Grundlagen der Programmierung (1)

Konventionen für Java-Quelltextdateien und Dateinamen

- Groß- und Kleinschreibung sind relevant
- Anweisungen werden mit Semikolon ; abgeschlossen
- Dateinamen von Java-Quelltextdateien ergeben sich aus dem Klassennamen, der als Wort hinter dem Schlüsselwort class steht, ergänzt um die Endung .java
- Bytecode-Dateinamen ergeben sich aus dem Namen der Java-Quelltextdatei gemäß der Konvention, dass die Endung .java durch .class ersetzt wird

Variable

- bezeichnet Speichereinheit/-stelle zur Aufnahme von Datenwerten (z. B. (Zwischen-) Ergebnisse einer Berechnung)
- hat Namen und Datentyp, der angibt, welcher Typ von Werten in ihr gespeichert werden kann
- muss vor ihrer ersten Verwendung deklariert (auch: vereinbart) worden sein
- Bei der Deklaration ist der Typ der Daten anzugeben, der in der Variable zu speichern ist.
- Da intern jede Information (z. B. Zahl, Zeichen) als Bitstring (aus 0 und 1) repräsentiert wird (wie, folgt gleich), dient Datentyp dazu, Speicherinhalt korrekt zu interpretieren

Benennung von Variablen

- Konventionen/Programmierstil:
  - Variablennamen sollten mit Kleinbuchstaben beginnen.
  - Von der Verwendung des $-Zeichens wird abgeraten.
  - Namen sollten selbsterklärend sein (sog. mnemonische Namen) und die Lesbarkeit eines Programms fördern
  - Bei zusammengesetzten Substantiven sollte jedes neue Wort aus Gründen der Lesbarkeit mit einem Großbuchstaben beginnen.

Wertzuweisung an eine Variable

- Wertzuweisung ist eine Operation, die dazu dient, Wert in Speichereinheit zu schreiben, die durch Variable bezeichnet ist (schreibender Zugriff)
  - Sie hat die Form: <Name> = <Ausdruck>;
- auch möglich (und auch sinnvoll), einer Variablen bereits bei ihrer Deklaration einen Wert zuzuweisen
  - Form: <Datentyp> <Name> = <Ausdruck>;
- Variablen können erst dann in einem Programm verwendet werden, wenn sie ZUVOR deklariert wurden

(Symbolische) Konstante

- Wert, der sich zur Laufzeit eines Programms nicht ändern kann
- verwendbar wie Variablen, aber nur lesender Zugriff
- zwei Arten von Konstanten
  - statische (globale) Konstanten, überall im Programm verfügbar
  - (lokale) Konstanten, nur in bestimmtem Programmteil verfügbar (folgt später)
- Form: final int MAXKUNDENNR = 1_00_000;
- Zweck
  - häufig verwendete, konstante Werte in einem Programm als benannte Konstante deklarieren
  - wenn Wert geändert werden muss, reicht es aus, die Änderung einmal bei der Deklaration vorzunehmen und nicht an allen Stellen, an denen die Konstante verwendet wird
  - erleichtert die Änderbarkeit und Wartbarkeit eines Programms
- Konventionen
  - Konstanten werden am Anfang eines Blocks deklariert.
  - Bezeichner von Konstanten werden in GROSSBUCHSTABEN geschrieben.
  - Bei zusammengesetzten Bezeichnern Unterstrich _ zur Worttrennung verwenden.

Zahlensysteme

Dezimal	Dual	Sedezimal	Dezimal	Dual	Sedezimal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Verschiedene Zahlensysteme

- Dezimalsystem ≡ Zahlensystem zur Basis 10
  - ri ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- Binärsystem ≡ Zahlensystem zur Basis 2
  - ri ∈ {0, 1}
- Sedezimalsystem ≡ Zahlensystem zur Basis 16
  - ri ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}
- Oktalsystem ≡ Zahlensystem zur Basis 8
  - ri ∈ {0, 1, 2, 3, 4, 5, 6, 7}

Umrechnung: beliebiges System→ Dezimalsystem

$$S = \sum_{I=0}^{I-1} (r_i * b^i)$$

Beispiel (von Dezimal auf Dezimal) : $1303_{(10)} =^{(10)} (1 * 10^3) + (3 * 10^2) + (0 * 10^1) + (3 * 10^0) = 1303$

Beispiel (von Oktal auf Dezimal) : $1303_{(8)} =^{(10)} (1 * 8^3) + (3 * 8^2) + (0 * 8^1) + (3 * 8^0) = 707$

Umrechnung: Dezimalsystem 7→ beliebiges System

Divisionsmethode/Modulo-Methode

- Zahl N aus dem Dezimalsystem in System zur Basis b umwandeln:
- N ganzzahlig durch b teilen ; → ganzzahliger Quotient N' , Rest R
- R als Ziffer „vorne an das Ergebnis anhängen"
- falls N'≠ 0: mit N' nach gleichem Schema verfahren

N		b	Quotient N'	Rest R
4867	/	16	304	3
304	/	16	19	0
19	/	16	1	3
1	/	16	0	1

$\rightsquigarrow 4867_{(10)} = 1303_{(16)}$

Datentyp

- Datentyp: Menge von Daten gleicher Art
- Beispiele für direkt verfügbare Datentypen in Java:
  - ganze Zahlen: byte, short, int, long
  - Fließkommazahlen: float, double
  - Zeichen: char
  - boolescher Wert: boolean
- Diese Datentypen werden als primitive Datentypen bezeichnet.
- Daneben: direkt verfügbare, zusammengesetzte Datentypen, z. B. Arrays und Zeichenketten
- Auch möglich: selbst neue Datentypen deklarieren, Details später in der Lehreinheit zur objektorientierten Modellierung und Programmierung.

Bit und Bitfolgen

- Bit
  - kleinstmögliche Einheit der Information
  - Informationsmenge in einer Antwort auf eine Frage, welche zwei Möglichkeiten zulässt, Man benutzt dazu die Zeichen 0 und 1.
  - Bit: Maßeinheit, bit: Einheit(szeichen) in Größenangaben
- Bitfolgen
  - erforderlich, wenn mehr als zwei Alternativen, z. B. 00: Süd, 01: West, 10: Nord, 11: Ost
  - Es gibt genau 2^N mögliche Bitfolgen der Länge N.
  - Spezialfall N = 8: 1 Byte = 8 Bits mit Einheitszeichen B

Möglichkeiten zur Darstellung negativer Zahlen

- Vorzeichen-Betrags-Darstellung
  - Das vorderste Bit gibt das Vorzeichen an, wobei 0 positive Zahlen und 1 negative Zahlen kennzeichnet. Die restlichen Bits stellen den Betrag der Zahl dar.
  - $+13_{(10)} \rightarrow 00001101_{(2)}$
  - $-13_{(10)} \rightarrow 10001101_{(2)}$
  - Probleme : Zwei Darstellungen der Null (±0), positive und negative Zahlen müssen bei Rechnungen getrennt behandelt werden.
- Einerkomplement-Darstellung
  - Positive Zahlen werden wie vorhin gezeigt dargestellt (auf führende Null achten!).
  - Für negative Zahlen werden alle Bits der dazugehörigen positiven Zahl invertiert.
  - $+13_{(10)} \rightarrow 00001101_{(2)}$
  - $-13_{(10)} \rightarrow 11110010_{(2)}$
  - Probleme: Zwei Darstellungen der Null (±0).

- Zweierkomplement-Darstellung
  - Positive Zahlen werden wie vorhin gezeigt dargestellt (auf führende Null achten!).
  - Für negative Zahlen werden alle Bits der dazugehörigen positiven Zahl invertiert, anschließend wird 1 addiert.
  - $+13_{(10)} \rightarrow 00001101_{(2)}$
  - $-13_{(10)} \rightarrow 11110010_{(2)}$
  - Vorteile Nur noch eine Darstellung der Null, positive und negative Zahlen können bei Rechnungen gleich behandelt werden.

Entscheidungsgehalt

Der Entscheidungsgehalt H einer Menge entspricht der Anzahl an Bit, die man benötigen würde, um jedes Element dieser Menge eindeutig mit einem Bitmuster zu identifizieren.

Berechnung H(M) = $\lceil \log_2(|M|) \rceil$

Ganze Zahlen

Datentyp	Beschreibung	Wertebereich
byte	vorzeichenbehaftete Ganzzahlen von 8 bit Länge	-128 ≤ byte ≤ 127
short	vorzeichenbehaftete Ganzzahlen von 16 bit Länge	-32 768 ≤ short ≤ 32 767
int	vorzeichenbehaftete Ganzzahlen von 32 bit Länge	-2 147 483 648 ≤ int int ≤ 2 147 483 647
long	vorzeichenbehaftete Ganzzahlen von 64 bit Länge	-9 223 372 036 854 775 808 ≤ long long ≤ 9 223 372 036 854 775 807

Aufzählungstypen

- sind gedacht für int-Variablen, die nicht jeden beliebigen Wert annehmen dürfen, sondern auf eine begrenzte Anzahl von Werten beschränkt sind
- Diese Werte werden über Namen angesprochen.
- Typische Kandidaten für solche Aufzählungen sind die Tage einer Woche, die Himmelsrichtungen, die Monate eines Jahres oder die Einheiten einer Währung.

```
enum Orientation {
    NORTH, SOUTH, WEST, EAST
}
```

Gleitkommazahlen (auch: Fließkommazahlen)

- Ziel: approximative Darstellung einer reellen Zahl
- Festkommazahl: begrenzte Ziffernfolge der Länge n, Komma an festgelegter Stelle und damit $1 \leq k < n$ Vorkommastellen und n - k Nachkommastellen. Gibt es in Java nicht.
- Gleitkommazahl (Fließkommazahl, engl. floating point number): Darstellung der Zahl x mit zwei Werten, der Mantisse m (mit $1 \leq m < 10$) und dem Exponenten e: $x = \mp m \cdot 10^e$
  - float Gleitpunktzahlen von 32 bit gemäß IEEE 754-1985
  - double Gleitpunktzahlen von 64 bit gemäß IEEE 754-1985

Zeichen

- Rechner werden auch zur Textverarbeitung eingesetzt, Programme müssen Textausgaben für Interaktion mit Benutzer bereitstellen.
- Programmiersprachen stellen daher auch Datentypen zur Speicherung und Verarbeitung v. Zeichen zur Verfügung.
- Datentyp char repräsentiert Menge der Zeichen.
- Literale: Zeichenkonstante wird durch Angabe des Zeichens in Hochkommata repräsentiert.

Binärdarstellung von Zeichen: Erweiterter ASCII-Code (8 Bits)

Binärdarstellung von Zeichen: Unicode

- Problem: Vielfalt der ASCII-Erweiterungen
- Ziel: Zusammenfassen sämtlicher relevanter Zeichen verschiedener Kulturkreise in einem universellen Code
- sog. Basic multi-lingual plane of Unicode, 16 Bit-Codierung, enthält Platz für 65536 Zeichen
- char kann ein solches 16-Bit-Zeichen aufnehmen; für Zeichen anderer „planes“ werden zwei char-Werte benötigt.

Wahrheitswerte

- Bedingung in Alternative oder Schleife kann wahr oder falsch sein, „wahr“ und „falsch“ sind sogenannte Wahrheitswerte.
- Literale: Wahrheitswerte true („wahr“, 1) und false („falsch“, 0)
- Java: Datentyp boolean

Array

- = zusammengesetzter Datentyp
- = endliche Folge von Werten dieses Datentyps
- Beachte: Elemente sind beginnend ab 0 durchnummeriert.
- Form: int [] a = {15,12,13}

Erzeugung von Arrays

- Die Größe wird bei Erzeugung des Arrays festgelegt und kann dann nicht mehr geändert werden.
- Variante 1: Deklaration und erste Wertzuweisung
  - int [] a = {15,12,13}
- Variante 2: Deklaration und Erzeugung eines Arrays
  - Int a [] = new int[10]; (Array länge 10)
- Variante 3: Deklaration einer Array-Variablen, dann Erzeugung
  - int[] a;
  - a = new int[10]; (Array länge 10)
- nicht initialisierte Arrays werden mit Standardwerten belegt:
- 0 bei byte, short, int, long, float, double false bei boolean; null bei Referenzen

Arbeiten mit Arrays

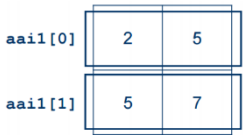
- lesender Zugriff: int y = a[1] + a[4]; // y = 45
- schreibender Zugriff: a[2] = 3; a[4] = a[5] / 2 + 1;

Mehrdimensionale Arrays

- Der für die Array-Elemente verwendete Typ kann beliebig sein (primitiver Typ, zusammengesetzter Typ (z. B. wiederum ein ArrayTyp)).
- Daher sind mehrdimensionale Arrays (Array von Arrays) möglich.
- Deklarationsbeispiel: int[] [] aa;

„Rechteckige“ mehrdimensionale Arrays

- gut geeignet, um z. B. feste tabellarische Strukturen von Daten zu verwalten
- Beispiel: int[][] aai1 = {{2, 5}, {5, 7}};



Zeichenfolgen

- Zeichenfolge (auch: String) setzt sich aus Folge von Zeichen zusammen, also z. B. "Peter Mueller", "a+b", "abcd"
- Menge aller Zeichenfolgen definiert Datentyp, der in Java (und anderen Sprachen) String genannt wird.
- Literale: Zeichenfolge wird in Java als Folge der Zeichen in Anführungszeichen geschrieben.
- Deklarationsbeispiel: String name = "Peter Mueller";

Operatoren und Ausdrücke

- Ausdrücke sind Formeln zum Berechnen von Werten, deren Operanden mittels Operatoren verknüpft werden.
- Unterscheidung zwischen verschiedenen Typen von Ausdrücken, abhängig davon, welchen Wert sie am Ende liefern, z. B. arithmetische Ausdrücke, boolesche Ausdrücke.

Operator	Ausdruck	Beschreibung
+	x + y	Addition
-	x – y	Subtraktion
*	x * y	Multiplikation
/	x / y	Division
%	x % y	Modulo, Restwert

Operator	Ausdruck	liefert `true`, wenn
>	x > y	x größer als y
>=	x >= y	x größer oder gleich y
<	x < y	x kleiner als y
<=	x <= y	x kleiner oder gleich y
==	x == y	x gleich y
!=	x != y	x ungleich y

(Post-/Pre-) Inkrementierung (ein Operand◊ unärer Operator)

- a++: Inkrementiere a um 1 (nach der Auswertung von a)
- int a = 0, b; // a hat den Wert 0, b nicht initial.
- b = a++; // b hat den Wert 0, a den Wert 1
- ++a: Inkrementiere a um 1 (vor der Auswertung von a)
- int a = 0, b; // a hat den Wert 0, b nicht initial.
- b = ++a; // b hat den Wert 1, a den Wert 1

Arithmetische Ausdrücke

- Arithmetische Ausdrücke ähneln Formeln aus Zahlenwerten in der Mathematik, die u. a. die üblichen Operatoren (+, -, *, /) verwenden und geklammert sein können
- Auswertung eines Ausdrucks erfolgt „von links nach rechts“ unter Berücksichtigung der Regeln „Klammern zuerst“, „Punktrechnung vor Strichrechnung“ und sog. Bindungsregeln (Details folgen).
- möglich: Verknüpfung von Wertzuweisung und Ausdruck

Boolesche Ausdrücke

- Boolesche Ausdrücke sind Formeln, in denen Operanden durch boolesche Operatoren verknüpft werden, und die als Ergebnis einen Wahrheitswert „richtig“ (true) oder „falsch“ (false) liefern.
- Operanden können sein:
  - die Wahrheitswerte true oder false,
  - Vergleiche zwischen arithmetischen Ausdrücken oder
  - boolesche Variablen
- Logische Verknüpfungen erfolgen mit booleschen Operatoren.
- Die bekanntesten booleschen Operatoren sind ! (nicht), && (und), || (oder)

x	!x
true	false
false	true

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

x	y	x||y
false	false	false
false	true	true
true	false	true
true	true	true

Weitere Grundoperationen für primitive Typen

- logische Bitoperatoren (Hinweis: &, |, ^ auch für boolean)
  - & (bitweises Und): 0101 & 1011 => 0001 („nur wenn beide 1“)
  - | (bitweises Oder): 0101 | 1011 => 1111 („wenn min. eine Zahl 1 ist“)
  - ^ (XOR, bitweises Entweder-Oder): 0101 ^ 1011 => 1110 („nur wenn beide Zahlen unterschiedlich“, „die eine oder die andere Stelle 1 ist“)
  - ~ (bitweises Komplement): 0110 => 1001 (bitweise Invertierung)
- Bit-Verschiebung
  - << (Verschiebung nach links)
    - 15 << 3: (15 =) 00001111 wird zu 01111000 (= 120)
  - >> (Verschiebung nach rechts; beachtet Vorzeichen, von links altes Vorzeichen-Bit einschieben)
    - 15 >> 3: (15 =) 00001111 wird zu 00000001 (= 1)
  - >>> (Verschiebung nach rechts ohne Berücksichtigung der Vorzeichen, ignoriert Vorzeichen, von links neue Nullen einschieben)
    - -1 >>> 3: (-1 =) 11111111 wird zu 00011111 (= 31)

Kurzschreibweise: Grundoperatoren kombiniert mit Zuweisung

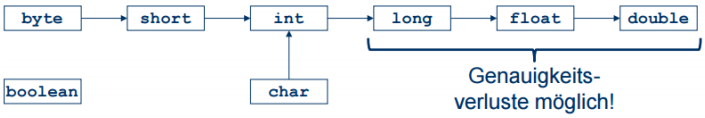
- **=** Zuweisung
- += Inkrementierung, z. B. alter+=80 (in etwa alter=alter+80)
- -= Dekrementierung, z. B. alter-=80 (in etwa alter=alter-80)
- *= Skalierung, z. B. alter*=2 (in etwa alter=alter*2)
- /= Division, z. B. alter/=2 (in etwas alter=alter/2)
- %= Restebildung, z. B. alter%=10 (in etwa alter=alter%10)
- |= bitweises Oder
- &= bitweises Und
- ^= bitweises Entweder-Oder (Exklusives Oder)
- <<= Verschieben nach links, z. B. alter<<=2 (in etwa alter=alter<>= Verschieben nach rechts (mit Vorzeichenerhalt), z. B. alter>>=2 (in etwa alter=alter>>2)
- >>>= Verschieben nach rechts (ohne Vorzeichenerhalt) z. B. alter>>>=2 (in etwa alter=alter>>>2)

Operatoren für Zeichenketten

- Konkatenation von Zeichenketten und Vergleich:
  - boolean istGleich; istGleich = "hello" + "students" == "hellostudents"; // istGleich: true
- Länge der Zeichenkette: .length();

Typkonvertierung (engl. type casting)

- In Mathematik erlaubt: 10 + 0.9, da ganze Zahlen Teilmenge der reellen Zahlen.
- In Programmen möglich: Teilberechnungen führen zu Ganzzahl bzw. Fließkommaergebnis, die abschließend durch eine Operation verknüpft werden sollen.
- Lösung: Typkonvertierung (engl. type casting)
- Erweiterungsbeziehung zwischen primitiven Datentypen:
- Datentyp an Spitze des Pfeils ist Erweiterung des Datentyps am Anfang des Pfeils.
- in Pfeilrichtung: automatische erweiternde Konvertierung



Explizite Typkonvertierung

- problematischer: Konvertierung gegen Pfeilrichtung
- Wie soll Gleitkommazahl 2.5 in Ganzzahl konvertiert werden?
- Zur ausdrücklich gewünschten expliziten Typkonvertierung stellt Java einen Typkonvertierungsoperator zu Verfügung (eventuell: Genauigkeitsverlust)
- zwei Operanden: (<Zieldatentyp>) <Wert>
    - double x = 2.0; int y = (int) x; // y = 2
    - int z = (int) (x / 3); // x / 3 wird ausgew. zu 0.66..
- Anwendung des (int)-Operators führt zur Rundung hin zur 0, d. h. z wird der Wert 0 zugewiesen.
- Weglassen des Konvertierungsoperators führt hier zu Fehlermeldung

Typsicherheit

- jede Variable, jede Konstante und jedes Literal besitzt einen Typ:
    - Festlegung des gültigen Wertebereichs
    - Festlegung der anwendbaren Operationen
- Typsicherheit: auf Operanden können nur die ihrem Typ entsprechenden Operationen angewandt werden.
- In typsicheren Programmiersprachen kann der Übersetzer für jeden Operanden zu jeder Zeit den zugehörigen Typ ermitteln und damit feststellen, ob eine Operation anwendbar ist:
    - wenn nicht, dann liefert der Übersetzer eine Fehlermeldung.
    - Somit lassen sich anhand/mittels der Typen manche Fehler ermitteln, bevor ein Schaden angerichtet wird; damit verbesserte Korrektheit von Programmen (statische Typsicherheit)

Grundlagen der Programmierung (2)

Ablaufstrukturen

- Anweisung
  - einzelne Vorschrift, die im Rahmen der Abarbeitung eines Programms auszuführen ist und den Programmzustand (z. B. Variablenwerte) ändert
- Sequenz
  - Folge von nacheinander auszuführenden Anweisungen A1, A2, …, An
  - wird verwendet, wenn im Algorithmus Schritte nacheinander ausgeführt werden sollen
- Block
  - Zusammenfassung einer Folge von Anweisungen
  - an allen Stellen innerhalb eines Programms verwendbar, an denen auch eine einzelne Anweisung stehen kann

Hinweis zur Programmzeilenformatierung

- Einrücktiefe stets 4 Leerzeichen, meist per Tabulator.
- Keine Zeile sollte mehr als 80 Zeichen enthalten, Kommentarzeilen nicht mehr als 70 Zeichen.
- Wenn eine Zeile zu lang wird, möglichst an folgenden Stellen umbrechen:
  - nach einem Komma int a, | b
  - vor einem Operator 47 | + 11
  - möglichst gemäß der Bindungsstärke
- Folgezeile wird so weit eingerückt, wie der Ausdruck „auf gleicher Schachtelungsebene“ in der vorhergehenden Zeile.

Bedingte Anweisung, Fallunterscheidung, Alternative

- Erlaubt es, eine Anweisung nur dann ausführen zu lassen, wenn eine zugehörige Bedingung erfüllt ist.
- Bedingte Anweisung in Java: if-else-Anweisung
- Allgemeine Form

```
if (<Bedingung>) {
<Anweisungen1>
} else {
<Anweisungen2>
}
```

- Auswertung:
  - Zuerst wird Bedingung ausgewertet, die als Ergebnis einen Wert vom Datentyp boolean liefern muss.
  - Falls Bedingung erfüllt (true) ist, werden die Anweisungen im then-Block ausgeführt, ansonsten die Anweisung in else-Block
- else-Fall kann auch weggelassen werden, wodurch sich die if-Anweisung ergibt
- Wenn die Bedingung nicht erfüllt ist, wird mit der ersten Anweisung nach der if-Anweisung fortgesetzt.

