 1/(1- α) = 5.0 für α = 0.8
Anz. der Schritte mit `double Hashing` bei erfolgreicher Suche: ≈ -(ln(1-α))/α = 2.01 für α = 0.8

Hashing in Java

`HashSet` von `Set` abgeleitet
bereits bekannt Methoden `add`, `contains` und `remove` werden über eine Hashorganisation gelöst.
Der Konstruktor erlaubt (optional) die Angabe einer initialen Kapazität

`HashMap` von `Map` abgeleitet
→ anders als Collection! - Assoziation von Schlüsseln und Werten (→ getrennte Speicherung)
Methoden:

```
void put (int, Object)
boolean containsKey(int)
boolean containsValue(Object)
```

Kapitel 11: Graphen

Behandelte Probleme

Graphen, verschiedene Möglichkeiten der Implementation von Graphen in Java, Typische Probleme die durch Graphen dargestellt und gelöst werden können.

Begriffe

Graph: Ein (ungerichteter) Graph G besteht aus einer Knotenmenge V und einer Kantenmenge E, Wobei E alle zweielementigen Teilmengen von V enthält.

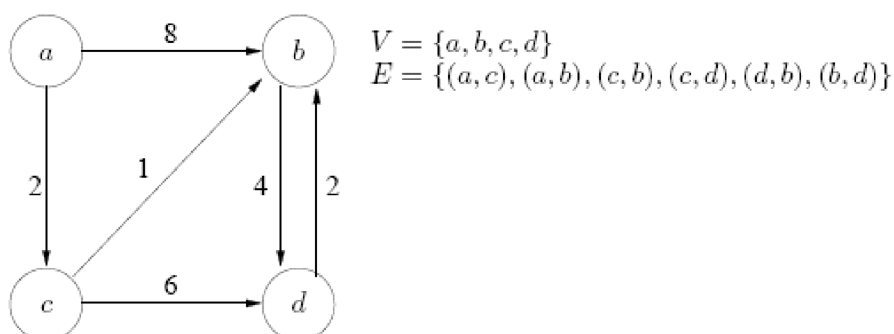
gerichteter Graph: Ein gerichteter Graph besitzt Kanten die nur in bestimmte Richtungen „begehbar" sind.

gewichteter Graph: Ein gewichteter Graph besitzt Kanten die mit einer bestimmten Kostenfunktion ($c: E \rightarrow R$) belegt sind.

Adjazenz: Zwei Knoten heißen in einem *ungerichteten* Graph *adjazent* oder benachbart, wenn sie in diesem durch eine Kante verbunden sind. Zwei Kanten heißen *adjazent* oder benachbart, wenn sie sich an einem Knoten berühren, das heißt diesen gemeinsam besitzen.

Details

Bild eines gerichteten und gewichteten Graphen



Implementationsmöglichkeiten

Implementation durch Adjazenzmatrix

n-Elemente → $n \times n$ - Matrix (Kanten werden in Felder für Wertepaare der Knoten eingetragen)

aus Spalte: von
aus Zeile: nach

für Graph ohne Gewichtung: nur 0,1, wobei 0 für vorhandene Kante steht

für Graph mit Gewichtung: Wert der Kostenfunktion, falls Kante existiert, 0, falls gleiches Element, ∞ falls keine Kante existiert

Vorteile: effizienter Test, ob Kante existiert; Bestimmung der Nachbarn in O(n)

Platzbedarf: $O(V^2)$ → nur günstig, falls G dicht besetzt

Implementation durch Adjazenzliste:

zu Knoten werden Listen ihrer Nachbarn geführt

Vorteil: effizienter Nachbarn durchlaufen

Nachteil: Test, ob Kante existiert dauert länger

Platzbedarf: $O(|E|)$ → proportional zur Kantenanzahl
→ günstig nur, falls Graph dünn besetzt