Bedingter Ausdruck (auch: bedingter bzw. ?-Operator)

- Zweck:
  - spezielle Form der Fallunterscheidung, die als Ergebnis einen Ausdruck liefert
  - if ist kein Operator und hat kein Ergebnis
- Allgemeine Form:

```
<Bedingung> ? <Ausdruck1> : <Ausdruck2>
```

  - Wenn <Bedingung> wahr ist, dann ist <Ausdruck1> das Ergebnis des ?-Operators, sonst ist <Ausdruck2> das Ergebnis

Mehrfachauswahl: Schachtelung einfacher Alternativen

- Prinzip: Anweisung des else-Zweigs enthält weitere bedingte Anweisung
- Formatierung gemäß Konvention

```
if (a > 0) {
        vorzeichen = 1;
} else if (a < 0) {
        vorzeichen = -1;
} else {
        vorzeichen = 0;
}
```

Mehrfachauswahl: switch-Anweisung

- Wenn Werte zur Auswahlsteuerung vom Typ byte, short, int, char, String oder vom Typ enum sind, ist komfortablere Variante der Mehrfachauswahl (sog. switch-Anweisung) verfügbar
- Beispiel:

```
Orientation direction = Orientation.NORTH;
// um 90 Grad im Uhrzeigersinn drehen switch (direction) {
case NORTH:
        direction = Orientation.EAST;
         break;
case EAST:
        direction = Orientation.SOUTH;
        break;
...
default:
}
```

while-Schleife

- Grundgedanke der Steuerung: wiederholte Ausführung von Aktionen, bis eine Zielbedingung erfüllt ist.
- Form:

```
while (<Bedingung>) {
        <Anweisungen>
}
```

- Auswertung
  - Ausführung beginnt mit Auswertung der Bedingung (Schleifentest).
  - Falls diese erfüllt (true) ist, werden die Anweisungen im Schleifen-rumpf ausgeführt, an den Anfang der Schleife zurückgekehrt und auf dieselbe Weise verfahren.
  - Anderenfalls wird Ausführung der Schleife abgeschlossen und Programm setzt mit erster Anweisung nach Schleife fort.
- <Anweisung> sollte das Ergebnis der Bedingung beeinflussen (z. B. durch Veränderung von Variablenwerten, deren Wert Gegenstand der Schleifenbedingung ist), sonst sog. Endlosschleife, Algorithmus terminiert dann nicht.
- while-Schleifen heißen auch abweisende (oder kopfgesteuerte) Schleifen, weil ihr Rumpf u. U. nie durchlaufen wird (wenn die Schleifenbedingung beim ersten Test nicht erfüllt ist).

do-while-Schleife

- Schleife, bei der bekannt ist, dass der Schleifenrumpf mindestens einmal durchlaufen wird.
- Deshalb heißen do-while-Schleifen auch nicht-abweisende (auch: fußgesteuerte) Schleifen
- Standardform:

```
do {
        <Anweisung>
} while (<Bedingung>);
```

for-Schleife (auch: Zählschleife)

- kann verwendet werden, wenn bereits vor der Ausführung bekannt ist, wie oft der Schleifenrumpf durchlaufen werden soll
- Standardform:

```
for (<Initialausdruck>; <Bedingung>; <Inkrementausdruck>) {
        <Anweisungen>
}
```

- Auswertung:
  - Initialisierungsausdr. auswerten; ggf. dort deklarierte Variablen anlegen.
  - Schleifenbedingung auswerten; wenn nicht erfüllt: Schleifenende, also: abweisende (oder: kopfgesteuerte) Schleife
  - Ausführung der <Anweisungen>
  - Ausführung des <Inkrementausdrucks>, der typischerweise die Variablen verändert (hochzählt), die in der Initialisierung gesetzt und im Schleifentest mit Grenzwerten verglichen werden

Einschub: Kommentare

- In Java unterscheidet man zwischen Kommentaren zur Implementierung und Kommentaren zur Dokumentation:
- Implementierungskommentare
  - werden gekennzeichnet durch // (sog. Zeilenkommentar; alles danach bis zum Ende der Zeile gilt als Kommentar) oder /* … */ (sog. Blockkommentar; alles dazwischen - ggf. auch über mehrere Zeilen - gilt als Kommentar),
  - dienen der Erläuterung des Programmtextes,
  - werden auch verwendet, um vorübergehend nicht benötigte Programmteile „auszukommentieren".
- Dokumentationskommentare
  - werden durch /**…*/ gekennzeichnet,
  - erlauben standardisierte Beschreibung von Programmtexten.
  - Mittels des Werkzeugs javadoc lassen sich solche Dokumentations-kommentare aus Quelltexten extrahieren und html-Seiten generieren (analog zur Dokumentation der Java-Standard-Bibliotheken).

Deklaration von Methoden

- Methoden müssen im Rumpf der Klasse deklariert werden:

```
static <Rückgabetyp> <Methodenname> (<Param 1>, …,<Param n>) {
<Anweisung 1>
<Anweisung m>
}
```

- Hinweis: es gibt auch nicht-statische Methoden (Konzept aus der Objektorientierung), deren Deklaration analog, nur ohne das Schlüsselwort static erfolgt. (Details später)
- Deklarierte Methode kann in einem Programmstücks (in einer Methode) aufgerufen (ausgeführt) werden.
- Deklaration als statische Methode:
  - Schlüsselwort static am Beginn der Deklaration
- Rückgabedatentyp:
  - gibt an, von welchem Datentyp der von der Methode zurück-gegebene Wert ist
  - kein Rückgabetyp: Schlüsselwort void anstelle des Rückgabe-datentyps
- Methodenname:
  - bezeichnet die Methode
  - dient zum Aufruf der Methode an der Stelle, an der sie verwendet werden soll
  - gleiche Regeln wie für die Bildung von Variablennamen
- Liste von (formalen) Parametern
  - Eingabewerte an Methoden übergeben, die in Berechnung eingehen
- Block (Rumpf der Methode)
  - enthält die eigentlichen Anweisungen der Methode.
  - Falls der Rückgabetyp der Methode nicht void ist, enthält der Rumpf eine (oder mehrere) sog. Rückgabeanweisung(en) (auch: return-Anweisungen), mittels derer ein zuvor berechneter Wert an die jeweils aufrufende Stelle zurückgegeben wird.
  - Hinweis: Methode (gilt nicht für Typ void) hat genau einen (möglicherweise komplexen, Details bei Objektorientierung) Rückgabewert; je nach Verlauf der Berechnungen kann der aber von verschiedenen Stellen der Methode an die aufrufende Stelle zurückgegeben werden.

Formale Parameter, Parameterliste

- definieren die Schnittstelle der Methode zur Übergabe bzw. Übernahme von Eingabewerten
- Deklaration eines formalen Parameters:
  - analog Variablendeklaration
  - Form: <Datentyp> <Parametername>
- Deklaration im Rahmen der Methodendeklaration:
  - einzeln, oder im Fall von mehreren durch Kommata getrennt, in Klammern nach dem Methodennamen
  - Folge von Parameterdeklarationen heißt auch Parameterliste

Methodenrumpf

- enthält ggfs. Deklaration von sog. lokalen Variablen
- Folge von Anweisungen mit lesenden und schreibenden Zugriffen auf formale Parameter und lokale Variablen
- Unterschied zwischen formalen Parametern und lokalen Variablen: formalen Parametern muss vor Verwendung im Rumpf kein Wert zugewiesen werden, denn sie erhalten einen beim Aufruf
- eine oder mehrere Rückgabeanweisungen (return-Anweisung)
    - Fall 1: Rückgabetyp ist nicht void
        - Rückgabe des Wertes, der sich durch Auswertung des Ausdrucks <Rückgabewert> ergibt
        - Abarbeitung des Methodenrumpfs wird nach Auswertung der return-Anweisung beendet. Dann wird an der Aufrufstelle in der aufrufenden Methode fortgefahren.
    - Fall 2: Rückgabetyp ist void
        - Methoden mit Rückgabetyp void geben keinen Wert zurück
        - Optional kann Ausführung des Methodenrumpfs bei Bedarf explizit mit return; beendet werden.
        - Falls keine return-Anweisung angegeben wurde, wird der Methodenrumpf komplett abgearbeitet und anschließend zur aufrufenden Stelle zurückgekehrt

Methodenaufruf

- Aufruf einer Methode erfolgt durch Angabe ihres Namens, gefolgt von einer geklammerten Liste von Argumenten
- allgemeine Form

```
<Methodenname>(<argument 1>, …, <argument n>);
```

- Aufruf von in Java bereits vorhandenen Methode
    - Umwandlung von String in int: Integer.parseInt(...)
    - System.out.println(...): Ausgabe eines Strings auf Standardausgabe

```
public class Quadrierer {
        public static void main(String[] args) {
                int x = Integer.parseInt(args[0]);
                int xquadrat = x * x;
                System.out.println(xquadrat);
        }
}
```

Beachten der Schnittstelle der Methode beim Aufruf

- Als Argument kann alles eingesetzt werden, was auf der rechten Seite einer Wertzuweisung zu einer Variablen vom Typ des jeweiligen formalen Parameters stehen kann.
- Für jeden formalen Parameter der Parameterliste muss genau ein Argument angegeben werden

Wirkung eines Methodenaufrufs

- Jedem formalen Parameter wird der Wert des/seines Arguments zugewiesen; analog einer Variablendeklaration mit Wertzuweisung
- Anweisungen im Rumpf der Methode mit dem gegebenen Namen werden ausgeführt.
- Ausführung des Rumpfes endet, wenn return-Anweisung erreicht wird (oder bei letzter Anweisung im Fall von void-Methoden); der sich dort ergebende Rückgabewert wird an die aufrufende Stelle zurückgegeben (nicht bei void) und kann dort verwendet werden
- üblich: Methodenaufruf innerhalb eines Ausdrucks
- Beispiel:

```
int x = 5;
int wert = square(x) + x;
```

- Zurückgegebener Wert des Aufrufs square(x) wird zur Auswertung des Ausdrucks square(x) + x verwendet.
- Methoden, die keinen Wert zurückgeben, werden als alleinstehende Anweisung aufgerufen

Sichtbarkeitsbereich von Deklarationen

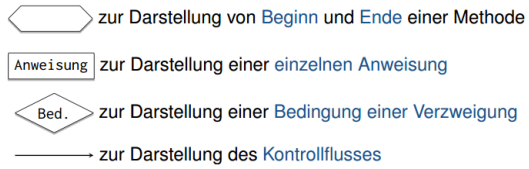
- Methoden sind voneinander unabhängige Unterprogramme, in denen somit auch Variablen deklariert werden können.
- Darf eine Variable mit gleichem Namen in zwei unterschiedlichen Unterprogrammen verwendet werden?
- Ja, in unterschiedlichen Blöcken dürfen Variablen gleichen Namens deklariert werden. Warum?
- Variable ist (nur) in dem Block, in dem sie deklariert wurde, sowie in allen Unterblöcken (weiterer Block, der zwischen den geschweiften Klammern eines Blocks auftritt) dieses Blocks sichtbar und damit verwendbar.
    - sog. Sichtbarkeitsbereich (engl: Scope) der Deklaration: Abschnitte eines Programms, in dem die Variable benutzt werden darf
    - Variable heißt lokal bezüglich des Blocks, in dem sie deklariert wurde.

Rechnerinterne Verwaltung lokaler Variablen mittels Stapel

- Jedes Programm besitzt zur Laufzeit einen sogenannten (Programm-) Stapel (engl. (program) stack), auf dem lokale Variablen bzgl. Blöcken oder Methoden gespeichert werden.
- Beim Betreten eines Blocks bzw. beim Aufruf einer Methode werden die zugehörigen lokalen Variablen auf dem Stapel erzeugt und konzeptuell in der Reihefolge ihrer Erzeugung nacheinander auf den Stapel gelegt.
- Beim Verlassen des Blocks bzw. beim Beenden des Aufrufs werden die lokalen Variablen in umgekehrter Reihenfolge gelöscht und vom Stapel entfernt.
- All das geschieht automatisch.

Ablaufdiagramme

- Ablaufdiagramme dienen zur grafischen Darstellung eines Algorithmus
  - Knoten stellen auszuführende Anweisungen und Bedingungen dar
  - Kanten stellen den möglichen Kontrollfluss dar
- Elemente eines Ablaufdiagramms:



- für Zuweisungen in Anweisungen wird üblicherweise „←“ verwendet

Rekursion

Voraussetzungen, damit rekursiver Ansatz erfolgreich

- Nicht direkt lösbare Teilprobleme müssen dem Ausgangsproblem ähnlich sein (Selbstähnlichkeit)
- Teilprobleme werden bei jedem weiteren rekursiven Aufruf bzgl. einer Ordnung kleiner (Monotonieeigenschaft)
- Größe der Teilprobleme ist nach unten beschränkt (Beschränktheit)

Fakultätsfunktion

- Fakultätsfunktion $f(n) = n!$
- rekursive Definition $n! = \begin{cases} 1 & falls\ n = 1 \\ n * (n-1)! & falls\ n > 1 \end{cases}$

Umsetzung in Java:

Variante 1: mit if-else-Anweisung (Fallunterscheidung)

```
static long fakultaet1(int n) {
        if (n == 1) {
                return 1;
        } else {
                return n * fakultaet1(n - 1);
        }
}
```

Variante 2: mit ?-Operator (bedingter Ausdruck)

```
static long fakultaet2(int n) {
        return (n == 1) ? 1 : (n * fakultaet2(n - 1));
}
```

Rekursion

- Rekursion: allgemeines Prinzip zur Lösung von Problemen, bei der die Berechnung auf die eigene Berechnung direkt oder indirekt zurückverweist
- mit dieser wird auf dieselbe Weise verfahren
- Durchführung bricht ab, wenn eine vorgegebene Abbruchbedingung erreicht wird
- Beispiel für die Anwendbarkeit von Rekursion ist das „Teile und Herrsche/Divide and Conquer-Prinzip“:
    - Zerlege ein großes Problem in kleinere Versionen des Problems.
    - Falls die kleineren Versionen noch zu groß sind, löse sie auf die gleiche Weise.
    - Falls die kleineren Versionen hinreichend klein sind, löse sie direkt.
    - Füge die Lösung kleinerer Versionen des Problems zu einer Lösung des großen Problems zusammen.

(Linear) Rekursive Methode, rekursive Methodendefinition

- Eine Methode f heißt rekursiv, gdw. zur Berechnung von f diese Methode f wieder benutzt wird.
- Eine Methode f heißt linear rekursiv, wenn sie in jedem Zweig einer Fallunterscheidung ihres Rumpfes höchstens einmal aufgerufen wird.

Korrektheitsbeweis (für alle möglichen Eingaben):

- mit geeigneten Beweismethoden den Nachweis erbringen, dass die Methode für jede beliebige Eingabe immer das korrekte Ergebnis bzgl. der Spezifikation berechnet
- im Zusammenhang mit rekursiven Methoden kommt dabei oft die Beweistechnik der vollständigen Induktion zum Einsatz

Vollständige Induktion

- Vollständige Induktion: mathematische Beweismethode, mit der eine Aussage für alle natürlichen Zahlen nn bewiesen wird
- da Menge der natürlichen Zahlen unendlich ist, kann ein solcher Beweis nicht für alle Einzelfälle durchgeführt werden
- Beweis erfolgt daher i. d. R. in zwei Etappen:
    - sog. Induktionsanfang: beweise die Aussage für eine kleinste Zahl n_0 (meist 0 oder 1)
    - sog. Induktionsschluss (oder: -schritt): beweise unter der Hypothese, dass die Aussage für ein beliebiges $n \geq n_0$ bereits gilt (sog. Induktionsvoraussetzung), dass die Aussage dann auch für n+1 gilt
- Basisform der Induktion (Variante 1: „+1“)
    - Bedingungen (zu beweisen):
        - $A(1)$ ist wahr
        - Für jedes $n \geq 1$ gilt: $A(n) \Rightarrow A(n+1)$
    - Dann gilt $A(n)$ für alle n.

- Basisform der Induktion (Variante 2: „-1“)
  - Bedingungen (zu beweisen):
    - $A(1)$ ist wahr
    - Für jedes n>1 gilt: $A(n-1) \Rightarrow A(n)$
  - Dann gilt $A(n)$ für alle n.

Zur Terminierung

Gefahr: fehlerhafter Entwurf einer rekursiven Methode (z. B. nicht-rekursiver Basis-/Abbruchfall vergessen) führt möglicherweise dazu, dass diese nicht terminiert (sog. Endlosrekursion)

Terminierungsüberlegungen

- Terminierungsbeweis (für alle möglichen Eingaben)
  - Vorgehensweise:
    - e_0, e_1, e_2, ... sei Argumentfolge der Rekursionsschritte bei Auswertung eines Methodensaufrufs
    - finde ganzzahlige Terminierungsfunktion t(e_i) und beweise (z. B. per Induktion):
      - t fällt bei jedem Rekursionsschritt streng monoton
      - t ist nach unten beschränkt

Größter gemeinsamer Teiler (ggT)

- größter gemeinsamer Teiler zweier ganzer Zahlen m und n $ggt(m,n)$ ist die größte natürliche Zahl, durch die sowohl m als auch n ohne Rest teilbar sind
- Verfahren zur Bestimmung des ggTs:
  - Variante 1: Primfaktorzerlegung
  - Variante 2: klassischer Euklidischer Algorithmus
    - subtrahiere die kleinere Zahl von der größeren
    - ersetze größere ursprüngliche Zahl durch die so gewonnene Differenz
    - wiederhole diese Schritte solange, bis die beiden Zahlen gleich groß sind
    - Ergebnis ist der ggT der beiden Zahlen
  - Implementierung in Java

```
static int ggT(int a, int b) {
        if (a == b) {
                return a;
        } else if (a < b) {
                return ggT(a, b - a);
        } else {
                return ggT(a - b, b);
        }
}
```

- Variante 3: optimierter Euklidischer Algorithmus
  - Nachteil des klassischen Euklidischen Algorithmus: bei großen Ausgangszahlen ggf. viele Subtraktionen erforderlich deshalb:
    - in aufeinander folgenden Schritten jeweils Division mit Rest
    - Divisor wird im nächsten Schritt zum neuen Dividenden und Rest zum neuen Divisor
    - Divisor, bei dem sich Rest 0 ergibt, ist der ggT der Ausgangszahlen
  - Implementierung in Java:

```
static int ggT2(int a, int b) {
        if (b == 0) {
                return a;
        } else {
                return ggT2(b, a % b);
        }
}
```

Endrekursion

- Eine Methode f heißt wenn der rekursive Methodenaufruf stets die letzte Aktion des Zweigs zur Berechnung von f ist.
- Endrekursion ist ein Spezialfall der linearen Rekursion.
- Eine endrekursive Methode kann unmittelbar entrekursiviert werden. Das ist deshalb erstrebenswert, da rekursive Lösung zwar i. d. R. intuitiv aufzuschreiben, iterative Lösungen i. d. R. aber schneller in der Ausführung sind

Entrekursivierung endrekursiver Methoden

- Entrekursivierung: überführe eine rekursive Methode in eine bzgl. der Spezifikation äquivalente iterative Methode

Entrekursivierung am Beispiel der Methode sum

1. rekursive (aber nicht endrekursive) Implementierung

```
static int sum1(int n) {
        if (n == 0) {
                return 0;
        } else {
                return n + sum1(n-1);
        }
}
```

2. endrekursive Implementierung

```
static int sum2(int n) {
        return add_sum(0, n);
}
static int add_sum(int m, int n) {
        if (n == 0) {
                return m;
        } else {
                return add_sum(m+n, n-1);
        }
}
```

entrekursivierte Implementierung:

```
static int sum3(int n) {
        int m = 0;
        while (n > 0) {
                m = m + n;
                n = n - 1;
        }
        return m;
}
```

- typische Frage im Zusammenhang der Algorithmik: gibt es eine noch bessere (effizientere) Lösung?
- ja: Gaußsche Summenformel anwenden
- Warum ist diese Lösung effizienter?
  - Anzahl der Operationen als Vergleichsmaß
  - bei sum3 ist die Anzahl der Schleifendurchläufe abhängig von n: Schleife wird n mal durchlaufen
  - bei sum4 ist die Anzahl der Operationen konstant und unabhängig vom Parameter n

```
static int sum4(int n) {
        return (n * (n + 1)) / 2;
}
```

Warum Rekursion und warum Entrekursivierung?

- Frage: Warum führen wir Rekursion als Programmiertechnik ein, wenn wir dann anschließend versuchen, rekursiv gestaltete Problemlösungen wieder zu entrekursivieren?
- für Rekursion spricht:
  - Elegante, intuitive, übersichtliche Problemlösungen.
  - Programmtext ist im Allg. lesbarer und weniger fehleranfällig.
  - Besonders geeignet für Algorithmen, die auf induktiv definierten Daten arbeiten, wodurch das Problem leicht in kleinere selbstähnliche Teilprobleme zerfällt.
  - Beweise von Eigenschaften, wie Übereinstimmung mit Spezifikation und Terminierung, sind oft nahe liegend per vollständiger Induktion.
- Verwendung von Endrekursion zur Effizienzsteigerung:
  - Abarbeitung einer rekursiven Methode:
    - Speicherplatzverbrauch steigt linear mit der Rekursionstiefe.
    - Bei jedem Methodenaufruf wird Speicherplatz auf dem Laufzeitstapel für das Sichern der Rücksprungadresse, der Methodenparameter und ggf. methodenlokaler Variablen belegt.
  - Bei endrekursivem Methodenaufruf:
    - Die im für die aufrufende Methode belegten Speicherbereich abgelegten Werte werden nur noch für die Parameterübergabe an die endständig aufgerufene Methode benötigt.
    - Dieser Speicherbereich kann wiederverwendet werden.
    - Speicherbedarf unabhängig von Rekursionstiefe.
  - Manche optimierende Übersetzer entrekursivieren Endrekursion automatisch.

Türme von Hanoi

- Problemspezifikation gegeben:
  - k Scheiben unterschiedlichen Durchmessers, der Größe nach geordnet auf Position A
  - Positionen B und C ohne Scheiben
- Problem: versetze Scheiben-Turm von A nach B, wobei
  - jede Scheibe nur einzeln bewegt werden kann
  - immer nur die oberste Scheibe eines Turmes bewegt werden kann
  - niemals eine größere Scheibe auf einer kleineren liegen darf und
  - C zur Zwischenablage (als Hilfsstapel) benutzt werden kann

Algorithmenentwurf

- Anwendung des Induktionsprinzips
  - Basisfall k=0: keine Scheiben; trivial, weil sofort fertig
- Rückführung auf kleinere Probleme (k>0)
  - Teilproblem 1: versetze die oberen k−1 Scheiben von A nach C unter Verwendung von B als Zwischenablage; kleineres Problem: rekursiv lösbar
  - Teilproblem 2: versetze die verbleibende k-te Scheibe von A nach B
  - Teilproblem 3: Versetze die k−1 Scheiben von C nach B unter Verwendung von A als Zwischenablage; kleineres Problem: rekursiv lösbar
  - Verknüpfung: Hintereinanderausführung der 3 Teillösungen

```
static void hanoi(int scheiben, char start, char ziel, char hilfe) {
        if (scheiben > 0) {
                hanoi (scheiben - 1, start, hilfe, ziel);
                System.out.println("Versetze Scheibe " + scheiben + " von " + start +
                                                                    " nach "ziel);
                hanoi(scheiben - 1, hilfe, ziel, start);
        }
        // else Fall = Basisfall der Induktion ist leer
}
```

Kaskadenartige Rekursion

- Eine Methode f heißt kaskadenartig rekursiv, wenn in einem Zweig einer Fallunterscheidung im Rumpf von f zwei oder mehr Aufrufe von f auftreten.
- Kaskadenartig rekursive Methoden sind gekennzeichnet durch eine baumartige Aufrufstruktur und ein lawinenartiges Anwachsen der anfallenden rekursiven Aufrufe.
- Kaskadenartige Rekursion lässt sich nicht (leicht) in Iterationen umwandeln.

Fibonacci-Zahlen

- ursprüngliche Fragestellung aus dem Jahr 1202:
  - gegeben:
    - ein neugeborenes Kaninchenpaar
    - jedes Kaninchenweibchen bringt jeden Monat ein neues Paar zur Welt
    - junge Kaninchen haben nach 2 Monaten ihre ersten Jungen
    - Kaninchen sterben nicht
  - gesucht: Wie viele Kaninchenpaare gibt es nach 12 Monaten?

Algorithmenentwurf

Entwurf mittels Induktionsprinzip

- Basisfall:
  - n = 1: fibonacci(1) = 1
  - n = 2: fibonacci(2) = 1
- Rückführung auf kleinere Probleme:
  - Teilproblem 1: fibonacci(n – 2) (zweimonatige Kaninchen haben erste Nachkommen)
  - Teilproblem 2: fibonacci(n – 1) (Kaninchen aus dem vergangenem Monat bleiben da)
  - Verknüpfung: Addition
- der Fall n wird also auf zwei kleinere Fälle (kaskadenartige Rekursion) und zwei Basisfälle zurückgeführt
- also (rekursive Definition):
  - fibonacci(1) = 1 und fibonacci(2) = 1,
  - fibonacci(n) = fibonacci(n – 2) + fibonacci(n – 1)

Implementierung in Java

Variante 1: mit if-else-Anweisung

```
static int fibonacci1(int n) {
        if (n <= 2) {
                return 1;
        } else {
                return fibonacci1(n - 2) + fibonacci1(n - 1);
        }
}
```

Variante 2: mit ?-Operator

```
static int fibonacci2(int n) {
        return (n <= 2) ? 1 : fibonacci2(n - 2) + fibonacci2(n - 1);
}
```

Verschränkte (auch: wechselseitige) Rekursion

- Zwei Methoden f und g heißen verschränkt (auch: wechselseitig) rekursiv, wenn die Methodendeklaration von f einen Aufruf der Methode g enthält und die Methoden-deklaration von g einen Aufruf der Methode f enthält.
- Verschränkte Rekursion liegt auch vor, wenn sich mehr als zwei Methoden zyklisch gegenseitig aufrufen.
- Beispiel:
- die Methoden istGerade und istUngerade sind verschränkt rekursiv.

Test, ob ganze Zahl n gerade bzw. ungerade ist

```
static boolean istGerade(int n) {
        if (n == 0) {
                return true;
        } else {
                return istUngerade(n - 1);
        }
}
static boolean istUngerade(int n) {
        if (n == 0) {
                return false;
        } else {
                return istGerade(n - 1);
        }
}
```

Verschachtelte Rekursion

- Eine Methode f heißt verschachtelt rekursiv, wenn die rekursiven Aufrufe von f nicht nur im Rumpf der Methoden-deklaration von f sondern zusätzlich auch in einem Argumentausdruck eines rekursiven Aufrufs von f auftreten.
- Bei verschachtelt rekursiven Methoden sind Terminierungs-beweise i. d. R. schwer.
- Verschachtelt rekursive Methoden sind damit in der Praxis eher unbedeutend.
- Beispiel:
    - Die ackermann-Funktion ist verschachtelt rekursiv; sie ist in der Informatik von großer theoretischer Bedeutung.

Ackermann-Funktion

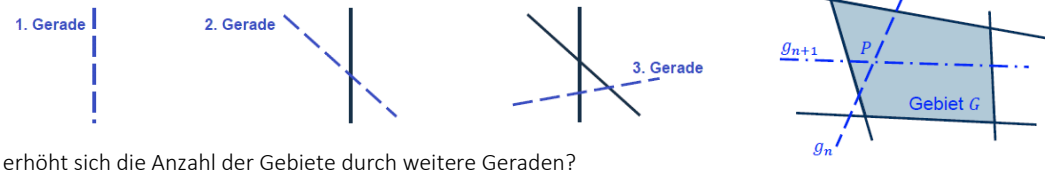
```
static int ackermann(int m, int n) {
        if (m == 0) {
                return n + 1;
        } else {
                if (n == 0) {
                        return ackermann(m - 1, 1);
                } else {
                        return ackermann(m - 1, ackermann(m, n - 1));
                }
        }
}
```

Teil 1: Beispiele zur Algorithmenherleitung

Durch Geraden erzeugte Gebiete in der Ebene

Behauptung: n Geraden in der Ebene, die in allgemeiner Position zueinander liegen, bilden $\frac{n+(n+1)}{2}$ +1 Gebiete

Erläuterung: „allgemeine Position“ bedeutet dabei, dass, keine zwei Geraden parallel sind und keine drei Geraden sich im selben Punkt schneiden



Frage: wie erhöht sich die Anzahl der Gebiete durch weitere Geraden?

Strategie: nicht direkt die Gebietsanzahl ermitteln, sondern über das Wachstum der Gebietsanzahl beim Hinzufügen einer weiteren Gerade nachdenken.

Behauptung: durch Hinzufügen einer Geraden zu n – 1 Geraden in allgemeiner Position in der Ebene erhöht sich die Zahl der Gebiete um n

Beweis durch vollständige Induktion:

- Induktionsanfang:
  - bei 0 Geraden gibt es 1 Gebiet; kommt die erste Gerade dazu, dann ergeben sich zwei Gebiete, eines auf der einen und eines auf der anderen Seite der Geraden (Gebietsanzahl dann 2).
- Induktionsschritt „n → n +1“:
  - Wegen der allgemeinen Lage gilt, dass eine Gerade ein Gebiet entweder in zwei Teile schneidet, oder nicht einmal berührt. Es muss also gezeigt werden, dass die (n+1)-te Gerade (n+1) Gebiete teilt.
  - Wenn man die n-te Gerade g_n entfernen würde, dann würde die n+1-te Gerade g_{n+1} als n-te Gerade eingefügt und laut Induktionsvoraussetzung n Gebiete hinzufügen, also schneiden (IV1).
  - Es muss also nur gezeigt werden, dass in Gegenwart der n-ten Geraden die n+1-te Gerade ein zusätzliches Gebiet schneidet.
  - Es gibt genau ein Gebiet G, in dem sich die n-te und n+1-te Gerade schneiden (im Schnittpunkt P).
  - Wenn die n-te Gerade g_n nicht vorhanden wäre, dann würde die n+1-te Gerade g_{n+1} das Gebiet in zwei Teile schneiden: dort entsteht ein neues Gebiet von insgesamt n neuen (IV2).
  - Wenn Gerade g_n vorhanden ist, dann fügt die Gerade g_{n+1} zwei neue Gebiete hinzu, also insgesamt n+1 neue Gebiete. q.e.d.

Bemerkungen:

- keine Induktion über Gebietszahl, sondern über das Wachstum der Gebietszahl
- Induktionshypothese wurde doppelt verwendet:
  - für die Gebiete, die durch die n-te Gerade eingeführt werden (IV1)
  - für die Gebiete, die durch die n+1-te Gerade eingeführt werden, wenn die n-te Gerade nicht vorhanden ist (IV2)

Färben von Gebieten

Behauptung: Gebiete, die durch verschiedene Geraden in einer Ebene geformt werden, können so mit zwei Farben eingefärbt werden, dass keine zwei Gebiete die gleiche Farbe haben, wenn sie eine gemeinsame Kante haben (allgemeine Lage ist nicht erforderlich!).

Beweis durch vollständige Induktion:

- Induktionsanfang:
  - Fläche wird von einer Geraden in zwei Teile geteilt, die trivial mit zwei Farben gefärbt werden können
- Induktionshypothese:
  - Gebiete, die durch weniger als nn Geraden geformt werden, können mit zwei Farben gefärbt werden
- Induktionsschritt („strenge Induktion, $\forall k < n \rightarrow n$“):
  - wenn die n-te Gerade hinzugefügt wird, werden zwei Gruppen von Gebieten gebildet
  - die eine Gruppe befindet sich auf der einen, die andere auf der anderen Seite der n-ten Geraden
  - die Farben aller Gebiete in nur einer Gruppe werden ausgetauscht
- betrachte zwei benachbarte Gebiete $G1$ und $G2$:
  - Fall 1: $G1$ und $G2$ befinden sich auf einer Seite der n-ten Gerade
  - nach Induktionsvoraussetzung hatten sie vor Einführung der n-ten Gerade unterschiedliche Farben
  - hinterher immer noch, ggf. aber getauscht
  - Fall 2: $G1$ und $G2$ entstehen durch die n-te Gerade (d.h. n-te Gerade bildet gemeinsame Kante von $G1$ und $G2$)
  - vor Einführung der n-ten Gerade hatten $G1$ und $G2$ die gleiche Farbe
  - durch den Farbtausch haben sie nach Einführung der n-ten Gerade unterschiedliche Farben q.e.d.

Gray-Codes

Problem:

- Sich stetig (nicht sprunghaft) ändernde Daten (z. B. Signale eines Temperatursensors) werden digitalisiert und sollen über parallele Datenleitungen übertragen werden.
- Abhängig von der Messwertänderung müssten sich ggf. Bits auf mehreren Leitungen exakt gleichzeitig ändern.
- Bei echter Hardware ändern sich die Bits auf den Leitungen nicht exakt gleichzeitig, z. B. aufgrund von Bauteilestreuung, Laufzeiten, Asymmetrien usw.
- Dadurch kommt es zu ungewollten Zwischenzuständen

Gray-Code: zyklischer Binärcode mit der Eigenschaft, dass sich aufeinander folgende Code-Wörter in genau einem Bit unterscheiden

- bei Gray-Codes entstehen keine Übertragungsfehler bei stetig veränderlichen Signalen
- Beispiele für Gray-Codes der Länge n

1-Bit-Gray-Code ($n = 2$)	*2-Bit-Gray-Code* ($n = 4$)
0	00
1	01
	11
	10

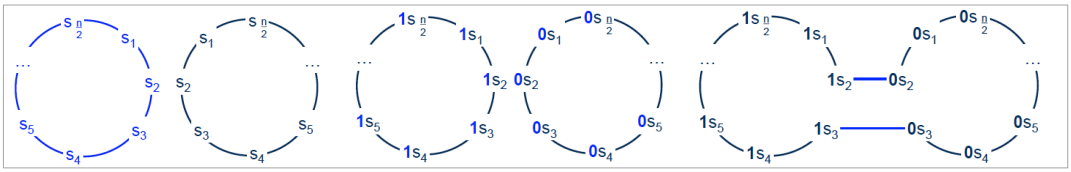
- Hinweis: für gerade n, die keine Zweierpotenzen sind, gibt es auch eine erweiterte Gray-Code-Definition

Behauptung:

- Für jedes $n = 2^s > 0, n \in \mathbb{N}$ existiert ein Gray-Code der Länge n, wobei jedes Codewort aus $s = \log_2 n$ Bits besteht.

Beweis durch Induktion:

- Induktionsanfang:
  - für n=1 ist 0 der gesuchte Gray-Code
  - für n=2 ist 0,1 der gesuchte Gray-Code
- Induktionshypothese: es existiert ein Gray-Code der Länge $\frac{n}{2}$
- Induktionsschritt:
  - verdopple Gray-Code
  - ergänze Präfix 0 bzw. 1
  - füge beide zusammen



Maximale Summe einer zusammenhängenden Teilfolge

Problemspezifikation:

- gegeben:
  - Folge x1, x2, …, xn von Gleitkommazahlen (ggf. auch negativ)
- gesucht:
  - zusammenhängende Teilfolge $x_i, x_{i+1}, \ldots, x_j$ so dass die Summe ihrer Elemente maximal ist (es soll also keine zusammen-hängende Teilfolge mit höherer Elementsumme geben.)
- Die Summe der leeren Teilfolge wird als 0 festgesetzt.

Beispiel:

Buchungsvorgänge auf einem Bankkonto: Einzahlungen und Auszahlungen gesucht wird die zusammenhängende Folge von Buchungsvorgängen, die zur größten Summe von Einnahmen führt, um z. B. einen potenziellen Geldgeber durch Beleg dieser Vorgänge von der eigenen Finanzsituation zu überzeugen

- Induktionshypothese: maximale Summe einer zusammenhängenden Teilfolge in einer Folge der Länge kleiner als n kann berechnet werden
- Basisfall: bei n = 1 ist die Folge ein-elementig
  - Falls $x_1 < 0$: maximale Summe ist 0, leere Teilfolge
  - andernfalls: maximale Summe ist x_1, Teilfolge = Gesamtfolge

- Induktionsschritt „n – 1 → n“: laut Induktionshypothese kann die maximale Teilfolge in (x_1, x_2, …, x_{n-1}) berechnet werden
  - falls maximale Teilfolge leer: betrachte nur x_n, wie im Basisfall
  - andernfalls sei (x_i, x_{i+1}, …, x_j) die maximale Teilfolge:
    - Falls j = n – 1: (maximale Teilfolge am Ende, sog. Suffix-Folge):
      - falls x_n positiv, Verlängern der maximalen Teilfolge um x_n
      - sonst: maximale Teilfolge bleibt unverändert
    - andernfalls: (maximale Teilfolge irgendwo in der Mitte)
  - Problem: neue Suffix-Folge könnte einen höheren Wert liefern

Idee: Man merkt sich nicht nur die bisher erreichte maximale Summe, sondern auch die maximale Summe derjenigen Teilfolge, die beim Element x_n aufhört

- allgemeinere Aufgabenstellung, Induktionshypothese: Für eine Folge der Länge kleiner als n kann berechnet werden:
  - maximale Summe einer zusammenhängenden Teilfolge
  - maximale Summe einer Suffix-Folge
- Basisfall: wie oben
- Induktionsschritt „n – 1 → n“:
  - größtenteils wie oben …
  - falls maximale Teilfolge „irgendwo in der Mitte“: maximale Suffixfolge ggf. um x_n verlängern, mit der bisherigen maximalen Teilfolge vergleichen, ggf. neue maximale Teilfolge setzen

Um aus der Lösung für die reduzierte Problem- größe eine Lösung für das gegebene Problem zu konstruieren muss manchmal eine Teillösung bei der Rekursion mitgeführt/fortgeschrieben werden

Implementierung in Java:

```
static public double maxSubVektor(double[] x) {
        double xMax = 0.0;
        double xSuffix = 0.0;
        for (int i = 0; i < x.length; i++) {
                xSuffix = Math.max(xSuffix + x[i], 0.0);
                xMax = Math.max(xMax, xSuffix);
        }
        return xMax;
}
```

Prominentenproblem

Gegeben sind n Personen. Gesucht: Gibt es einen „Prominenten“ und wenn ja, wer ist es?

Ein „Prominenter“ unter den Personen ist eine Person, die jeder kennt, die aber selbst keinen anderen kennt.
Die Identifizierung des Prominenten findet durch Fragen der Art: „Kennt Person A Person B ?“ statt.

Ziel: Minimierung der Zahl der Fragen

Beachte:
- Die Beziehung „sich kennen“ ist asymmetrisch.
- Der Prominente kennt sich selbst
- Ziel ist die Minimierung der Anzahl der Fragen
- Ein naiver Algorithmus benötigt n · (n – 1) Fragen zum vollständigen Aufbau der Beziehungsmatrix

Problemspezifikation
- gegeben: n · n-Beziehungsmatrix K
- gesucht: Bestimme, ob ein ***j*** existiert, so dass gilt:
  - für alle i gilt: K[i, j] = 1 (Spalte mit lauter 1-Einträgen)
  - für alle k ≠ j gilt: K[j, k] = 0 (Zeile mit lauter 0-Einträgen, außer bei j selbst)

Induktionsansatz, erster Versuch
- Basisfall: in einelementiger Menge ist Person prominent
  - Jeder aus der Menge kennt die Person.
  - Die Person kennt keinen der anderen – was kein Wunder ist, weil es keinen anderen in der Menge gibt.
  - Die „Beziehungsmatrix“ erfüllt obige Forderungen.
- Induktionshypothese: Man kann in den ersten n – 1 Personen bestimmen, ob es einen Prominenten gibt und wer dies ggf. ist.

- Induktionsschritt „n – 1 → n“:
  - Fall 1: Der Prominente ist unter den ersten n – 1 Personen.
    - Prüfe, dass die n-te Person den Prominenten kennt, aber nicht umgekehrt. Es müssen also zwei Fragen gestellt werden.
  - Fall 2: Der Prominente ist die n-te Person.
    - 2(n – 1) Fragen müssen gestellt werden!
  - Fall 3: Es gibt unter den n Personen keinen Prominenten.
    - 2(n – 1) Fragen müssen gestellt werden (schlimmstenfalls; zur Berechnung der kompletten Matrix)!

Für Person n braucht man 2(n – 1) Fragen. Für Person n – 1 daher 2(n – 2).Insgesamt ähnlich viele Fragen wie beim Trivialansatz. Benötigt wird eine schlauere Methode, um die Problemgröße von n auf n – 1 zu reduzieren. Einfache Verwendung des letzten Elements ist (oft) ungeeignet

Einsicht (für unterschiedliche Personen A und B):
- Falls K(A, B) = 1, kann A kein Prominenter sein (A kennt jemanden).
- Falls K(A, B) = 0, kann B kein Prominenter sein (A kennt B nicht).

Mit einer Frage kann also für eine der beiden Person (entweder A oder B) ausgeschlossen werden, dass sie der Prominente ist. Bei der Reduktion von n auf n – 1 wird nicht die n-te Person weggelassen, sondern d. Nicht-Prominente von A u. B.Damit treten weniger Fälle auf. Um die Problemgröße von n auf n – 1 zu reduzieren, fragt man die n-te Person nach irgendeiner Person p der übrigen n – 1 Personen. Nach obiger Einsicht entweder die n-te Person oder p als Nicht-Promi entlarven → neue (n – 1)-Personen-Menge ohne diesen Nicht-Promi.

Löse das Problem gemäß Induktionsvoraussetzung. Dann
- Fall 1: Der Prominente ist unter den n – 1 Personen. Es kann mit zwei Fragen leicht festgestellt werden, ob dieser Prominente auch innerhalb der n Personen prominent ist:
  - kennt die n-te Person den Prominenten?
  - kennt der Prominente die n-te Person?
- Fall 2: Der Prominente ist die n-te Person. Kann nicht auftreten: durch Auswahl gemäß Einsicht wird ein Nicht-Prominenter entfernt.
- Fall 3: Es gibt unter den n – 1 Personen keinen Prominenten. Dann gibt es auch keinen Prominenten innerhalb der n Personen, weil bei Auswahl gemäß Einsicht ein Nicht-Prominenter entfernt wurde.

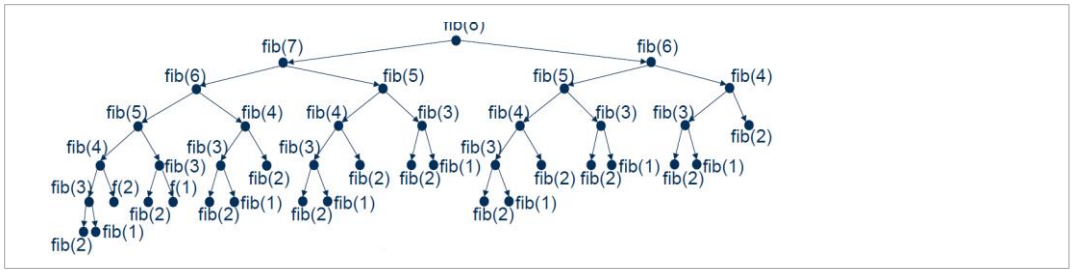
pro Induktionsschritt: eine Frage zur Vorbereitung der Auswahl, eventuell zwei weitere Fragen im Fall 1. Es zeigt sich also: Arbeit zur Reduktion der Problemgröße lohnend

Teil 2: Von Aufrufbäumen und Suchräumen

Noch einmal Fibonacci-Zahlen Laufzeitvergleich

- Rekursive Lösung hat eine deutlich höhere Laufzeitdauer als die vergleichbare iterative Lösung

Ursache des extremen Laufzeitsverhaltens



- Erkenntnis: Mehrfache Auswertung der meisten fib-Aufrufe längere Laufzeiten der naiven rekursiven Variante.

Gegenmaßnahmen:

- Beschneidung des Aufrufbaums
- Idee Wert aufbewahren und wiederverwenden
- Beschnittener Aufrufbaum
  - Es lohnt sich, den rekursiven Aufrufbaum zu beschneiden und benötigte Teilergebnisse nur einmal zu berechnen. Zwei fundamentale Techniken:
    - Durchreichen von Zwischenergebnissen
    - Dynamisches Programmieren

Durchreichen von Zwischenergebnissen

Wenn:

- Es ist eine kaskadenartige Rekursion.
- Teilprobleme werden nur minimal kleiner.
- Es werden nur Zwischenergebnisse verwendet, die „kleinen Abstand haben“.

Dann:

- Zwischenergebnisse durchreichen

Beispiel Fibonacci

```
static long fibR(int n) {
        return (n <= 2) ? 1 : fibR(n-1) + fibR(n-2);
}
```

```
static long fibAux(long minus2, long minus1, int i, int n) {
        return (i >= n) ? minus2 : fibAux(minus1, minus1 + minus2, i+1, n);
}
static long fib(int n) {
        return fibAux(1, 1, 1, n);
}
```

Behandlung der Zwischenergebnisse

- Zwischenergebnisse „rücken auf“ und werden zur Berechnung neuer Zwischenergebnisse verwendet.
- Speicherbedarf: 2 Zwischenergebnisse in jeder der n Methodenschachteln.
- Keine kaskadenartige Rekursion mehr

Terminierung und Ergebnis

- Mit i Abstieg bis auf tiefste Rekursions-ebene (i=n), also zum Basisfall.
- Dann wird Ergebnis von Aufruf zu Aufruf zurück nach oben kopiert

Nach diesem Umbau zum Durchreichen der Zwischenergebnisse kann die lineare Rekursion entrekursiviert werden →(schnelle) iterative Variante. (Nur 2 Zwischenergebnisse; ohne künstliches Hochkopieren des Resultats.)

Dynamisches Programmieren

Wenn:

- Es ist eine kaskadenartige Rekursion.
- Teilprobleme werden nur minimal kleiner.
- Es werden „so wenige“ Ergebnisse wieder- verwendet, dass man diese speichern und nachschlagen kann.