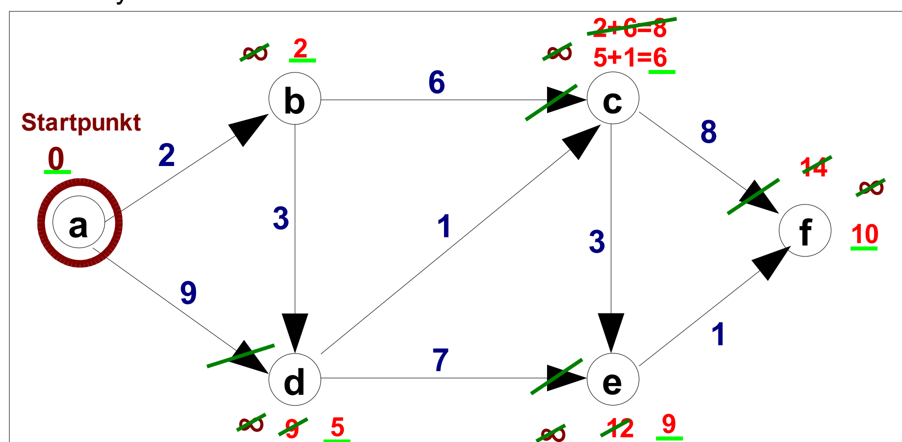
Typische Probleme

Traversieren jeden Knoten genau einmal besuchen

all-pairs-shortest-path-Problem günstigster Weg für jede Start-Ziel-Variante → $n \times n$ Matrix
→ in Polynomzeit lösbar

single-source-shortest-path-Problem

von einem bestimmten Knoten → kürzester Weg zu allen anderen Knoten
→ in Polynomzeit lösbar



Vorläufige Bewertung

als fest einstufen, d.h.
Es gibt keinen günstigeren Weg

günstigster unter den Vorläufigen

Verursacher= Vorgänger

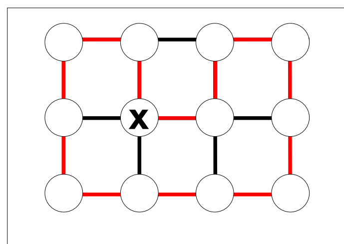
Weg von a nach f: bestens 10

Sequenz: a → b → d → c → e → f

Laufzeit: $O(|E|\log(|V|))$ auf Grund von *Heapsortierungen* (Anwendung)

Hamilton-Kreis

jeden Ort besuchen, Rückkehr an Ausgangsort



Laufzeit: $O(2^n)$

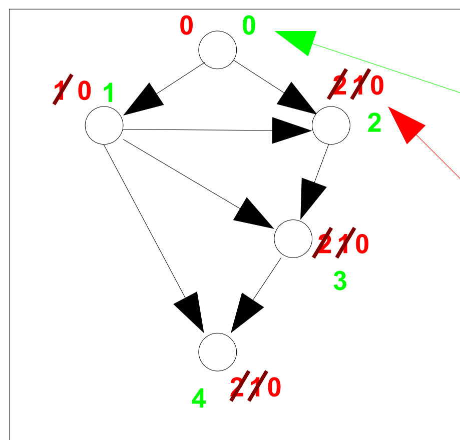
Travelling-Salesman

günstigster Hamilton-Kreis

topologisches Sortieren

Nummerierung der Knoten, so dass jeder Knoten ($\in V$ *eine* Zahl $\in \{0,\dots,n-1\}$ erhält

→ Suche $f : V \to \{0,\dots,n-1\}$ mit $f(x) < f(y)$ falls $(x,y) \in E$



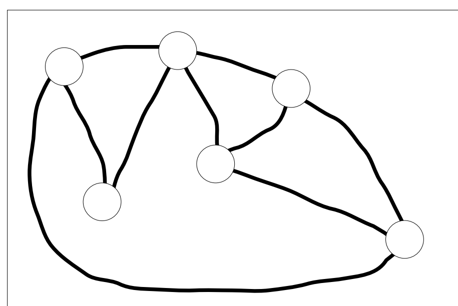
Nächste Nummer nur für Knoten mit logischer Eingangsmenge '0'

setzt logischen Eingangsgrad der Nachfolger um eins herab

Laufzeit: $O(|E|)$
→ jeden Knoten 1x ansehen → *konstant*

Chinese Postman

jeden Weg (Kante) einmal ablaufen (vgl. Königsbergproblem)



Laufzeit: $O(n^k)$