Dann:

- Teilergebnisse zwischenspeichern und wiederverwenden, falls vorhanden.
- Dynamisches Programmieren

Beispiel Fibonacci

```
static long fibR(int n) {
        return (n <= 2) ? 1 : fibR(n-1) + fibR(n-2);
}
```

```
static long[] mem = new long[MAX_N];
static final long UNKNOWN = -1L;

//in main:
        java.util.Arrays.fill(mem, UNKNOWN);      //unset
        mem[1] = 1L; mem[2] = 1L;                 //Basisfaelle

static long fibDP(int n) {
        if (mem[n] == UNKNOWN)                    //Rekursion nur bei unbek. Erg.
                mem[n] = fibDP(n-2) + fibDP(n-1);
        return mem[n];
}
```

Array mem dient der Zwischenspeicherung.nicht im Lösungsraum; zeigt, dass Ergebnis noch nicht vorhanden

a) speichert Zwischen-ergebnis in mem[3]
b) Zwischenergebnis ist hier bereits in mem[3] vorhanden. Aufrufbaum beschnitten

Fast so schnell wie iterative Variante! Aber: Speicherbedarf für das Zwischenergebnis-Array. Laufzeitaufwand der Rekursion/viele Methodenschachteln. Erkenntnis: Da mem[i] nur von mem[i-2] und mem[i-1] abhängt, könnte man auch iterativ entgegen der Pfeilrichtung vorgehen

Beispiel Fibonacci mit Memoization Tafelübung

```
static long fibR(int n) {
        return (n <= 2) ? 1 : fibR(n-1) + fibR(n-2);
}
```

```
public static long fibDP (int n ) {
        long [] table = new long [ n +1]; // Tabelle für die Memoization
        return fibHelper (n , table );
}
public static long fibDPHelper ( int n , long [] table ) {
        // Lookup
        if ( table [ n ] != 0) {
                return table [ n ];
        }
        long result ;
        if ( n == 1 || n == 2) {
                result = 1;
        } else {
                result = fibDPHelper (n -1 , table ) + fibDPHelper (n -2 , table );
        }
        // Memoization
        table [ n ] = result ;
        return result ;
}
```

Beispiel: Binomialkoeffizienten

```
public static long binomBad ( int n , int k ) {
        if ( k == 0 || n == k ) {
                return 1;
        }
        return binomBad (n -1 , k -1) + binomBad (n -1 , k );
}
```

```
public static long binomNice ( int n , int k ) {
        long [][] table = new long [ n +1][];
        for (int i = 0; i < table . length ; i ++) {
                table [ i ] = new long [ i +1];
        }
        return binomNiceHelper (n , k , table );
}
public static long binomNiceHelper ( int n , int k , long [][] table ) {
        // Lookup
        if ( table [ n ][ k ] != 0) {
                return table [ n ][ k ];
        }
        long result ;
        if ( k == 0 || k == n ) {
                result = 1;
        } else {
                result = binomNiceHelper (n -1 , k -1 , table ) + binomNiceHelper (n -1
                                                                        , k , table );
        }
        table [ n ][ k ] = result ;
        return result ;
}
```

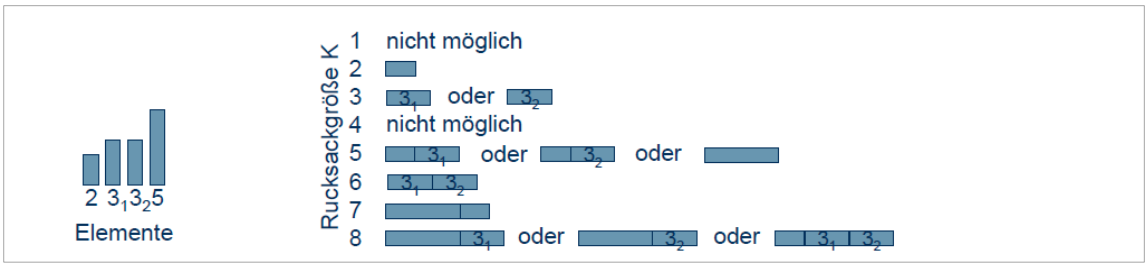
Von der Rekursion zur Memoization

- rekursive Version implementieren
- Tabelle zum Zwischenspeichern der Werte erzeugen und initialisieren
- rekursive Version um Lookup/Memoization erweitern,
  - Tabelle als zusätzliche Eingabe
  - zu berechnenden Wert in der Tabelle nachschlagen (; Lookup)
  - falls bereits berechnet: Wert direkt zurückgeben
  - ansonsten:
    - Wert rekursiv berechnen ,Wert in der Tabelle zwischenspeichern (; Memoization)
    - Wert zurückgeben

Rucksackproblem

Problem: Es soll ein Rucksack mit Fassungsvermögen K mit einer Teilmenge von n Elementen verschiedener Größe ganz voll gepackt werden (also ohne verbleibenden Leerraum). Existiert eine Lösung?

Beispiel:



Problemspezifikation:

- gegeben:
  - Rucksackgröße $K \in \mathbb{N}$
  - Menge von Elementen $S=\{k_1,k_2,\ldots,k_n\}$ mit $k_i \in \mathbb{N}$
- gesucht:
  - $T \subseteq S$, so dass $\Sigma k_{i \in T}$ $k_i = K$, wenn existent
- Kurzschreibweise $P(i,K)$: liefert Elementmenge T oder $\emptyset$
- Für die Rückführung auf kleinere Probleme ist die tatsächliche Größe der Elemente unerheblich.
- Die Elemente seien in beliebiger aber fester Reihenfolge durchnummeriert.

Rekursionsidee – formal

Induktionshypothese: $P(n-1,k)$ kann für alle $0 \le k \le K$ berechnet werden
Induktionsschritt: reduziere $P(n,K)$ auf

- Teilproblem 1 (Element k_n wird nicht verwendet): $P(n-1,K)$
- Teilproblem 2 (Element k_n wird verwendet): $P(n-1,K-k_n n)$ (nur, falls $K-k_n \ge 0$)

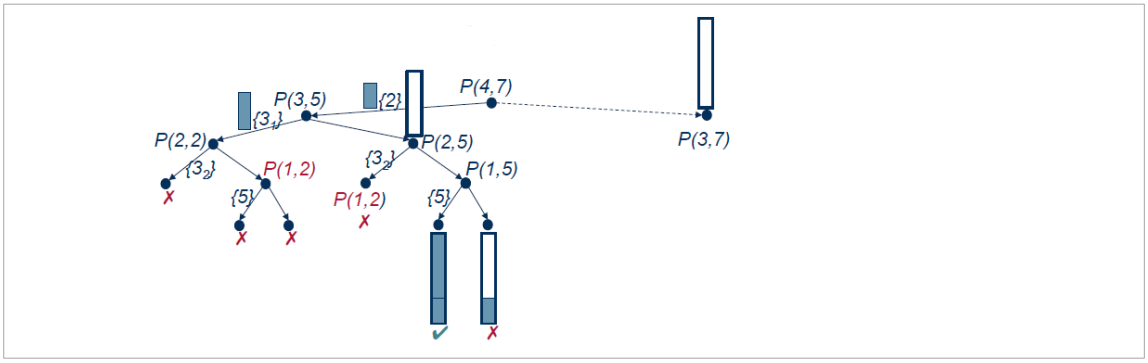
die beide nach Induktionsvoraussetzung berechnet werden können

Zusammenfügung der Ergebnisse:

- Falls für $P(n-1,K)$ eine Lösung existiert, dann ist sie auch Lösung für $P(n,K)$.
- Falls für $P(nn1,K-k_n)$ eine Lösung T existiert, dann ist $T \cup \{k_n\}$ Lösung für $P(n,K)$.
- Andernfalls gibt es keine Lösung von $P(n,K)$.

Lösungsraum rekursiv durchsuchen

- Kaskadenartige Rekursion.
- Die rekursiv betrachteten Teilprobleme werden nur minimal kleiner (1 Element weniger, Rucksack ggf. etwas kleiner).
- Teilproblem P(1,2) ist schon früher gelöst worden!
- Speichern u. Nachschlagen für dynamisches Programmieren?
- Maximal müssten hier (Elementzahl x Rucksackgröße) viele Werte gespeichert werden – das ist hier akzeptabel.

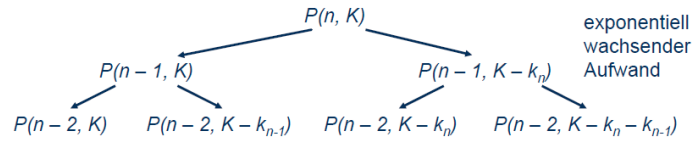


Rekursionsidee – formal

Bei der Reduktion von P(n, K) entstehen 2 Teilprobleme

- Teilprobleme 1: $P(n-1, K)$
- Teilprobleme 2: $P(n-1, K-k_n)$ (nur, falls $K - k_n \geq 0$)

Im Unterschied zu „Teile-und-Herrsche“ sind diese jedoch nicht „deutlich kleiner“ als das Ausgangsproblem.
Es ist daher zu befürchten, dass durch die Reduktion insgesamt sogar mehr zu arbeiten ist!



Ansatzpunkt für dynamisches Programmieren

- Es gibt nur endlich viele Problemgrößen. Schon berechnete Lösungen können wieder verwendet werden, anstatt sie neu zu berechnen.
- Es gibt genau $n \cdot K$ mögliche Problemgrößen und glücklicherweise ist (für realistische K) $n \cdot K \ll 2^n$
- Implementierung: Speicherung der schon berechneten Ergebnisse in einer Matrix der Größe $n \cdot K$

Backtracking

- Wenn aber kein Durchreichen möglich ist oder zu viele Zwischenergebnisse gespeichert werden müssen, dann muss im Allgemeinen das vollständige Durchsuchen also der komplette rekursive Aufrufbaum eben doch sein.
- Rücksetzverfahren (engl. „backtracking“).

Sudoku

9x9-Matrix . Einige Felder vorbesetzt mit Ziffern 1 – 9, einige Felder leer
Ein Sudoku-Rätsel ist gelöst, wenn jede der Ziffern 1 – 9

- in jeder der neun Zeilen, jeder der neun Spalten und in jeder der neun 3x3 Untermatrizen genau einmal vorkommt.

Nach und nach solche Schlussregeln anwenden, um Felder zu finden, bei denen nur genau eine Ziffer möglich ist.
Suchraum hat 9^{81} Möglichkeiten bei naiver Aufzählung viel zu groß zum vollständigen naiven Durchsuchen

Die Schwierigkeit des Rätsels hängt ab

- von der Komplexität der einsetzbaren Schlussregeln,
- von der Anzahl der Felder, bei denen nur eine Ziffer möglich ist,
- vom Vorhandensein von Situationen (= Punkten im Suchraum), in denen es kein Feld (mehr) gibt, in dem eine Ziffer zwingend erschlossen werden kann „diabolische Sudokus“

Programm-Muster:

```
static int[][] backtrack(int[][] state) {
        if (isFinal(state)) {
                return state;
        } else {
                // moegliche Erweiterungen aufzaehlen
                int[] candidates = getExtensions(state);
                for (int i = 0; i < candidates.length; i++) {
                        int c = candidates[i];
                        state = apply(state, c); //z.B. Ziffer setzen
                         if (backtrack(state) != null) { // Rekursion
                                return state;
                        }
                        state = revert(state, c); //z.B. Ziffer loeschen
                }
                return null;
        }
}
```

Rücksetzverfahren, engl. „backtracking“

- Auch beim Rucksack-Problem waren die Entscheidungen entlang des Pfades erforderlich, um das Ergebnis abzulesen und um bei der rekursiven Variante Elemente wieder aus dem Rucksack zu entfernen.
- Achtung: Das Anwenden der Entscheidung (applyChange) auf dem Hinweg wird in der Regel noch gemacht. Das Rücksetzen (revertChange) wird leider beim Programmieren oft vergessen.
- Die Liste der Entscheidungen (im Beispiel die Liste der Felder die man mit Ziffern besetzt hat nennt man auch Ariadne-Faden.
- getExtensions schneidet auch Nachfolger aus dem Suchraum, die sicher keine Lösung sein können. (obwohl noch Leerfelder.)

Münzminimierung

- Problem: Einen Geldbetrag n < 1€ (in Cent) mit möglichst wenig Münzen auszahlen
- Bei Knotenwert ≤0 ist Basisfall erreicht.
- Hoher Verzweigungsgrad führt zu teurer naiver Rekursion.
- Suchraum/rekursiver Aufrufbaum muss beschnitten werden!
- Dynamisches Programmieren einsetzbar, weil es nur maximal 100 mögliche Zwischenlösungen gibt.
- Die Lösung ist bei sog. „kanonischen Münzsystemen" einfacher

Entscheidungsregel eines gierigen Algorithmus

- Problem: Einen Geldbetrag n < 1€ (in Cent) mit möglichst wenig Münzen auszahlen.
- Entscheidungsregel: Wähle größte Münze, die in den zu wechselnden Betrag passt.
- Diese (lokal optimierende) Entscheidungsregel führt
  - zur (global optimalen) minimalen Münzanzahl (Beweis ist schwer),
  - in jedem Schritt des Suchraums genau zu einer Möglichkeit

Münzminimierung am Beispiel

- Problem: Einen Geldbetrag n < 1€ (in Cent) mit möglichst wenig Münzen auszahlen.
- Ein gieriger Algorithmus entscheidet in jedem Schritt richtig (meist durch Wahl der am besten ausschöpfenden/lokal optimalen Lösung) und landet dabei in einem globalen Optimum.
- Perfekte Suchraumverkleinerung/Beschneidung des Aufrufbaums.
- Wenn es einen gierigen Algorithmus gibt, ist er immer besser als Suche/rekursive Aufrufe.
- Oft gibt es keine Entscheidungsregel durch die eine Lösung mit gierigem Algorithmus möglich/optimal ist.

Gegenbeispiele:

- Münzminimierungsalgorithmus beim Münzsystem der Goldmark (1873-1914):
  - 1, 2, 5, 10, 20, 25, 50 Pfennig und 1, 2, 3, 5 Mark.
  - Mit obiger gieriger Entscheidungsregel ergibt sich suboptimal: 40 Pfennig → 25 Pf + 10 Pf + 5 Pf.
- Rucksackproblem:
  - Die Entscheidungsregel „Wähle immer das größte noch verfügbare Element aus, das noch in den Rucksack passt" scheitert bei Rucksackgröße 40 und Elementen wie bei obigen Pf.
- Schach:
  - Die Entscheidungsregel „Wenn eine Figur schlagbar ist, schlage diese. Wenn mehrere Figuren schlagbar sind, dann schlage diejenige mit höchstem Figurenwert" ist nicht immer optimal.
- Klausur:
  - Die Entscheidungsregel „Wähle als nächste zu bearbeitende Aufgabe immer die, die am meisten Punkte bringt." ist nicht immer optimal.

Asymptotische Aufwandsanalyse

Qualitätskriterien für Algorithmen

- Korrektheit
- Hohe Verständlichkeit
  - erhöht die Chance, dass Algorithmus tatsächlich korrekt ist
  - erleichtert Implementierung und spätere Änderungen
- Einfache Implementierbarkeit
  - zeit- und damit kostenbestimmende Faktoren sind das Schreiben und Debuggen
- Geringstmöglicher Aufwand: Bedarf an Ressourcen,insbesondere an Laufzeit und Speicherplatz.

Hinweis:

- Oft „trade-off" zwischen den Kriterien erforderlich, d. h. bestimmte Eigenschaft ist nur erreichbar, wenn man in Kauf nimmt, dass sicheine andere Eigenschaft dabei verschlechtert.
- Beispiel: sehr effizienter Algorithmus könnte sehr schwer verständlich sein

Analyse der Laufzeiten von Algorithmen

- Intuitive Idee: Messen der Rechenzeit eines Programms (also eines implementierten Algorithmus) in Millisekunden.
- Probleme:
  - Größe ist abhängig von vielen Parametern, z. B. verwendeter Rechner, Compiler, Betriebssystem, Programmiertricks usw.
  - Größe nur messbar für Programme, aber nicht für Algorithmen.
- Lösung:
  - Zähle für eine gegebene Eingabe die Anzahl von durchgeführten Elementaroperationen.
  - Laufzeitverhalten eines Algorithmus kann dann durch eine Funktion beschrieben werden, die die Anzahl der durchgeführten Elementaroperationen in Abhängigkeit von der Größe (auch: dem Umfang) der Eingabe darstellt.

Elementaroperationen und nicht elementare Operationen

- Elementaroperationen : Arrayzugriff, arithmetische Operation, Vergleich, Zuweisung
- nicht elementare Operationen: Schleife, Methodenaufruf (insb. Rekursion)
- In AuD: Eine Elementaroperation entspricht genau einer Instruktion des zugrunde liegenden Von-Neumann-Rechners und hat das Kostenmaß 1.
- Laufzeit $\boldsymbol{T}$: Summe der Kosten aller ausgeführten Elementaroperationen abhängig von der Größe (dem Umfang) der Eingabe.

Abstraktion des Verfahrens

- Art der Elementaroperationen wird nicht unterschieden
- Menge aller Eingaben wird aufgeteilt in sog. Komplexitätsklassen (Größe der Eingabe)
- Innerhalb einer Komplexitätsklasse wird abstrahiert von vielen möglichen Eingaben durch
  - Betrachtung von Spezialfällen: bester Fall (best case,) schlimmster Fall (worst case,)
  - Betrachtung des Durchschnittsverhaltens (average case,) als gewichtetes Mittel aller möglichen Eingaben (mit Wahrscheinlichkeiten)
- Betrachtung nur d. Größenordnung der Laufzeitfunktion $T(n)$ d. Weglassen additiver u. multiplikativer Konstanten.

Asymptotischer Aufwand

- Der exakte Aufwand eines Algorithmus (z. B. als Anzahl Rechenschritte bis zur Terminierung oder als verbrauchte Zeit) kann i. d. R. nur schwer als Funktion von Umfang und Eingabewerten allgemein angegeben werden.
- Lineare Faktoren sind für die Theorie uninteressant
- Wichtiger ist ungefähre Größenordnung, mit der der Aufwand in Abhängigkeit vom Umfang der Eingabe wächst.
- Hierbei interessiert vor allem der Aufwand für sehr große Umfänge n

Charakterisierung von Aufwänden mittels O-Kalkül

Um die Größenordnung des Aufwandswachstums festzulegen, wird das asymptotische Verhalten der Aufwandsfunktion zumeist mit einem Repräsentanten einer bestimmten Funktionsklasse $(2^n, n^3, n^2, n\log_2 n, , \log_2 n)$ verglichen.
O-Kalkül erlaubt die Beschreibung solcher Vergleiche zwischen Funktionen.

O-Notation (asympt. obere Schranke), „Groß-O"

- Abschätzung des Programmaufwands/einer Funktion f des Eingabeumfangs durch eine asymptotische obere Schranke g.
- $f(n) \in O(g\ (n)\ )$:
  - $f(n)$ wächst höchstens so schnell wie $g(n)$.
  - Es gibt Konstante c, so dass für große n immer $f(n) \leq c \cdot g(n)$ gilt.
- Definition:
  - $O\ (g(\ n)) = f: \mathbb{N} \rightarrow \mathbb{R}+ \mid \exists n_0 \in \mathbb{N}, c \in \mathbb{R}, c > 0: \forall n \geq n_0\ f(n) \leq c \cdot g(n)$

O-Notation (asympt. obere Schranke), „Groß-O“

- Die Schreibweise $f(n) \in O(g(n))$.
- Achtung: „Gleichung“ $f(n) = O(g(n))$kann nur von links nach
- rechts gelesen werden
- Hinweis: I. d. R. sind $f, g: \mathbb{N} \to \mathbb{N}$ definiert, da das Argument die Größe der Eingabe und der Funktionswert die Anzahl der Elementaroperationen sind; wegen Durchschnittsanalysen ist aber auch $\mathbb{R}+$ möglich.

Beispiele zur O-Notation

$T_1(n) = n^2 + n$
Es gilt: $n^2 + n \leq 2n^2$ ab $n_0 = 0$
In der Menge $O(n^2)$ sind (auch) alle Funktionen, deren Wert ab $n_0 = 0$ den Funktionswert $2n^2$ nicht mehr überschreitet.
Daher gilt: $T_1(n) = n^2 + n \in O(n^2)$

Ω-Notation (asympt. untere Schranke), „Groß-Omega“

$f(n) \in \Omega(g(n))$: $f(n)$ wächst mind. so schnell wie $g(n)$
Definition:

- $\Omega(g(n)) = \{ h(n) \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : 0 \leq c \cdot g(n) \leq h(n) \}$

Beispiele:

- $5 \cdot n^2 + 42 \cdot n + 2 \in \Omega(n^2) \notin \Omega(n^3)$
- $2^n + 5 \cdot n \in \Omega(2^n)$

Θ-Notation (asympt. gebunden), „Groß-Theta“

$f(n) \in \Theta(g(n))$: $f(n)$ wächst ebenso schnell wie $g(n)$
Definition:

- $\Theta(g(n)) = \{ h(n) \mid \exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n \geq n_0 : 0 \leq c_1 \cdot g(n) \leq h(n) \leq c_2 \cdot ggn) \}$

Sandwich-Gedanke

- O ist eine (möglicherweise viel zu große obere Schranke).
- Ω ist eine (möglicherweise viel zu kleine untere Schranke).
- Beide helfen bei der Charakterisierung von Aufwänden nur bedingt.
- Es gilt: $f(n) \in \Theta(g(n))$ genau dann, wenn $f(n) \in O(g(n))$ und $f(n) \in \Omega(g(n))$.
- Aufwandsabschätzungen mit Hilfe von Θ wären ideal, aber leider lassen sich Beweise oft nur schwer führen.
- Daher werden oft in der Literatur nur O-Abschätzungen verwendet, aber mit der Maßgabe, dass die kleinstmögliche obere Schranke angegeben wird Auch in AuD!

Rechenregeln für die O-Notation

Es sei: f(n) ∈ O(r(n)) g(n) ∈ O(s(n)) d > 0 konstant

- Dann gilt:
  - f(n) ± d ∈ O(r(n))
  - d · f(n) ∈ O(r(n))
  - f(n) · g(n) ∈ O(r(n) · s(n))
  - f(n) + g(n) ∈ O(r(n) + s(n)) = O(max(r(n), s(n)))
- Außerdem:
  - Transitivität: x(n) ∈ O(y(n)), y(n) ∈ O(z(n)) ⇒ x(n) ∈ O(z(n))

Beispiele zum O-Kalkül

- $(9n^3 + 17 + 4n) \in O(n^3)$
- $(4n \cdot \sqrt{n}) \in O(n \cdot \sqrt{n})$
- $(120095 + 0.01003n) \in O(n)$
- $(n^5 + 2^n) \in O(2^n)$
  - PS: Wichtig Logarithmusregeln lernen

Laufzeiten der Ablaufstrukturen in O-Notation

- Annahmen:
  - Seien S_1 und S_2 Anweisungen (oder Programmteile) mit den Laufzeiten $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$.
  - $f(n)$ und $g(n)$ seien von 0 verschieden, also z. B. $O(f(n))$ ist mindestens in $O(1)$
- Elementaroperation
  - Laufzeit einer Elementaroperation ist in $O(1)$
  - Folge von c Elementaroperationen hat Laufzeit $c \cdot O(1) = O(1)$

- Sequenz S_1; S_2;
  - Laufzeit: $T(n) = T_1(n) + T_2(n) \in O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max f(n), g(n))$
  - Laufzeit einer Sequenz kann daher i. d. R. mit der Laufzeit der aufwändigsten Operation abgeschätzt werden.
- Schleife
  - Jeder Schleifendurchlauf kann andere Laufzeit haben; all diese Laufzeiten müssen dann summiert werden (s. Rechenregeln).
  - Oft: Laufzeit ist bei jedem Durchlauf gleich, z. B. $O(1)$, dann Multiplikation mit der Anzahl der Iterationen möglich.
  - Sei $T_0 \in O(g(n))$ die Laufzeit für einen Schleifendurchlauf, dann:
  - Fall 1 (Anzahl der Schleifendurchläufe hängt nicht von n ab, ist also Konstante c):
  - Laufzeit: $T(n) \in O(1) + c \cdot O(g(n)) = O(g(n))$, falls $c > 0$
  - Fall 2 (Anzahl der Schleifendurchläufe ist $O(f(n))$:
  - Laufzeit: $T(n) \in O(f(n)) * O(g(n)) = O(f(n) * g(n))$
- Bedingte Anweisung if (B) {S_1;} else {S_2;}
  - Wenn nicht klar ist, welcher Zweig gewählt wird:
  - Laufzeit: $t(n) \in O(1) + O(f(n)) + O(g(n)) = O(f(n)+g(n)) = O(\max(f(n),g(n)))$
- Methodenaufrufe
  - Separate Analyse des Aufwands der Methode.
  - Laufzeit wird an der Stelle des Methodenaufrufs eingesetzt.
  - Sonderfall rekursive Funktionen: Laufzeit muss durch sogenannte Rekursionsgleichung beschrieben werden, die dann zu lösen ist.

Beispiel: Horner-Schema

Ziel: Funktionswert $P_n(x)$ eines Polynoms $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$

Algorithmus: Horner-Schema

```
p = a[n];
for (int i = n - 1; i >= 0; i--) {
        p = p * x + a[i];
}
```

- Aufwand: Schleife wird n-mal durchlaufen, jeder Durchlauf hat konstanten Zeitbedarf, also: Zeitaufwand ist $O(n)$

Beispiel: Palindrom-Test (iterative Variante)

Gegeben: Zeichenfolge $A = a_0, a_1, \ldots, a_{n-1}$

Aufgabe: Stelle fest, ob A ein Palindrom ist, d. h. ob gilt: $\forall i \in \left\{0, \ldots, \frac{n}{2}\right\}: a_i = a_{n-1-i}$

```
static boolean istPalindrom(char[] a) {
        int n = a.length;
        for (int i = 0; i <= n/2; i++) {
                if (a[i] != a[n - 1 - i]) {
                        return false;
                }
        }
        return true;
}
```

- Schleife wird höchstens n/2-mal durchlaufen
- Für jeden Durchlauf kann ein maximaler, aber konstanter oberer Zeitbedarf angegeben werden.
- Zeitaufwand ist in $O(n)$.

Beispiel: MinSort2

```
class MinSort2 {

        static int minSuche2(int[] r, int s) {
                // gibt den Index eines Elements von r mit kleinstem
                // Wert im Bereich ab Index s zurueck
                int wmerker = r[s]; // merkt den kleinsten Wert
                int imerker = s; // merkt einen Index zum kleinsten Wert
                int i = s;
                int n = r.length;
                while (i < n){
                        if (r[i] < wmerker) {
                                wmerker = r[i];
                                imerker = i;
                        }
                        i = i + 1;
                }
                return imerker;
        }
        public static void main(String[] args) {
                int[] a = {11, 7, 8, 3, 15, 13, 9, 19, 18, 10, 4};
                int n = a.length;
                int i = 0;
                int k; // speichert den Minimumindex
                while (i < n - 1) {
                        k = minSuche2(a, i);
                        int merker = a[i];
                        a[i] = a[k];
                        a[k] = merker;
                        i = i + 1;
                }
                i = 0;
                while (i < n) {
                        System.out.println(a[i]);
                        i = i + 1;
                }
        }
}
```

- minSuche2 = O(n)→MinSort2 = O (n^2)

Laufzeiten für unterschiedliche Eingaben am Beispiel

gegeben: Java-Funktion, die testet, ob eine Integer-Zahl c im Integer-Array s enthalten ist (hier: sog. lineare Suche)

```
static boolean enthaelt(int[] s, int c) {
        boolean gefunden = false;
        for (int i = 0; i < s.length; i++) {
                if (s[i] == c) { gefunden = true; }
        }
        return gefunden;
}
```

Analyse der linearen Suche im unsortierten Array verbesserte Version

Abbruch der Schleife, wenn Element gefunden

```
static boolean enthaelt2(int[] s, int c) {
        boolean gefunden = false;
        for (int i = 0; !gefunden  && (i<s.length); i++) {
                if (s[i] == c) {
                        gefunden = true;
                }
        }
        return gefunden;
}
```

Binäre Suche im sortierten Array

```
static boolean enthaelt3(int[] s, int u, int o, int c) {
        if (u > o) {
                return false;
        } else {
                int m = (u + o) / 2;
                if (s[m] == c) {
                        return true;
                } else {
                        if (s[m] < c) {
                                return enthaelt3(s, m + 1, o, c);
                        } else {
                                return enthaelt3(s, u, m - 1, c);
                        }
                }
        }
}
```

- $T(n)$ sei die worstcase-Laufzeit von enthaelt3, angesetzt auf ein Array mit n Elementen
- Rekursionsgleichungen für Laufzeit:
  - $T(0) = a_{kr}$
  - $T(n) = a_r + T(\frac{n}{2})$
- a_{kr}, a_r Konstanten: a_{kr} beschreibe die Laufzeit (obere Schranke für Anzahl der Elementaroperationen), falls kein rekursiver Aufruf erfolgt und a_r die Laufzeit bis zum rekursiven Aufruf im anderen Fall
- Wiederholtes Einsetzen der Rekursionsgleichungen liefert: $O(\log n)$

Einschub: Formeln zur Auflösung von Rekursionsgleichungen

$$1 + 2 + 3 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

$$1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1$$

$$1 + 4 + 9 + \ldots + n^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$$

Lineare vs. binäre Suche

- Binäre Suche ist also deutlich besser als lineare Suche.
- Erkenntnis: verschiedene Algorithmen zur selben Problemspezifikation können bei gleicher Eingabe unterschiedlichen Aufwand haben.

Aufwandswachstum

- Die Größe der Menge der Eingabedaten für einen Algorithmus kann stark variieren.
- Das Verhalten eines Algorithmus sollte durch die Größe der Eingabedaten nur moderat beeinflusst werden:
  - Z. B. sollte ein Sortieralgorithmus 10, 100, 1000, 10000, ... Datenelemente sortieren können, ohne bei mehr Elementen bedeutend mehr Ressourcen (Zeit, Speicherplatz) zu benötigen.
- Bei der Entwicklung eines Algorithmus sollte man also danach streben, dass der Zeit- und Speicheraufwand nur moderat mit der Größe der Eingabedaten wächst.

Typische Laufzeitklassen und zugehörige Algorithmen

	Sprechweise	typische Algorithmen
$O(1)$	konstant	
$O(\log n)$	logarithmisch	Suchen auf einer sortierten Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \log n)$		gute Sortierverfahren, z. B. Sortieren durch Teilen und Verschmelzen, Skyline-Problem
...		
$O(n^2)$	quadratisch	primitive Sortierverfahren, z. B. Sortieren durch wiederholte Auswahl des Minimums (`minSort`)
$O(n^k)$, $k \geq 2$	polynomiell (viele Klassen)	
...		
$O(b^n)$, $b > 1$	exponentiell (viele Klassen)	Backtracking-Algorithmen

Einschub: Nützliche Formeln zum Logarithmus

a) $\log_a(b \cdot x) = \log_a b + \log_a x$

b) $\frac{\log_a x}{\log_a b} = \log_a x - \log_a b$

c) $\log_a x^b = b \cdot \log_a x$

d) $\log_a x = \frac{\log_b x}{\log_b a} \left(= \frac{\ln x}{\ln a}\right)$

e) $\log_a x = \frac{1}{\log_x a}$

f) $a^{\log_a x} = x$

g) $b^{\log_a x} = x^{\log_a b}$

Polynomieller und exponentieller Zeitaufwand

- Wichtige Beobachtung:
  - Exponentielle Funktionen wachsen auch bei kleiner Basis für große n stärker als polynomielle Funktionen mit großen Exponenten.
  - Für $c > 0$ und $a > 1$ gilt: $n^c \in O(a^n)$
- Konsequenzen:
  - Algorithmen mit exponentiellem Zeitaufwand liefern nur bei z. T. sehr kleinen Eingabegrößen noch in akzeptabler Zeit Ergebnisse.
  - Algorithmen mit polynomiellem Zeitaufwand gelten noch als praktisch handhabbar, man sucht deshalb immer nach solchen Lösungen.
  - Es gibt eine Reihe von z. T. sehr praxisrelevanten Problemen, für die man bis heute keine polynomiellen Algorithmen zur Bestimmung der optimalen Lösung gefunden hat und auch noch nicht sicher sagen kann, ob diese überhaupt existieren (z. B. sog. Rucksack-Problem).

Objektorientierte Modellierung und Programmierung

Klasse und Objekt

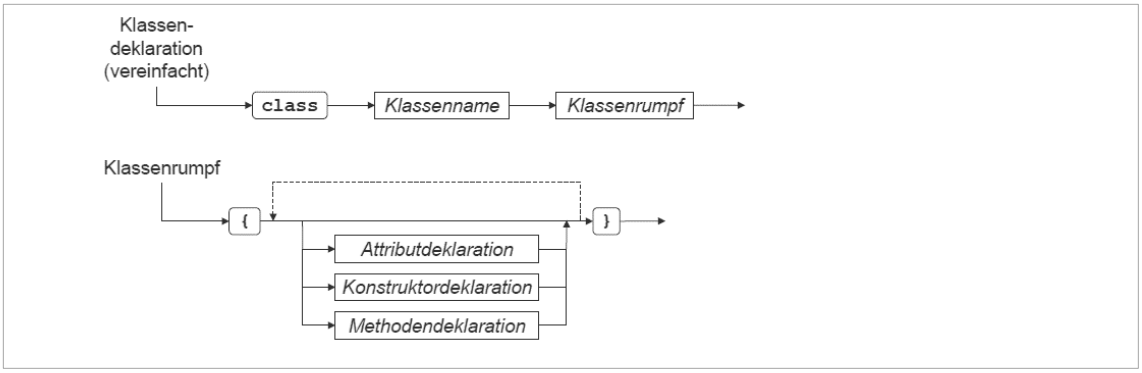
- Klasse (auch: Objekttyp, engl.: class, object type): Konzept der objektorientierten Modellierung und Programmierung, mit dem neue Datentypen definiert werden können.
- Die Deklaration einer Klasse stellt damit eine Art Bauanleitung oder Schablone für sog. Objekte (auch: Exemplare, Instanzen) dieser Klasse dar.
- Konkret wird in der Klassendeklaration festgelegt:
  - aus welchen Datenkomponenten (sog. Attributen), ein Objekt besteht,
  - welche Operationen (sog. Methoden), z. B. zur Manipulation der zugehörigen Attributwerte, zur Verfügung stehen und
  - welche sog. Konstruktoren zur Erzeugung von Objekten (man sagt auch: Instanziierung) verwendet werden können

Beispiel: Mitglieder einer Bibliothek

```
class BibMitglied {
        // --- Attribute ---
        String name;
        String privAnschrift;
        int mitgliedsNr;
        // --- Konstruktor ---
        BibMitglied(String n, String pA, int mNr) {
                name = n;
                privAnschrift = pA;
                mitgliedsNr = mNr;
        }
        // --- Methoden ---
        // --- Getter und Setter ---
        String getName() {
                return name;
        }
        void setName(String n) {
                name = n;
        }
        String getPrivAnschrift() {
                return privAnschrift;
        }
        void setPrivAnschrift(String pA) {
                privAnschrift = pA;
        }
        int getMitgliedsNr() {
                return mitgliedsNr;
        }
        void setMitgliedsNr(int mNr) {
                mitgliedsNr = mNr;
        }
        // andere Methoden
        void print() {
                System.out.println("Mitgliedsnr.: " + mitgliedsNr);
                System.out.println("Name: " + name);
                System.out.println("Anschrift: " + privAnschrift);
        }
}
```

Deklaration von Klassen

- Klassendeklaration legt den Namen einer Klasse sowie ihre Attribute, Konstruktoren und Methoden fest und definiert damit einen neuen Datentyp.
- Sie hat folgenden Aufbau:



Deklaration von Attributen und Methoden

- Attribute sind die Datenkomponenten, aus denen die Objekte einer Klasse bestehen.
  - Deklaration und Benennung erfolgen analog zu Variablen.
  - Erlaubt sind Attribute primitiven und zusammengesetzten Typs
  - Beispiele: int mitgliedsNr; Datum geburtsDatum; Buch[] entlieheneBuecher;
- Methoden (Deklaration und Benennung) sind Ihnen schon bekannt.
  - Im Methodenrumpf kann auf die Attribute eines Objekts zugegriffen werden.
  - Später: Unterschied zwischen Methoden mit/ohne static

Benennung von Klassen

- Klassennamen
  - sollten selbsterklärend sein und die Lesbarkeit eines Programms fördern, z. B. Spielfeld
  - sollten Substantive sein und mit einem Großbuchstaben beginnen

Deklaration von Konstruktoren

- sind Operationen, die es ermöglichen, Objekte einer Klasse zu instanziieren, d. h. anzulegen o. zu erzeugen.
- Alle Konstruktoren einer Klasse haben denselben Namen, nämlich den der Klasse; zusätzlich können sie Parameter haben.
- Ein Rückgabetyp wird nicht angegeben und sie enthalten damit nur optional eine return-Anweisung.
- Beispiel:

```
BibMitglied(String n, String pA, int mNr) {
        name = n;
        anschrift = pA;
        mitgliedsNummer = mNr;
}
```

Man unterscheidet zwischen zwei Arten von Konstruktoren

- explizite Konstruktoren
  - Sie werden bei der Klassendeklaration angegeben und i. d. R. dazu genutzt, die Attributwerte des zu erzeugenden Objekts auf für den Anwendungszweck sinnvolle Startwerte zu setzen.
  - Explizite Konstruktoren können Parameter haben.
  - Eine Klasse kann mehrere explizite Konstruktoren haben, die aber unterschiedliche Parameterlisten haben müssen (entscheidend sind dabei unterschiedliche Typen und nicht nur unterschiedliche Parameterbezeichner!)
- impliziter Konstruktor (auch: Standard-Konstruktor, Default-Konstruktor)
  - Wird kein expliziter Konstruktor bei der Klassendeklaration angegeben, so legt der Übersetzer automatisch einen impliziten Konstruktor an.
  - Ein impliziter Konstruktor hat immer die Form <Klassenname>() { } und damit keine Parameter

Objektorientiertes Programm in Java (vom Typ Applikation)

- ...besteht aus Folge von Klassendeklarationen, die jeweils Attribut-, Konstruktor- und Methodendeklarationen enthalten.
- In einer der Klassen muss die Methode main deklariert sein, mit der die Ausführung des Gesamtprogramms startet.
- Klasse, die die Methode main deklariert, nimmt i. d. R. Sonderrolle ein, nämlich die der Spezifikation des Hauptprogramms und nicht die der Deklaration des Datentyps einer Klasse von Objekten.
- In der Regel beginnt dort die Erzeugung der für die Problemlösung relevanten Objekte anhand der Klassendeklarationen.
- Diese Objekte verwalten eigene Daten (Werte der Attribute) und sie reagieren auf Nachrichten (Aufrufe von Methoden, die in der Klasse des Objekts deklariert wurden).
- Problemlösung durch schrittweise Objekterzeugung, -verknüpfung und -kommunikation.

Objekt

- Objekt ist Datenstruktur, deren Datentyp (Struktur, Verhalten) durch eine Klasse festgelegt ist.
- Objekt ist konkrete Ausprägung einer Klasse, deshalb auch Instanz (engl. instance) genannt.
- Klassendeklaration legt fest, welche Attribute und Methoden für ein Objekt der Klasse zur Verfügung stehen und in welcher Form eine Instanziierung stattfinden kann.
- Während eine Klasse die Attribute i. d. R. nur deklariert, werden bei einem Objekt die Attribute durch den verwendeten Konstruktor oder durch Methodenaufrufe mit Attributwerten belegt.
- Die aktuelle Belegung der Attribute eines Objekts mit Attributwerten definiert den aktuellen Zustand des Objekts.

Objektvariablen

- Attribute von primitivem Typ sind „Behälter im Objekt“. Sie verhalten sich wie die Ihnen bekannten Variablen.
- Attribute vom Typ einer Klasse (sog. Objektvariablen) können einen Verweis/eine Referenz auf ein Objekt enthalten.
    - Bei der Deklaration einer Objektvariable wird eine Variable angelegt, die einen Verweis (auch: Referenz, Zeiger) auf ein Objekt der zugehörigen Klasse aufnehmen kann (deshalb wird solche Variable auch als Referenzvariable bezeichnet; vgl. „Referenzsemantik“)
    - Es wird bei der Deklaration noch keine Objektinstanz erzeugt.
    - Wert der neu angelegten Variable ist eine sog. Null-Referenz
        - zeigt an, dass die Objektvariable (noch) nicht auf ein Objekt verweist.
        - Schlüsselwort null (z. B. für Vergleiche und Wertzuweisungen).

Instanziierung von Objekten

- Instanziierung:
    - bezeichnet die Erstellung eines neuen Objekts einer Klasse.
    - Objekt wird erzeugt, d. h. der entsprechende Platz im Speicher reserviert.
    - Verweis (Referenz, Zeiger) auf dieses Objekt bzw. auf den Speicherbereich wird zurückgegeben.
- Form: Schlüsselwort new gefolgt von Konstruktoraufruf.
    - Beispiel: new BibMitglied("Petra Buch-Wurm", "Alte Gasse 1, 86807 Buchloe",12345);
- erfolgt durch den new-Operator, gefolgt von Angabe eines Konstruktors aus der Klassendeklaration des Objekts mit zugeh. Argumenten
- Objektvariablen können bereits bei ihrer Deklaration mit einem Verweis auf ein Objekt initialisiert werden

Wertzuweisung bei Objektvariablen

- Dieselbe Form wie für Variablen primitiven Datentyps.
- Auf der linken Seite des Zuweisungsoperators „=“ steht Objektvariable, auf der rechten Seite steht ein Ausdruck, der einen mit der Objektvariablen kompatiblen Klassentyp hat
- Deklaration und erste Wertzuweisung kann wie auch bei primitiven Datentypen kombiniert werden
- Initialisierung kann aber auch erst später erfolgen

Annahme: es wird eine weitere Objektvariable deklariert : BibMitglied bm2;

- Nun: Wertzuweisung bm2 = bm1;
- Wirkung:
    - Der in bm1 gespeicherte Wert (Referenz) wird bm2 zugewiesen.
    - bm1 und bm2 verweisen damit auf dasselbe Objekt.
- Wichtig: bei Zuweisung an Referenzvariablen wird der Verweis kopiert, nicht aber das Objekt, auf das verwiesen wird.

Vergleich Wertsemantik versus Referenzsemantik

- Variablen primitiven Typs
    - Mit Variablen einfachen Typs wird bei deren Vereinbarung zugleich ein Behälter für den Wert bereitgestellt.
    - Mit Angabe des Namens wird unmittelbar der in der zugeordneten Speicherzelle abgelegte Wert ausgelesen (sog. Wertsemantik).
- Variablen zusammengesetzten Typs/Objektvariablen
    - Bei solchen Variablen wird nur ein Behälter für einen Zeiger (auch: Referenz, Verweis) auf ein Objekt des entsprechenden Typs angelegt (sog. Referenzsemantik).
    - Der eigentliche Speicherplatz für das eigentliche Objekt muss dann noch angelegt werden und ist somit nicht fest angebunden.
- Wirkungen:
    - Zuweisungen bei Wertsemantik

```
int i1 = 1;
int i2;
i2 = i1; // i2: 1
 i1 = 2; // i1: 2; i2: 1
```

    - Zuweisungen bei Referenzsemantik

```
int[] a1 = {1, 2, 3};
int[] a2;
a2 = a1;   // a2 zeigt jetzt auch auf das durch a1
           // referenzierte Objekt
a1[2] = 0; // a1: 1, 2, 0 und a2: 1, 2, 0
```

    - Faustregel: bei Objekten, die i. d. R. mit new erzeugt werden, gilt die Referenzsemantik

Vorbesetzte Attribute

Beispiel: Uhrenklasse, deren Objekte eine übergebene Minutenzahl in Stunden und Minuten umrechnen

```
class Uhr {
        // Attribute
        long minuten;
        long stunden;
        TimeZone zeitzone = new TimeZone("UTC");

        // expliziter Konstruktor
        Uhr(long min) {
                stunden = min / 60; // Division ohne Rest (Ganzzahl)
                minuten = min % 60; // Modulo-Operation
        }
}
```

- Attribute werden bei Objekterzeugung mit dem angegebenen Wert initialisiert (analog für primitive Typen)
- TimeZone sei eine an anderer Stelle deklarierte Klasse.
- UTC: Coordinated Universal Time, koordinierte Weltzeit, wird zur Berechnung der Mitteleuropäischen Zeit verwendet

Referenzkonstanten

Beispiel: Uhrenklasse, deren Objekte eine übergebene Minutenzahl in Stunden und Minuten umrechnen

```
class Uhr {
        // Attribute
        long minuten;
        long stunden;
        TimeZone zeitzone = new TimeZone("UTC");
        final Date herstellungsdatum = new Date();

        // expliziter Konstruktor
        Uhr(long min) {
                stunden = min / 60; // Division ohne Rest (Ganzzahl)
                minuten = min % 60; // Modulo-Operation
        }
}
```

- Referenzkonstanten fixieren den Verweis.
- das Objekt, auf das sie verweisen, kann geändert werden (bei bleibender Identität).
- Semantik: es gibt genau ein unveränderliches Herstellungsdatum für eine Uhr

Methodenaufruf

Hier können zwei Fälle unterschieden werden:

- Fall 1 (Zugriff von Außen, nicht immer erlaubt, vgl. Sichtbarkeits-modifikatoren, folgen noch): Deklaration einer Objektvariablen, Instanziierung eines Objekts und dann Aufruf einer Methode (d. Objekts).
  - Aufruf einer Methode geschieht für ein Objekt der Klasse, in der auch die Methode deklariert ist, durch Aufruf der Methode mit vorangestellter Angabe des Objekts getrennt durch einen Punkt (sog. Punktnotation)
  - Beispiel: p.length();
- Fall 2 (Zugriff von Innen): Aufruf einer Methode innerhalb derselben oder einer anderen Methode der Klasse, in der sie deklariert ist.
- Aufruf erfolgt wie Funktionsaufruf (also ohne Angabe eines Objekts und Punkt „.“) oder mittels sog. this-Referenz (folgt auch noch).
- Beispiel: weitere Methode der Klasse Vec3 void printLength() { System.out.println("Länge: " + length()); }

Zugriff auf Attribute eines Objekts

Zwei Fälle (analog zu Methodenaufrufen):

- Fall 1 (Zugriff von Außen, nicht immer erlaubt, vgl. Sichtbarkeitsmodifikatoren, folgen noch):
  - über <objektname>.<attributname>
- Fall 2 (Zugriff von Innen): über den Attributnamen

Selbstreferenz: this-Referenz

- Jedes Objekt führt eine Referenz auf sich selbst mit sich.
- Auf diese kann man mit dem Schlüsselwort this im Konstruktor und in Methoden zugreifen.
- Damit ist ein alternativer Zugriff eines Objekts auf die eigenen Attribute und Methoden möglich.

Selbstreferenz: this-Referenz

Das Schlüsselwort this ermöglicht den expliziten Bezug auf Attribute oder Methoden des Objekts, in dessen Klasse es verwendet wird.

```
// --- Konstruktor ---
BibMitglied(String name, String privAnschrift, int mitgliedsNr) {
        this.name = name;
        this.privAnschrift = privAnschrift;
        this.mitgliedsNr = mitgliedsNr;
}
```

Vergleiche bei Referenzvariablen

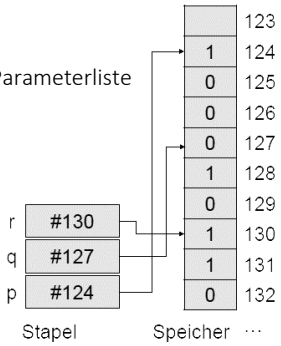
Vergleich der Referenzvariablen prüft die Identität und nicht die Bestandteile (nur die Referenzen)

Objektreferenz als Rückgabewert einer Methode

Verschiedene Methoden derselben Klasse dürfen denselben Namen verwenden, sofern sie sich in der Parameterliste unterscheiden).

Programmstapel und Objektvariablen

- Objekt wird im Speicher angelegt.
- Referenz auf das Objekt liegt auf dem Stapel, nicht aber das Objekt selbst
- Am Ende eines Blocks werden Objekte im Speicher nicht automatisch gelöscht!



Einschub: Speicherbereinigung,

- Frage: was geschieht mit Objekten, die nicht mehr benötigt werden?
- andere Sprachen: Destruktor
  - Spezielle Methode, die ein Objekt entfernt, d. h. den von ihm belegten Speicherplatz wieder frei gibt.
  - Fehleranfällig, wenn dieser von Hand im Programm aufgerufen werden muss.
- Java: automatische Speicherbereinigung („garbage collection“)
  - Speicherbereiniger ist ein spezieller Teil der Laufzeitumgebung (virtuellen Maschine), der die Objektgraphen im Speicher durchsucht.
  - Wird ein Objekt gefunden, das nicht mehr referenziert wird, wird dieses gelöscht und der belegte Speicher frei gegeben.
  - Vorteil: Programmierer/-in muss sich nicht darum kümmern.

Zählen der instanziierten Objekte einer Klasse

- Beobachtung: die Vergabe von Mitgliedsnummern von Außen ist möglich, erfordert aber eine Instanz, die sich um die Zuordnung kümmert.
- Frage: Kann man das auch innerhalb der Klasse BibMitglied selbst automatisieren?
- Strategie: es müsste gezählt werden, wie viele Objekte der Klasse BibMitglied bereits erzeugt wurden.

Klassenattribut

```
class BibMitglied {
        // Klassenattribut
        static int mitgliederZaehler = 0;

        // --- Attribute ---
        String name;
        String privAnschrift;
        int mitgliedsNr;
        // --- Konstruktor ---
        BibMitglied(String n, String pA) {
                name = n;
                privAnschrift = pA;
                mitgliedsNr = BibMitglied.mitgliederZaehler++;
        }

        // --- Methoden ---
        // unveraendert
}
```

Klassenattribut

- Für jedes „normale“ Attribut gibt es Speicherplatz pro Objekt.
- Klassenattribut (auch: statisches Attribut) ist Sonderfall, bei dem sich alle Objekte der Klasse denselben Attributwert teilen.
- Schreiboperationen in einem Objekt wirken sich damit auf alle anderen Objekte aus.
- Deklaration hat die Form:
  - static <Datentyp> <Klassenattributname> (= <Anfangswert>);
- Verwendung:
  - Der Zugriff auf „normale“ Attribute von außen erfolgte in der Form: <objektname>.<attributname>
  - Der Zugriff auf Klassenattribute von außen hat die Form: <klassenname>.<attributname> oder <objektname>.<attnme>
  - Von innen jeweils in der Form: <attributname>

Klassenmethode

```
// --- Klassenmethoden ---
static void setMitgliederZaehler(int mz) {
        BibMitglied.mitgliederZaehler = mz;
}
static int getMitgliederZaehler() {
        return BibMitglied.mitgliederZaehler;
}
// ...
```

- Klassenmethoden: auch ohne instanziiertes Objekt verwendbar
- Getter/Setter wären auch als nicht-statische Methoden erlaubt, dann aber nur verwendbar, wenn Objekt instanziiert wurde

```
BibMitglied.setMitgliederZaehler(1000);
BibMitglied bm1 = new BibMitglied( "Petra Buch-Wurm", "Alte Gasse 1, 86807 Buchloe");
BibMitglied bm2 = new BibMitglied( "Paul-Les Eratte", "Schillerstr. 2, 89290 Buch");
BibMitglied bm3 = new BibMitglied( "David B.O. Okworm", "Buchenring 3, 76297 Büchig");
System.out.println(BibMitglied.getMitgliederZaehler());
...
```

Klassenmethoden (engl. class method)

- Methoden werden im Allg. von den Objekten einer Klasse ausgeführt.
- Klassenmethoden (auch: Klassenoperationen, statische Operationen/Methoden) sind einer Klasse und nicht einem Objekt zugeordnet und bieten damit die Möglichkeit, Methoden direkt auf einer Klasse selbst auszuführen.
- In diesem Fall ist keine Objekterzeugung vor dem Methodenaufruf erforderlich.
- Deklaration: wie normale Methode nur mit vorangestelltem Schlüsselwort static
- Beispiele:
  - die statische Methode main (Programmstart),
  - alle Methoden der vergangenen Lehreinheiten.
- Frage: Wann und wozu verwendet man Klassenmethoden?
  - als Getter und Setter für Klassenattribute
  - wenn für die Methode kein Objekt der Klasse benötigt wird oder generell zu einer Klasse keine Objekte instanziiert werden sollen, z. B. Java-Bibliotheksklasse Math, die mathematische Konstanten und Funktionen bereitstellt
    - Bibliotheksklassen sind bereits programmiert und können direkt verwendet werden
  - so sparsam wie möglich, da eigentlich nicht der objektorientierten Philosophie entsprechend
- Hinweis: Klassenmethoden können nicht auf Instanz- variablen (z. B. Attribute, Konstanten), sondern nur auf Klassen-variablen (z. B. Klassenattribute, Klassenkonstanten) operieren.
- Erlaubt ist aber:

```
static Vec3 addAndIncX(Vec3 v1, Vec3 v2) {
Vec3 r = v1.add2(v2);
r.x++;
return r;
}
```

- Es werden Objektreferenzen v1, v2 als Parameter übergeben.
- Für die übergebenen Objekte dürfen (soweit erlaubt) deren Instanzmethoden (z. B. add2) aufgerufen und auf deren Attribute (z. B. x) werden.
- Es ist nicht erforderlich, ein Objekt der Klasse Vec3 vor dem Aufruf von addAndIncX zu erzeugen

Sichtbarkeit von Information

- Vorteil der objektorientierten Sichtweise:
  - Es reicht, die Signaturen/Schnittstellen der Methoden eines Objekts zu kennen, um mit diesem „umgehen“ zu können.
  - Implementierungsdetails sind für die Anwendung einer Methode nicht relevant.
  - Man sagt, Implementierungsdetails sind verdeckt bzw. geheim.
- Oft (insb. in größeren Software-Projekten):
  - nur wichtig, was ein Objekt leistet,
  - nicht aber, wie es realisiert ist.
  - Konsequenz: es werden Programmiertechniken benötigt, mit denen man den Zugriff auf Attribute und Methoden von Objekten einschränken kann.
  - Die Anwendung dieser Techniken fördert die arbeitsteilige Entwicklung in Software-Projekten, da nicht jede/r Entwickler/-in den gesamten Programmtext verstehen muss.

Datenkapselung

- bezeichnet das Verbergen von Programmierdetails vor dem direkten Zugriff von Außen.
- Direkter Zugriff auf interne Datenstruktur wird unterbunden und erfolgt stattdessen über wohldefinierte Schnittstellen („Black-Box-Modell“).
- Objektorientierte Programmierung:
  - Kontrollierter Zugriff auf Attribute und Methoden von Objekten.
  - Objekte können den internen Zustand anderer Objekte nicht in unerwarteter Weise lesen oder verändern.
  - Klasse hat definierte Schnittstelle, die festlegt, auf welche Weise mit ihren Objekten umgegangen werden kann.
  - Vom Innenleben der Klasse (z. B. verwendete Algorithmen) soll Anwender möglichst wenig wissen (sog. Geheimnisprinzip).
  - Implementierung einzelner Methoden oder ganzer Klassen kann so ausgetauscht und verbessert werden, ohne dass das Programm an anderen Stellen geändert werden muss.

Sichtbarkeitsmodifikatoren private und public

- ... beeinflussen die Sichtbarkeit, d. h. den Zugriff z. B. auf Attribute und Methoden von Objekten.
- ... werden einer Deklaration vorangestellt, der damit ein Sichtbarkeitsgrad zugeordnet wird.
- private:
  - bewirkt bei einer Attributdeklaration, dass auf das Attribut nur innerhalb der Klassendeklaration lesend und schreibend zugegriffen werden kann, aber nicht von außerhalb.
  - bewirkt bei einer Methodendeklaration, dass die Methode nur innerhalb der Klassendeklaration aufgerufen werden kann, aber nicht von außerhalb.
  - Es können auch Konstruktoren oder ganze Klassen (sinnvoll z. B. in sog. Paketen) als private deklariert werden.
- public:
  - erlaubt vollen Zugriff von innerhalb und außerhalb der Klasse (wenn die Klasse nicht selbst als private deklariert wurde).

	Eigenschaft ist sichtbar aus/von			
Modifikator	eigener Klasse	selben Paket	Unterklasse	überall
`public`	ja	ja	ja	ja
`protected`	ja	ja	ja	nein
ohne/paketsichtbar	ja	ja	nein	nein
`private`	ja	nein	nein	nein

Bemerkungen

- Instanzvariablen sollten nur mit wirklich gutem Grund öffentlich zugänglich sein.
- Besser: öffentliche Getter- und Setter-Methoden verwenden, die den Wert der Instanzvariablen setzen bzw. zurückliefern.
- Vorteile: ein Setter kann gegenüber der reinen Zuweisung bspw. überprüfen, ob ein übergebener Wert innerhalb eines gewünschten Wertebereichs liegt.

Objektorientierte Modellierung und Programmierung (Teil 2)

Phasen der objektorientierten Software-Entwicklung

Objektorientierte Analyse (OOA)

- Ziel: allgemeines Systemmodell erstellen ohne Implementierungsdetails
- Erfassung der Anforderungen und Beschreibung, was das zu entwickelnde System machen soll (führt zu sog. Pflichtenheft)
- Erstellung eines objektorientierten Analysemodells (OOA-Modell) unter Verwendung von UML:
  - statisches Modell: Beschreibung der Struktur des Systems, insb. Mit Klassendiagrammen (Klassennamen, Attributnamen, Methodennamen, Klassenbeziehungen); Objektkonfigurationen mit Objektdiagrammen
  - dynamisches Modell: Anwendungsfalldiagramm

Objektorientierter Entwurf (OOE), auch: obj.-orient. Design (OOD)

- Erstellung eines objektorientierten Entwurfsmodells (OOE/D-Modell)
  - statisches Modell: Verfeinerung des OOA-Klassendiagramms mit Implementierungsdetails (z. B. Spezifikation von Attributen, Methoden, Klassenbeziehungen, Ergänzung weiterer für die Implementierung erforderlicher Klassen) zum sog. OOE-Klassendiagramm
  - dynamisches Modell: „zeitliches Verhalten“ des Systems, insb. mit Interaktions- u. Zustandsdiagrammen
  - Planung und Beschreibung, wie das geplante System die Anforderungen realisiert

Objektorientierte Programmierung (OOP)

- im engeren Sinne: systematische Überführung der zuvor entwickelten UML-Diagramme in Quelltext der verwendeten objektorientierten Programmiersprache (teilweise softwaregestützt möglich)
- im weiteren Sinne: gesamter Prozess

Motivation für die Verwendung von UML in allen Phasen

- Beschreibung verschiedener Ausbaustufen des geplanten Systems
- Ideen können auf der Konzeptebene diskutiert werden
- Idee wird zum Modell, Modell wird zum Programm
- Industriestandard
- bewährt auch im praktischen Einsatz interdisziplinärer Projektteams

Beispiel: Bibliotheksverwaltung

Vom Anwender zu Anwendungsfällen

1. Schritt: Beschreibung der Nutzer und ihrer Anforderungen

- Analyse aus Anwendersicht:
  - Akteure (engl. actor) identifizieren
    - Akteur: Anwender (Mensch, Maschine, Programm) des zu entwerfenden Systems in einer bestimmten Rolle
    - tritt mit dem zu entwerfenden System in Interaktion
    - hat Anforderungen an dieses System
  - Anwendungsfälle (engl. use cases) identifizieren und beschreiben
    - Anwendungsfall: Aufgabe, die ein Akteur mit Hilfe des zu entwerfenden Systems lösen will / muss
    - Identifikation durch Textanalyse der Aufgabenstellung im Hinblick auf Zeitwortphrasen (z. B. … leiht Buch aus …)
    - detaillierte textuelle Beschreibung der Abläufe (sog. Szenarien)

UML-Anwendungsfalldiagramm (engl. use case diagram)

- Modellierungselemente: Akteure als Strichfiguren, Anwendungsfälle als Ovale, Beteiligung eines Akteurs an einem Anwendungsfall als Linie.

2. Schritt: Klassenkandidaten identifizieren

- Ausgangspunkt dafür:
  - ausführliche, textuelle Beschreibung der Aufgabenstellung
  - ausführliche, textuelle Beschreibung der Anwendungsfälle
- Textanalyse („Abbot‘s noun approach“)
- Substantive im Text:
  - Kandidaten für Klassen, wenn damit etwas Systemrelevantes beschrieben wird, über das mehrere Informationen zu speichern sind, z. B. Mitglied, Buch
  - Kandidaten für Attribute, wenn damit nur einzelner Wert beschrieben wird, z. B. Name, Mitgliedsnummer
- Verben im Text:
  - Kandidaten für Methoden, wenn damit etwas Systemrelevantes beschrieben wird

Identifizierung von Klassenkandidaten

- Klassen für einzelnes Mitglied und Verwaltung von Mitgliedern
- entsprechend für Mitarbeiter, Buch, Hörbuch, Spiel und Ausleihvorgang
- Mitglied hat Attribute Name, Privatanschrift, Mitgliedsnummer
- Klassen zur Verwaltung der Mitglieder, Medien und Ausleihvorgänge sollen die Ausgabe auf dem Bildschirm ermöglichen, haben also z. B. eine Methode print()
- Daraus folgt, dass jedes Objekt der Klassen Mitglied, Mitarbeiter, Buch, Hörbuch, Spiel, Ausleihvorgang auch einzeln auf dem Bildschirm ausgegeben werden kann, Klassen haben also z. B. Methode print()
- Bibliothek beschreibt Gesamtsystem (z. B. denkbarer Name einer Klasse, die im Falle einer Java-Implementierung die main()-Methode enthält).

Von Klassenkandidaten zum Klassendiagramm: CRC-Karten

3. Schritt: Klassenkarten entwerfen

- CRC-Kartentechnik (CRC – Classes, Responsibilities, Collaborators)
- Didaktisches Hilfsmittel zur Vermittlung objektorientierten Denkens.
- Erstellung je einer Klassenkarte je gefundenem Klassenkandidaten.
- Informelle Erfassung
  - wofür Klasse zuständig ist (links) – das führt später zu Attributen und Methoden,
  - mit Objekten welcher anderen Klassen zusammengearbeitet wird, um das Systemziel zu erreichen – führt zu sog. Klassenbeziehungen,
  - Quellen: Klassenkandidaten, Aufgabenbeschreibung, Anwendungsfallbeschreibungen
- Aufbau:

<Klassenname>	
<Zuständigkeiten>	<Zusammenarbeit mit>

- Zwischenschritt:
  - Prüfen und ggf. Modifizieren der CRC-Karten der Klassenkandidaten.
  - Klassenkandidaten mit mehr als 4 bis 5 Zuständigkeiten sollten aufgeteilt werden.
  - Klassenkandidaten mit 0 bis 1 Zuständigkeiten sollten verworfen werden, sofern sich ein anderer Klassenkandidat findet, der diese Zuständigkeiten sinnvoll übernehmen kann.
- Dann: Anordnung der Karten z. B. an Tafel nach Zusammengehörigkeit, ggf. Verbindungslinien zwischen zusammenarbeitenden Klassen.
- Führt zu einer ersten Version eines Klassendiagramms.
- Dann: Überführung in UML-Klassendiagrammschreibweise.

UML-Klassendiagramm

- beschreibt die im betrachteten System vorkommenden Klassen und deren Beziehungen untereinander
- Bislang: nur einzelne Klassen
  - strukturelle Eigenschaften, innere Struktur: Attribute
  - Verhalten: Methoden
- Ab jetzt auch: Beziehungen zwischen den Objekten von Klassen
- Varianten:
  - Assoziationen, Aggregationen, Kompositionen; können genauer modelliert werden durch
    - Multiplizitäten
    - Rollennamen und Navigationspfeile
  - Spezialisierungen/Generalisierungen

Assoziationen

- (zweistellige) Assoziation (engl. binary association):
- setzt Objekte von genau zwei Klassen zueinander in Beziehung
- UML-Darstellung:
  - Einfache Linie zwischen den Klassen.
  - Name (Bedeutung) der Assoziationsbeziehung (an der Mitte der Linie notiert) und Leserichtung des Namens (ausgefülltes Dreieck).
  - Anzahlangaben (auch: Multiplizitätsangaben, Vielfachheit): mit wie vielen Objekten der gegenüberliegenden Assoziationsseite ist je ein Objekt der Ausgangsseite mindestens/höchstens verbunden.
  - Rollennamen: bezeichnen die Bedeutung der beteiligten Klassen bzw. ihrer Objekte.
  - Assoziationen können uni- oder bidirektional sein (Pfeilspitze an Linie), legt die Navigierbarkeit der Assoziation fest.

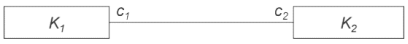
Assoziationen

Assoziationen werden zu Attributen von an der Assoziation beteiligten Klassen.

```
public class BibMitglied {
        ...
        // --- Attribute ---
        private AusleihVorgang[] av;
        private String name;
        private String privAnschrift;
        private int mitgliedsNr;
        ...
}

public class AusleihVorgang {
        ...
        // --- Attribute ---
        private BibMitglied ausleiher;
        private Buch leihObjekt;
        private Datum leihBeginn;
        private Datum leihEnde;
        ...
}

public class Buch {
        ...
        // --- Attribute ---
        private String autor;
        private String name;
        private int erscheinungsJahr;
        private String verlag;
        private int buecherNr;
        ...
}
```

Multiplizitäten (engl. multiplicity)

- Multiplizitäten geben an, wie viele Objekte der an der Assoziation beteiligen Klassen jeweils miteinander in Beziehung stehen können.
- Bei zweistelligen Beziehungen:



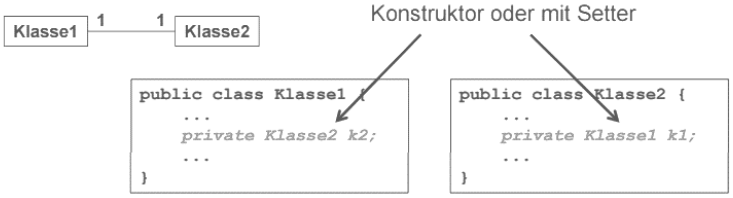
- Betrachte Assoziation zwischen Klassen K1 und K2, wobei jedes Ende der Assoziation eine Multiplizität hat.
- Multiplizität c2 drückt für jeden Zeitpunkt t die Anzahl jener Objekte von K2 aus, die mit genau einem Objekt von K1 in Beziehung stehen müssen/dürfen.
- Entsprechendes gilt analog für Multiplizität c1

Assoziationen mit 1:1-Multiplizität in Java

Fall 1: unidirektional, gerichtet

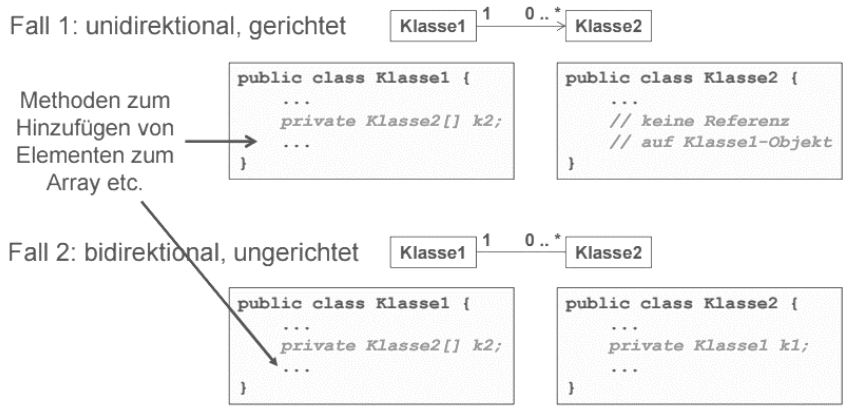
Fall 2: bidirektional, ungerichtet

Verknüpfung erfolgt im Konstruktor oder mit Setter



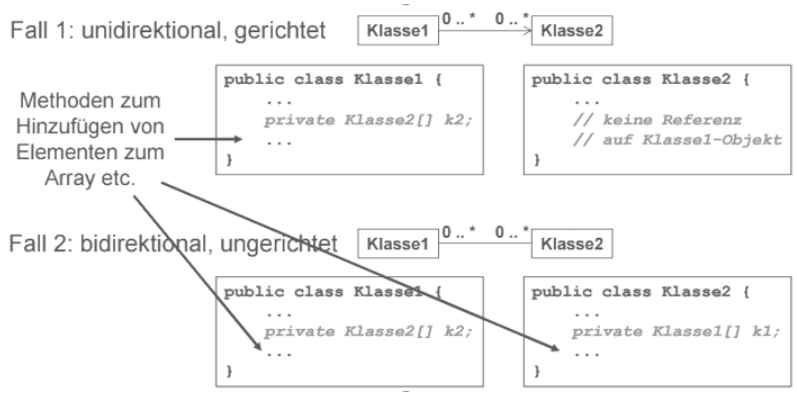
Assoziationen mit 1:n-Multiplizität in Java

Hinweis: „n“ bedeutet potentiell „viele“: zur Verwaltung von mehreren Objekten gleichen Datentyps kennen wir bislang nur Arrays. Später folgen sog. Container-Klassen, die analog verwendet werden können.



Assoziationen mit n:m-Multiplizität in Java

Hinweis: „n, m“ bedeutet potentiell „viele“: zur Verwaltung von mehreren Objekten gleichen Datentyps kennen wir bislang nur Arrays. Später folgen sog. Container-Klassen, die analog verwendet werden können.

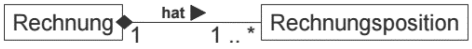


Aggregation (engl. aggregation)

- Aggregation: Spezialform der Assoziation
- Wie Assoziation Beziehung zwischen Klassen.
- Modelliert eine „Teil-Ganzes-“, „enthält-“ oder „hat-Beziehung“.
- UML-Darstellung: unausgefüllte Raute am Beziehungsende auf der Seite des Behälters/des Ganzen.
- Beispiele:

Komposition (engl. composition, composite aggregation)

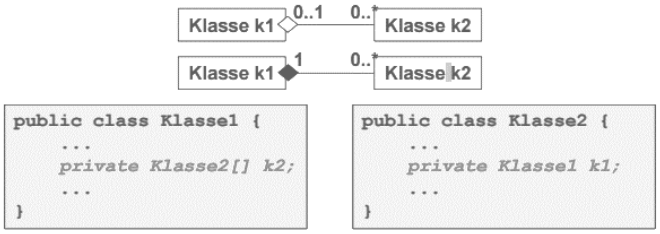
- Komposition: Spezialform der Aggregation
- Aggregation, bei der ein Bestandteil genau zu einem Ganzen gehört und nicht ohne das Ganze existieren kann.
- UML-Darstellung: ausgefüllte Raute am Beziehungsende auf der Seite des Behälters / des Ganzen
- Beispiel



- Semantik:
  - Eine Rechnungsposition gehört immer zu einer Rechnung.
  - Wird die Rechnung gelöscht, werden auch alle davon existenzabhängigen Teile gelöscht.

Aggregation und Komposition in Java

- Wie Assoziation, mit einigen Ergänzungen.
- Aggregation:
  - Für konkretes Objektpaar darf k1 und/oder k2 gleich null sein.
  - Typisch, dass das Ganze stellvertretend für seine Teile handelt.
  - Beispiel: Methode print() der Klasse BibMitgliederVerwaltung durchläuft alle verwalteten BibMitglieder und ruft jeweils deren print()-Methode auf.
- Komposition:
  - k2 darf gleich null sein, k1 muss stets ungleich null sein.



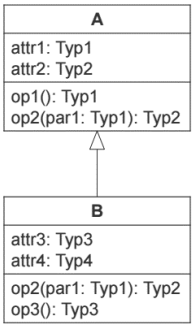
Einschub: String-Vergleich

- Für den Vergleich von Zeichenketten stellt die Java-API in der Klasse String folgende Methode bereit: boolean equals(Object anObject) vergleicht den gegebenen String mit dem spezifizierten Objekt.
- Ergebnis ist true, dann und nur dann, wenn das Argument nicht null und ein String-Objekt ist, das dieselbe Sequenz von Zeichen repräsentiert wie das gegebene Objekt.
- Beispiel:

```
String s1 = new String("hellostudents");
String s2 = new String("hellostudents");
String s3 = "hello" + "students";
boolean istGleich;
istGleich = s1.equals(s2); // istGleich: true
istGleich = s1 == s3; // istGleich: false
istGleich = s1.equals(s3); // istGleich: true
```

Vererbung

- Mechanismus, der es ermöglicht, sog. Unterklasse(n) aus einer gegebenen Klasse abzuleiten.
- Modelliert eine „ist ein“-Beziehung, z. B. Mitarbeiter ist ein BibMitglied, Buch ist ein Medium, Hund ist ein Tier, Informatik ist ein Studienfach, …
- Unterklasse erbt dann von der (Ober-) Klasse die Attribute und Methoden.
- Es können weitere Attribute und Methoden hinzukommen, die der Spezialisierung dienen (Spezialisierungsbeziehung zwischen den beteiligten Klassen).
- Aus Unterklasse können weitere Klassen abgeleitet werden, wodurch sich sog. Vererbungshierarchie ergibt.
- Darstellung im UML-Klassendiagramm
  - Linie von Unterklasse zur Oberklasse mit unausgefülltem Dreieck als Pfeilspitze auf der Seite der Oberklasse.
- Definition der Bestandteile (= Attribute und Methoden) der Unterklasse B einer Oberklasse A (Auswahl):
  - Neue Bestandteile von B werden in B
  - selbst festgelegt. Hier: attr3, attr4, op3
  - Abgeleitete Bestandteile von B werden von der Oberklasse A übernommen (Verhaltensgleichheit). Hier: attr1, attr2, op1, op2
  - Überschriebene Bestandteile (nur Methoden) entstehen, wenn die Unterklasse B eine Methode mit genau derselben Signatur wie in der Oberklasse A definiert (und eine neue Implementierung vorliegt). Das ist Spezialisierung im engeren Sinn. Hier: op2



Ober- und Unterklasse

- Eine Oberklasse (auch: Superklasse, engl. super class) beinhaltet die Grundlage für alle von ihr abgeleiteten Unterklassen und ist eine Verallgemeinerung aller ihrer Unterklassen.
  - Beispiel: BibMitglied ist Oberklasse (Superklasse) von / ist eine Generalisierung von BibMitarbeiter
- Eine Unterklasse (auch: Subklasse, engl. sub class) erbt die Attribute und Methoden der Oberklasse, beinhaltet aber noch zusätzliche oder veränderte Eigenschaften der Oberklasse. Die Unterklasse ist eine Spezialisierung der Oberklasse.
  - Beispiel: BibMitarbeiter ist Unterklasse von / ist abgeleitet aus / erbt von / ist eine Spezialisierung von BibMitglied

Die „oberste Oberklasse“: Klasse Object

- In Java ist jede Klasse automatisch Unterklasse der vordefinierten Klasse Object.
- Klasse Object stellt vordefinierte Methoden zur Verfügung, die somit allen Objekten zur Verfügung stehen (gilt auch für Arrays).
- Es kann sinnvoll sein, diese Methoden in Unterklassen passend zu überschreiben.
- Beispiel: toString() liefert textuelle Repräsentation des Objekts

Vererbung in Java

- Vererbung mittels extends bei der Klassen-Definition
- jede Klasse erbt von genau einer anderen Klasse
  - falls nicht explizit angegeben, ist die Oberklasse Object
  - alle Klassen in Java erben direkt oder indirekt von Object
- Einfachvererbung führt zu einer sog. Mono-Hierarchie
- Zugriff auf Methoden oder Attribute der Oberklasse: super
  - notwendig, falls Unterklasse Teile der Oberklasse verdeckt
- falls die Oberklasse keinen Standard-Konstruktor hat:
  - expliziter Aufruf eines Oberklasse-Konstruktors in jedem Konstruktor der Unterklasse notwendig

Vererbung: Beispiel

```
class Form {
        protected int x ;
        protected int y ;
        public Form ( int x , int y ) {
                this . x = x ; this . y = y ;
        }
}
class Rechteck extends Form { // jedes Rechteck ist auch eine Form
        protected int breite ;
        protected int hoehe ;
        public Rechteck ( int x , int y , int breite , int hoehe ) {
                super (x , y ); // Aufruf des Konstruktors der Oberklasse
                this . breite = breite ; this . hoehe = hoehe ;
        }
}
class Kreis extends Form { // jeder Kreis ist auch eine Form
        protected int radius ;
        public Kreis (int x , int y , int radius ) {
                super (x , y ); // Aufruf des Konstruktors der Oberklasse
                this . radius = radius ;
        }
}
```

Abstrakte Klassen

- abstrakte Klasse:
  - kann nicht instanziiert werden
    - es können also keine Objekte direkt von dieser Klasse erzeugt werden,
    - sondern nur von (nicht-abstrakten) Unterklassen dieser Klasse
  - kann als statischer Typ einer Referenz-Variable verwendet werden
- in Java: Deklaration einer abstrakten Klasse mittels abstract
- Beispiel

```
abstract class Form { /* ... */ }
class Rechteck extends Form { /* ... */ }
class Kreis extends Form { /* ... */ }
```

Abstrakte Methoden

- abstrakte Klassen können abstrakte Methoden deklarieren
  - legen lediglich die Signatur fest, nicht die Implementierung
- alle Unterklassen der abstrakten Oberklasse...
  - müssen diese abstrakten Methoden implementieren, oder
  - sind selbst abstrakt

Einschub: Wrapper-Klassen

- Viele Methoden (z. B. auch in Java eingebaute) erwarten Parameter vom Typ Object.
- Um diesen Methoden Werte eines primitiven Datentyps zu übergeben, stellt Java für jeden Datentyp eine sog. WrapperKlasse zur Verfügung, die diesen Wert kapselt.
- Wrapper-Klassen: Boolean, Byte, Character, Double, Float, Integer, Long, Short
- Jede Wrapper-Klasse besitzt u. a. eine Methode, um den primitiven Wert zurückzugewinnen.
- Vorgang des Ver- bzw. Entpackens: Boxing bzw. Unboxing
- Beispiel:
  - Double d = new Double(1.0);
  - double dv = d.doubleValue();
- Wrapper-Objekte als Parameter in Methoden

```
Integer getInt(Integer wert) {
    int i = wert.intValue();
    return new Integer(i);
}
...
System.out.println(getInt(new Integer(10)));
```

mit expliziter Erzeugung und Rückumwandlung

```
Integer getInt(Integer wert) {
    int i = wert;
    return i;
}
...
System.out.println(getInt(10));
```

mit sog. ***Auto(un)boxing*** (automatische Erzeugung und Konvertierung)

- Weiterer Vorteil von Wrapper-Klassen: können zusätzliche nützliche (Klassen-) Methoden bereitstellen, wie z. B. Double.isNaN(…) (s.o.), die die primitiven Datentypen nicht haben.

Polymorphismus (engl. polymorphism)

- Polymorphie = Fähigkeit, verschiedene Gestalt anzunehmen
- bedeutet, dass eine Erscheinung in vielfacher Gestalt auftritt
- Beispiel: polymorphe Methoden (engl. polymorphic methods)
  - Methoden haben gleichen Namen, tun aber etwas Unterschiedliches.
  - Beispiel (sog. Überladen):
    - Addition + ist für Datentypen int und float definiert
    - für jeden Datentyp eigene Operation
    - gleicher Name aus Komfortgründen
- Java: Deklaration polymorpher Methoden durch
  - Überladung von Methoden
  - Überschreibung von Methoden

Überladung einer Methode (engl. method overloading)

- Mehrfachdefinition von Methoden
- liegt vor, wenn in einem Programmstück mindestens zwei Methodendeklarationen sichtbar sind, die
  - denselben Namen haben und
  - Parameterlisten haben, in denen sich die Datentypen der Parameter in der aufgelisteten Reihenfolge an mindestens einer Stelle oder die sich in der Parameteranzahl unterscheiden.
  - Der Resultattyp spielt in Java beim Überladen keine Rolle.
- Auch ein Konstruktor kann überladen werden.
- Überladen von Methoden ist möglich innerhalb einer Klasse oder innerhalb einer Klasse und einer Unterklasse.
- Beispiel: System.out. …
  - println(long), println(float), println(Object), …
  - Je nach Typ des Arguments arg im Aufruf println(arg) wird die „passende Version“ der Methode println aufgerufen

Einschub: Methodenauswahl in Java

- Finden der anwendbaren und zugreifbaren Methoden
  - anwendbar:
    - Suche in angegebener Klasse und deren Oberklassen
    - Argumentanzahl = Parameteranzahl
    - „Method Invocation Conversion“: kann der Typ jedes Arguments in den Typ des formalen Parameters konvertiert werden?
  - zugreifbar:
    - kann Methode gemäß Modifikator public, private aufgerufen werden? (Später auch: protected und ohne Modifikator).
- Auswahl der spezifischsten Methode

Überschreibung einer Methode (engl. method overwriting)

- liegt vor, wenn es zu einer Methode, die in einer Klasse deklariert ist, eine Methode in einer Unterklasse gibt, die
  - denselben Namen und
  - dieselbe Parameterliste hinsichtlich der Parametertypen in der Reihenfolge der Liste und
  - denselben Resultattyp (oder einen zuweisungskompatiblen engeren Typ davon) hat.
- Anwendung:
  - Objekt der Oberklasse: Verwendung der Methode der Oberklasse
  - Objekt der Unterklasse: Verwendung der Methode der Unterklasse
- Beispiel 1: Methode print() im Bibliotheksbeispiel
  - BibMitglied: print() gibt Daten des BibMitglieds aus.
  - BibMitarbeiter: print() gibt Daten des BibMitglieds zusätzlich zu Daten des BibMitarbeiters aus
- Beispiel 2:
  - Klasse Object hält eine Standardimplementierung von toString() bereit, die (bei Bedarf) in eigenen Klassen überschrieben werden kann.
  - Je nach Typ eines Objekts obj wird bei obj.toString() unterschiedlicher Programmtext ausgeführt.
- Hinweis:
  - Auf eine überschriebene Instanzmethode der Oberklasse kann im Inneren der überschreibenden Klasse per super zugegriffen werden.
  - Ein Aufruf von außen ist nicht möglich

Überladen vs. Überschreiben

- Überladene Methoden
  - haben unterschiedliche Signaturen,
  - haben im Allgemeinen unterschiedliche Implementierungen.
  - Vorkommen:
    - in einer Klasse und einer ihrer Unterklassen oder
    - in einer Klasse (nebeneinander).
  - Klassenmethoden können überladen werden.
- Überschriebene Methoden
  - haben dieselben Signaturen (bis auf evtl. engeren Resultattyp)
    - Klassenmethoden können keine Instanzmethoden überschreiben
    - umgekehrter Fall auch nicht
  - haben im Allgemeinen unterschiedliche Implementierungen.
  - Vorkommen:
    - in einer Klasse und einer ihrer Unterklassen.
  - Klassenmethoden können nicht überschrieben werden (aber „verdeckt“ werden).

Klasse vs. Typ

- Typ
  - Eigenschaft von Variablen und Ausdrücken,
  - definiert Mindestforderung bzgl. anwendbarer Operationen,
  - rein syntaktisch festgelegt (statisch ermittelbar).
- Klasse
  - stellt Konstruktor(en) für Objekte bereit (nicht für abstrakte Klassen),
  - definiert Signatur oder Implementierung der Operationen.
  - Objekt gehört zu der Klasse, mit deren Konstruktor es konstruiert wurde.
  - Mit der Klassendefinition wird ein gleichnamiger Typ eingeführt.
- Hinweis
  - Variablen gleichen Typs können Objekte unterschiedlicher Klassenzugehörigkeit referenzieren (benennen).
  - Sie werden sich im Allgemeinen unterschiedlich verhalten, z. B. Oberklasse A mit zwei Unterklassen A1 und A2. Variable kann mit Typ A deklariert werden und kann Objekte aus A, A1 oder A2 referenzieren.

Polymorphe Variablen in typsicheren Sprachen

- Sei v eine Variable mit Referenzsemantik vom Typ T bzw. der Klasse T.
- Dann dürfen der Variablen v Verweise auf Objekte der Klasse T oder einer beliebigen Unterklasse von T zugewiesen werden.
- Erklärung: Unterklasse verfügt über alle Eigenschaften der Oberklasse und kann daher deren Platz einnehmen
- Typsicherheit: Es dürfen nur Methoden aufgerufen werden, die schon beim statischen Typ von m (Oberklasse Medium) verfügbar sind.

```
// statischer Typ von m: Medium
Medium m;
// dynamischer Typ von m: Hoerbuch
m = new Hoerbuch("Petra Panisch", "Das Super-Mega-Drama", 2007,
                                          "Dramaqueens", 8, 495);
// Typsicherheit:
// alle Methoden von Medium koennen auf m aufgerufen werden
m.print();           // die anzuwendende Implementierung der Objektmethode
                     // wird zur Laufzeit ausgewaehlt
```

- Eine polymorphe Variable (engl. polymorphic variable) kann im Laufe der Ausführung eines Programms Referenzen auf Objekte verschiedener Klassen haben. Eine polymorphe Variable hat einen
  - statischen Typ: wird durch Angabe der Klasse bei der Deklaration angegeben und kann bei der Übersetzung überprüft werden,
  - dynamischen Typ: wird durch die Klasse des Objekts angegeben, auf den die Variable zur Laufzeit zeigt

Dynamische und statische Bindung

- Klassenhierarchien:
  - Variable, die vom Typ der Oberklasse deklariert ist, kann auch auf Instanzen von deren Unterklassen verweisen.
  - Dort sind eventuell Methoden der Oberklasse überschrieben.
- Kandidaten für Methodenaufruf des in der Variablen referenzierten Objekts:
  - Methode der Oberklasse
  - überschriebene Methodenversion aus der Unterklasse
- Dynamische Bindung (bei Instanzmethoden): zur Laufzeit wird in Abhängigkeit von der Tatsache, ob das Objekt eine Instanz der Ober oder einer Unterklasse ist, die jeweilige Version der Methode aufgerufen.
- sog. virtueller Methodenaufruf
- Ermöglicht so die volle Flexibilität von Vererbung.
- Statische Bindung (bei Klassenmethoden): es wird immer die Methode des Typs genutzt, als der die Variable deklariert ist

Verdecken von (Klassen-) Attributen und Klassenmethoden

- bedeutet, dass
  - ein in der Unterklasse deklariertes Attribut oder Klassenattribut denselben Namen aufweist, wie ein entsprechendes (Klassen-) Attribut der Oberklasse,
  - eine in der Unterklasse deklarierte Klassenmethode dieselbe Signatur (gleicher Name, gleiche Parameterliste (bzgl. Reihenfolge und Typen der Parameter) und gleicher Resultattyp) aufweist, wie eine entsprechende Klassenmethode der Oberklasse.
- Hierbei wird innerhalb der Unterklasse
  - das Attribut oder Klassenattribut der Oberklasse durch das namensgleiche Attribut der Unterklasse verdeckt,
  - die Klassenmethode der Oberklasse durch die signaturgleiche Klassenmethode der Unterklasse verdeckt.
- Regel: beim Zugriff auf (Klassen-) Attribute und Klassenmethoden ist immer der statische Typ an der Zugriffsstelle der relevante

Verdecken von Attributen am Beispiel

Die in der Unterklasse deklarierten Attribute ergänzen die Attribute der Oberklasse; sie kommen zusätzlich hinzu.

```
class Ober {
    public String s = "Ober";
    public int i = 1;
}
```

```
class Unter extends Ober {
    public String s = "Unter";
    public int i = 2;

    public void ausgabe() {
        System.out.println(s + ", "
                           + super.s);
    }
}
```

statischer Typ: Ober

dynamischer Typ: Ober

```
...
Ober o = new Ober();
System.out.println(o.s + ", " + o.i);

Unter u = new Unter();
u.ausgabe();
System.out.println(u.s + ", " + u.i);

Ober o2 = (Ober)u;
System.out.println(o2.s + ", " + o2.i);
...
```

Ausgabe:

```
Ober, 1
Unter, Ober
Unter, 2
Ober, 1
```

(Explizite) Typkonvertierung (engl. casting)

Prüfe mit instanceof, zu welcher Klasse ein konkretes Objekt gehört.

```
Uhr c;
...
if (c instanceof DigitalUhr) {
 ...
}
```

- Im Inneren der if-Anweisung hat c immer noch den statischen Typ Uhr.
- Methoden, die nur die Unterklasse DigitalUhr anbietet, können wegen der geforderten Typsicherheit nur umständlich aufgerufen werden; Abhilfe: explizite Typwandlung

```
Uhr c;
...
if (c instanceof DigitalUhr) {
 DigitalUhr digital = (DigitalUhr)c;
 ...
}
```

Schnittstellen

- Mit einer Schnittstelle (eingeleitet durch Schlüsselwort: interface) wird in Java definiert, welche Methoden eine diese Schnittstelle implementierende Klasse (Schlüsselwort: implements) mindestens haben muss.
- In einem Interface können deklariert werden:
  - Konstanten und
  - Signatur(en) von Methode(n); Die Implementierung der angegebenen Methode(n) muss in einer Klasse erfolgen.
  - Alles andere ist nicht zulässig.
- In einer Interfacedeklaration haben
  - Konstantendeklarationen automatisch die Modifizierer static und final und public,
  - Methodendeklarationen automatisch die Modifizierer abstract und public.
  - Deshalb brauchen diese nicht angegeben zu werden.
- Typ einer Objektvariablen kann nun auch eine Schnittstelle sein (statt bisher: Klasse).
  - Referenz kann aber nur auf ein Objekt verweisen.
  - Wird ihr ein Objekt zugewiesen, so muss dessen Klasse die Schnittstelle implementieren.
- Vererbung zwischen Schnittstellen mit extends
- Mehrfachvererbung von Schnittstelle ist möglich:

```
interface Inter3 extends Inter1, Inter2 {
    ...
}
```

```
class Demo implements Inter1, Inter2 {
    ...
}
```

- Auflösung/Behandlung potentieller Mehrdeutigkeiten bei Mehrfachvererbung bei Schnittstellen:
  - Methoden: Werden zwei gleich benannte Methoden über verschiedene Pfade ererbt,
    - so wird bei identischen Signaturen eben diese Signatur übernommen,
    - so werden bei unterschiedlichen Signaturen beide übernommen und dann als überladene Methoden behandelt.
  - Konstanten:
    - Werden zwei gleich benannte Konstanten über verschiedene Pfade geerbt, meldet der Übersetzer Fehler, (nur) wenn die Konstante verwendet wird.
    - Ein Interface kann eine Konstante deklarieren, die eine/mehrere geerbte Konstanten gleichen Namens verdeckt.

Pakete

- Paket (engl. package): Zusammenstellung der Vereinbarungen von als zusammengehörig betrachteten Java-Klassen
- und Java-Interfaces.
- Paket besitzt einen Namen, der – wie von Dateiverzeichnissen her bekannt – hierarchisch aufgebaut sein kann:
  - java.lang
  - com.apple.quicktime.v7
- Paketname definiert Namensraum für die im Paket vorkommenden Klassendeklarationen und sollte nur aus Kleinbuchstaben bestehen.
- Hinweis:
  - Daher prinzipiell/theoretisch möglich, innerhalb eines Pakets eine neue Klasse String zu realisieren.
  - Eindeutigkeit erfolgt durch Verwendung qualifizierter Klassennamen (java.lang.String bzw. <mein_paket>.String).

Paket uhren: Interface Taktgeber und Klasse TimeZone

```
package uhren;
interface Taktgeber {
        long leseZeitInSekunden();
}
```

```
package uhren;
public class TimeZone {
        private String identifikator;
        public TimeZone(String id) {
              setzeIdentifikator(id);
        }
        public void setzeIdentifikator(String id) {
              identifikator = id;
        }
 ...
}
```

Klasse Uhr

```
package uhren;
public abstract class Uhr implements Taktgeber {
        protected static long zaehler = 0;
        protected long seriennummer;
        protected long minuten;
        protected long stunden;
        protected TimeZone zeitzone = new TimeZone("UTC");
        protected Uhr() {
              seriennummer = zaehler++;
        }
        protected Uhr(long min) {
               this();
               stunden = min / 60;
               minuten = min % 60;
        }
public long leseSeriennummer() {
       return seriennummer;
 }

 public abstract void anzeigen();
 public void setzeZeit(long std, long min) {
       stunden = std; minuten = min;
 }
 public void setzeZeit(long std) {
       setzeZeit(std, 0);
 }
 public long leseZeitInSekunden() {
       return (stunden * 3600) + (minuten * 60);
 }
public void setzeZeitzone(TimeZone tz) {
        if (tz == null) {
                     return;
        }
        zeitzone = tz;
 }
 public TimeZone leseZeitzone() {
       return zeitzone;
 }
 ...
}
```

Zugriff auf Klassen/Schnittstellen eines Pakets von Außerhalb

- Um außerhalb eines Pakets eine darin deklarierte Klasse oder Schnittstelle zu verwenden, gibt es zwei Möglichkeiten:
- Zugriff auf eine Klasse in einem Paket über den qualifizierten Namen <Paketname>.<Klassename>
- Alternative:
- Paketname wird mittels import bekannt gemacht.
- Dann kann auf eine im Paket enthaltene Klasse mittels <Klassenname> zugegriffen werden

Klassenbibliothek (engl. library)

- Sammlung vordefinierter, häufig verwendeter Klassen, auf die bei der Programmierung zugegriffen werden kann.
- Java: jeweils Menge von Paketen
- Java selbst liefert z. B. die Pakete:
  - java.io Ein-/Ausgabe in Dateien o. ä.
  - java.lang zentrale Klassen von Java (z. B. Object)
  - Alles in java.lang.* wird automatisch importiert.
  - java.net Verwaltung von Netzwerkverbindungen
  - java.sql Datenbankanbindung
  - java.util Nützliche Klassen (z. B. Kalender, Listen)
  - javax.swing graphische Oberfläche

Methoden ausgewählter Bibliotheksklassen

- Klasse java.lang.System stellt die Schnittstelle einer laufenden Java-Applikation zur Systemumgebung bereit.
- Zunächst am wichtigsten: Standard-Ein/Ausgabe, über die Zeichen mit der Shell ausgetauscht werden können:
  - static PrintStream out The “standard” output stream.
  - static PrintStream err The “standard” error output stream.
  - static InputStream in The “standard” input stream.
- Wichtigste Methoden von java.lang.PrintStream:
  - void print(String s) Print a string.
  - void println() Terminate the current line by writing the line separator string.
  - void println(String x) Print a String and then start a new line.

Konstruktoren der Klasse java.lang.String

- String() Initializes a newly created String object so that it represents an empty character sequence.
- String(char[] value) Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
- String(char[] value, int offset, int count) Allocates a new String that contains characters from a subarray of the character array argument.
- String(byte[] bytes) Constructs a new String by decoding the specified array of bytes using the platform's default charset.
- String(byte[] bytes, String charsetName) Constructs a new String by decoding the specified array of bytes using the specified charset.
- String(byte[] bytes, int offset, int length) ...
- String(byte[] bytes, int offset, int length, String charsetName) ...
- String(String original) Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string
- char charAt(int index) Returns the character at the specified index.
- int compareTo(String anotherString) Compares two strings lexicographically (returns 0 if equal).
- int compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
- boolean endsWith(String suffix) Tests if this string ends with the specified suffix.
- boolean equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
- int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
- int lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
- String replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
- String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
- String toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale

Bibliotheksklasse java.lang.Math

- Klasse java.lang.Math enthält Methoden, die mathematische Funktionen zur Verfügung stellen.
- Methoden dieser Klasse sind statische Methoden.
- Sie können verwendet werden, indem dem Methodennamen Math. vorangestellt wird.

Konstanten und Methoden der Klasse Math (Auswahl)

- static double E The double value that is closer than any other to e, the base of the natural logarithms.
- static double PI The double value that is closer than any other to pi, the ratio of the circumference of a circle to its diameter.
- static double abs(double a) Returns the abs. value of a double value.
- static float abs(float a) Returns the abs. value of a float value.
- static int abs(int a) Returns the absolute value of an int value.
- static long abs(long a) Returns the absolute value of a long value.
- static double sin(double a) Returns the trigonometric sine of an angle.

Konstanten und Methoden der Klasse Math (Auswahl)

- static double exp(double a) Returns Euler's number e raised to the power of a double value.
- static double log(double a) Returns the natural logarithm (base e) of a double value.
- static double max(double a, double b) Returns the greater of two double values.
- static double pow(double a, double b) Returns the value of the first argument raised to the power of the second argument.

Beispiel statischer und dynamischer Typ Übung

```
interface Sammlerstueck { /* ... */ }
class Oldtimer implements Sammlerstueck { /* ... */ }
class Briefmarke implements Sammlerstueck { /* ... */ }
// ===================================
Sammlerstueck objekt = null ;
       // stat . Typ: Sammlerstueck , dyn. Typ: keiner
objekt = new Oldtimer ();
       // stat . Typ: Sammlerstueck , dyn. Typ: Oldtimer
objekt = new Briefmarke ();
       // stat . Typ: Sammlerstueck , dyn. Typ: Briefmarke
```

Robustes Programmieren

Strategien zum Umgang mit Fehlern bei der Programmentwicklung

- Identifizierung möglicher Fehlerquellen mit Checklisten
  - Berücksichtigung von Checklisten trägt dazu bei, bestimmte Fehlerquellen zu identifizieren und eventuelle Fehler zu beseitigen.
- Absicherung von Quellcode gegen fehlerhafte Verwendung
  - Besondere Beachtung von Schnittstellen (Code, der fremden Code verwendet oder der von fremdem Code verwendet wird).
  - Strategien: Parameterprüfung bei Methoden, Fehlercodes, Ausnahmebehandlung zur Laufzeit.
- Testen von Programmen
  - Verschiedene Software-Testmethoden.
  - Umfassender Test aller erdenklichen Eingaben i. d. R. nicht möglich.
- Formale Verifikation von Programm(teil)en
  - Formaler Nachweis, dass Programm(stück) das Gewünschte leistet.
- Tolerenz gegen verbleibende Fehler
  - „Graceful Degradation“, sichere Abschaltung bei Ungereimtheiten

Programmentwicklung

- Gemeinsame Programmentwicklung
  - Es hat sich gezeigt, dass bei einer Programmentwicklung wesentlich weniger Fehler entstehen, wenn zwei Personen zusammen ein Programm entwickeln o. einer dem anderen d. entwickelten Code erklärt.
- Gemeinsames „Walkthrough“:
  - Begutachtung eines Programms.
  - Autor stellt den Gutachtern das Programm vor.
  - Meist werden typische Anwendungsfälle (Benutzungssituationen, Szenarien, Probeläufe) im Kreis gleichgestellter Mitarbeiter ablauforientiert durchgespielt.
  - Ziel: Fehler finden, z. B. anhand von Fragen aus Checklisten (s.u.) zur Fehleridentifizierung und -vermeidung.
  - Nacharbeit durch den Autor nach Ende der Sitzung.

Fehlervermeidung bei Methodenargumenten

- Übersetzer
  - Garantiert, dass Typ des aktuellen Parameterwertes beim Aufruf gleich dem Typ des zugehörigen formalen Parameters in der Deklaration ist (oder zumindest automatisch umgewandelt werden kann).
  - Beispiel: verhindert, dass Methode double sqrt(double d), mit String-Parameter aufgerufen wird.
- Dokumentation
  - Autor/-in einer Methode sollte in Kommentaren stets dokumentieren, was diese Methode macht, welche Argumente sie erwartet und was für einen Rückgabewert sie zurückliefert.
  - Bei Benutzereingaben sollte Benutzer/-in (z. B. durch passende Ausgabe) darauf hingewiesen werden, welche Daten einzugeben sind.
- Überprüfung der Parameterwerte am Methodenanfang
  - Naiv: Fehlercodes
  - Besser: Ausnahmen

Grenzen der Rückmeldung von Fehlern über Fehlercodes

- Programmcode und Fehlerbehandlungscode
  - Aufrufendes Programmstück muss Rückgabewert immer direkt testen.
  - Dadurch: Mischung von „normalem“ Programmcode und Fehlerbehandlungscode.
- Fehlermeldung
  - Will Methode Fehler mit Text dokumentieren oder erläutern, muss sie diesen Text selbst ausgeben oder in eine Protokolldatei schreiben.
  - Methode hat i. d. R. keine Möglichkeit, Text an den Aufrufer weiterzureichen, damit dieser entscheidet, was damit zu tun ist.
- Wertebereich
  - Oft sind alle Werte aus dem Wertebereich des Rückgabetyps gültige Ergebniswerte. Dann ist kein Wert für den oder die Fehlercodes übrig.

Ausnahmen (engl. exceptions)

- Java bietet einen leistungsstarken Mechanismus zur Behandlung von Fehlersituationen zur Laufzeit basierend auf sog. Ausnahmen.
- Eine Ausnahme (engl. exception) ist ein Ereignis, durch das die normale Ausführungsreihenfolge unterbrochen wird.
- Programmiertechnisch sind Ausnahmen Objekte bereits in Java enthaltener oder selbstdefinierter Ausnahmeklassen, wobei die Klasse die Ausnahmesituation näher charakterisiert, z. B.
  - IllegalArgumentException: illegaler Parameter
  - FileNotFoundException: benötigte Datei nicht gefunden
- Objekte solcher Ausnahmeklassen werden an der Fehlerstelle im Programm erzeugt und können dann Informationen z. B. über den Fehlerort oder eine Fehlermeldung an die Stelle im Programm „transportieren“, wo die Bearbeitung der Fehlersituation erfolgen soll (z. B. in die aufrufende Methode).
- Man kann nicht jede einzelne Ausnahme vorhersagen.
- Man kann jedoch die Arten von Ausnahmen vorhersagen. Daher werden Ausnahmen klassifiziert: Jeder Ausnahme wird ein Ausnahmetyp (die entsprechende Ausnahmeklasse) zugeordnet.
- Die Behandlung orientiert sich dann am Ausnahmetyp.
- Dies begründet die Trennung von Erkennung und Behandlung.

Aufgaben/Ablauf:

- Erkennen einer Ausnahmesituation
- Erzeugen einer Ausnahmebeschreibung (eines bestimmten Typs)
- Anstoßen der Ausnahmebehandlung („Werfen“)
- Behandeln der Ausnahme gemäß Typ

Beispiel

```
static double division(int zaehler, int nenner) {
        if (nenner == 0) {
                IllegalArgumentException e =
                        new IllegalArgumentException("Fehler: versuchte Division d. 0");
                throw e;
        }
       return zaehler / nenner;
}
```

- Den Konstruktoren der vordefinierten Fehlerklassen kann als Argument ein String mit einer Fehlerbeschreibung übergeben werden.
- Später: Abfrage dieses Strings möglich mit getMessage()-Methode des Ausnahmeobjekts.

```
class ExceptionDemo {
        static double division(int zaehler, int nenner) {
              ...
        }
        public static void main(String[] args) {
                try {
                        double bruch;
                        bruch = division(12,5);
                        System.out.println(bruch);
                        bruch = division(12,0); // loest Ausnahme aus
                        System.out.println(bruch);
                } catch (IllegalArgumentException e) {
                        System.err.println(e.getMessage());
                }
        }
}
```

Hierarchie in der Standardbibliothek definierter Ausnahmen

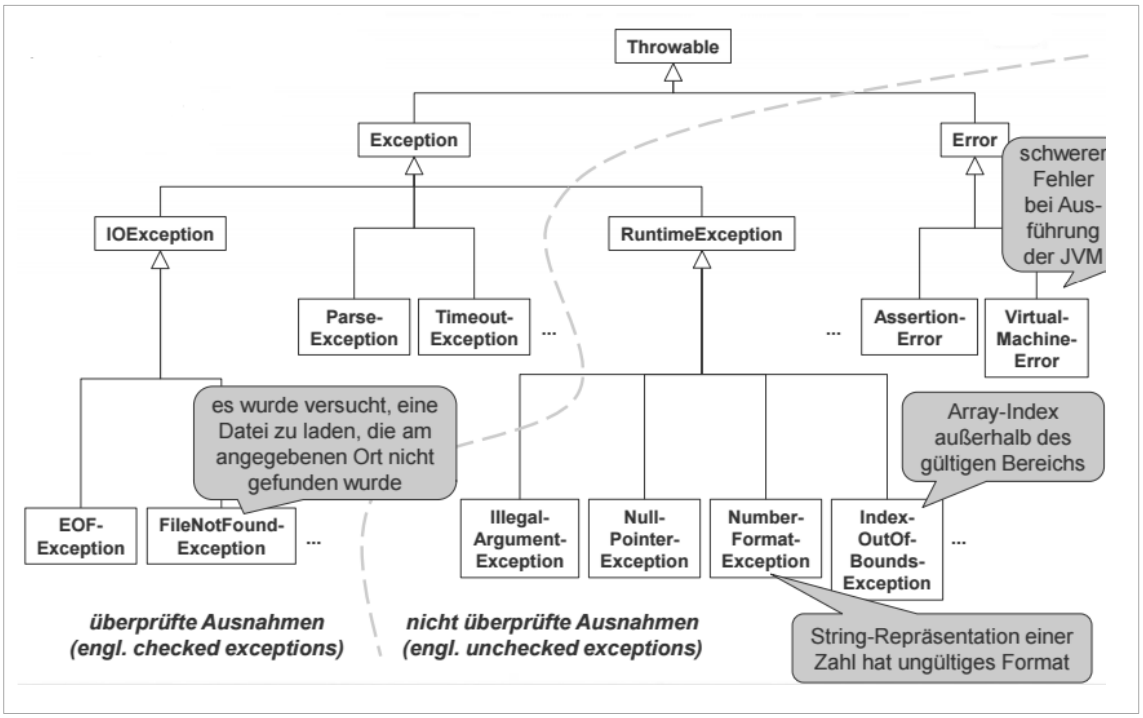
Ausnahmetypen sind in Java gewöhnliche Klassen;
Vorteile:

- Objekt kann Information über den Fehler (z. B. Fehlerbeschreibung, Ort des Auftretens etc.) an die Stelle transportieren, wo der Fehler behandelt wird.
- Über den Typ des catch-Parameters kann gezielt angegeben werden, welche Ausnahmen gefangen werden sollen.

Ausnahmetypen erben alle von der vordefinierten Klasse java.lang.Throwable
java.lang.Throwable hat (genau) zwei direkte Unterklassen:

- java.lang.Error zeigt Probleme bei der Programmausführung an, die das Programm i. d. R. nicht selbst beheben kann/sollte.
- java.lang.Exception ist die Oberklasse,unter der alle Ausnahmetypen zusammengefasst werden, die das Programm selbst behandeln möchte/sollte

Hierarchie in der Standardbibliothek definierter Ausnahmen



Nicht) überprüfte Ausnahmen (engl. (un-)checked exceptions)

- Überprüfte Ausnahmen (engl. checked exceptions)
  - Alle Ausnahmeklassen, die von der Klasse Exception mit Ausnahme von RuntimeException und deren Unterklassen abgeleitet sind.
  - Pflicht, solche Ausnahmen zu behandeln oder weiterzugeben.
  - Sonst Fehler bei der Übersetzung
- Nicht überprüfte Ausnahmen (engl. unchecked exceptions)
  - Alle Ausnahmeklassen, die von der Klasse Error oder RuntimeException sowie deren Unterklassen abgeleitet sind.
  - Deklaration nicht erforderlich, weil jede Methode RuntimeExceptions auslösen kann und Behandlung von Errors u. U. nicht möglich ist.

Ausnahmen behandeln

- try-Block umfasst Code, der Ausnahmen auslösen kann.
- Wird in einem try-Block eine Ausnahme geworfen, dann
  - wird die Ausführung des try-Blocks sofort abgebrochen.
  - Dann wird der Typ des Ausnahmeobjekts mit den Typangaben der nachfolgenden catch-Klauseln verglichen. Von oben nach unten.
  - Der erste catch-Block bei dem der Ausnahemtyp bzgl. Instanceof zur Typmenge passt (Untertyp passt auch!), behandelt die Ausnahme.
  - Der catch-Block gibt dem Ausnahmeobjekt einen Namen (analog formaler Parameter bei Methodendeklarationen).
  - Passt keiner der catch-Blöcke oder löst der catch-Block selbst eine Ausnahme aus, so wird die Ausführung der try-catch-Anweisung abgebrochen und die letzte Ausnahme an den Aufrufer übergeben (ggf. nachdem ein sog. finally-Block bearbeitet wurde, folgt gleich).
- Falls keine Ausnahme ausgelöst wurde oder nach Bearbeitung der passenden catch-Klausel wird (ggf. nach dem finally-Block) mit den nachfolgenden Anweisungen fortgesetzt.

Ausnahmen behandeln am Beispiel

```
try {
        ... // Code, der Ausnahmen ausloesen koennte
}
catch(IllegalArgumentException iae) {
        ... // Behandlungscode
}
catch(EOFException | ZipException fileEx) {
        ... // Behandlungscode
}
catch(Exception e) {
        ... // Behandlungscode
}
finally {
        ... // Abschlussarbeiten
}
System.out.println("Have a nice day!")
```

Der finally-Block

- Manchmal soll im Anschluss an den Code im try-Block ein Stück Code auf jeden Fall ausgeführt wird, egal ob im try-Block (oder in dem diese bearbeitenden catch-Block) eine Ausnahmesituation auftrat oder nicht. Z. B. Datenbanktransaktion rückabwickeln.
- Solcher Code kann am Ende der try-catch-Anweisung nach den catch-Klauseln in einem sog. finally-Block angegeben werden.

```
public static void m() {
        throw new RuntimeException("some exception");
}
public static void main(String[] args) {
         try {
                m();
         } catch (RuntimeException re) {
                // Behandlungscode
         } finally {
                // Dieser Code wird auf jeden Fall ausgefuehrt.
                System.out.println("Print this - no matter what!");
         }
}
```

Besondere try-Anweisung für zu schließende Ressourcen

- Für Objekte, die das Interface java.lang.AutoCloseable implementieren (und deshalb eine close()-Methode anbieten):

```
try (FileOutputStream fos = new FileOutputStream("out.txt");
         FileOutputStream bar = new ...) {
         fos.write(bytes, 0, bytes.length);
         bar.write(bytes, 0, bytes.length);
         fos.flush();
         bar.flush();
}
```

Anstoßen der Ausnahmebehandlung

- Die ausgeführte Methode wird sofort abgebrochen.
- Wenn die Methode besonderen Code für die Ausnahmesituation bereit hält (= Ausnahmebehandlung), dann wird dieser ausgeführt
- andernfalls:
    - Das Ausnahmeobjekt wird als eine Art Methodenresultat an die aufrufende Methode gegeben („die aufgerufene Methode wirft eine Ausnahme“).
    - Entweder wird dort die Ausnahmesituation behandelt oder das Ausnahmeobjekt wird zur nächsten Aufrufermethode weitergegeben usw.
    - Wird main()-Methode erreicht und auch dort kein passender Behandlungsblock gefunden, endet das Programm mit einer Fehlermeldung.

Ausnahmen weitergeben

```
import java.io.*;
class ExceptionDemo {
        // Deklarationen ...
        public static void write() {
               PrintWriter datei = new PrintWriter("daten.txt");
               ...
        }
        public static void main(String[] args) {
               try {
                      write();
               } catch (FileNotFoundException e) {
                      ... // Behandlungscode
               }
        }
}
```

- Möglichkeit 1: Ausnahme in der verwendenden Methode mit try-catch abfangen. (Möglichkeit hier nicht gewählt.)
- Möglichkeit 2: Ausnahme an aufrufende Methode weiterleiten (hier: main()) unter Verwendung einer throws-Klausel, die dem Übersetzer mitteilt, dass Methode Ausnahme des angegebenen Typs wirft.

Ausnahmen weitergeben

- Abhängig vom Typ der Ausnahme ist eine Behandlung oder Weitergabe erforderlich (bei überprüften Ausnahmen).
- Falls Behandlung (mittels catch-Block) nicht in der Methode selbst erfolgt, ist Weitergabe erforderlich.
- Methodenkopf deklariert Liste der Ausnahmtypen <ExceptionClass1>, …, <ExceptionClassn> die ggf. aus der Methode herausgegeben werden.
- Diese Liste ist Teil der öffentlichen Schnittstelle, damit Aufrufer weiß, welche Ausnahmen in der Methode auftreten können.
- Form:

```
<ModifierList> <MethodName>(<ParameterList>)
        throws <ExceptionClass1>, …, <ExceptionClassn>
```

- In der API-Dokumentation des JDK sind die von einer Methode ausgelösten Ausnahmen durch Angabe von throws in der Methodendeklaration erkennbar (und außerdem am Ende der Methodenbeschreibung einzeln und mit genauen Auslösebedingungen aufgeführt).
- Beispiel: Integer.parseInt

```
public static int parseInt(String s)
       throws NumberFormatException
```

- Nicht-überprüfte Ausnahmen (vom Typ java.lang.RuntimeException und java.lang.Error oder deren Unterklassen) können (aber müssen nicht) deklariert werden.

Methoden durch Ausnahmen absichern

```
class Zufallszahl {
        public static int zufallszahl(int min, int max) {
              ...
        }
        public static void main(String[] args) {
              System.out.println("Test der Zufallszahlen-Funktion");
               try {
                      int n = 0;
                      for (int i = 5; i > -5; i--) {
                            n = zufallszahl(0, i);
                            System.out.println(n);
                      }
               } catch(IllegalArgumentException e) {
                      System.err.println(e.getMessage());
                      System.err.println("Programm wird beendet");
               }
        }
}
```

Programmabsturz verhindern

```
class Zufallszahl {
        public static int zufallszahl(int min, int max) {
              ...
        }
        public static void main(String[] args) {
               System.out.println("Test der Zufallszahlen-Funktion");
               int n = 0;
               for (int i = 5; i > -5; i--) {
                      try {
                             n = zufallszahl(0, i);
                            System.out.println(n);
                      } catch(IllegalArgumentException e) {
                            System.err.println(e.getMessage());
                      }
               }
        }
}
```

- Aufruf und Ausgabe gehören logisch zusammen.
- Würde man die Ausgabe unter den catch-Block platzieren, würde in den Schleifendurchgängen, die zu einer Ausnahme führen, der jeweils alte Wert von n ausgegeben
- Schleife wird nicht mehr vorzeitig beendet

Eigene Ausnahmeklassen erstellen

- Die eigene Ausnahmeklasse muss von java.lang.Throwable erben:
  - Die Klasse sollte eine Unterklasse von java.lang.Exception sein.
  - java.lang.Error sollte man nicht nutzen, da man eigene (anwendungsorientierte) Ausnahmen davon unterscheiden können sollte.
- Ein Ausnahmeobjekt sollte den Fehlerzustand aufnehmen können, also muss man entsprechende Attribute vorsehen.
  - Standardmäßig können java.lang.Exception-Objekte optional eine Fehlernachricht (String) und/oder eine andere (verursachende) Exception aufnehmen (die z.B. in einer aufgerufenen Methode geworfen wurde sowie in der aktuellen Methode mit weiteren Informationen angereichert und anschließend nach oben weitergereicht wird).
  - z. B. throw new Exception("my exception message");

Eigene Ausnahmeklassen erstellen

```
public class Account {
        public static final double MAX_AMOUNT = 1000000;
        private double balance = 0;
        private double maxDebt = 0;
        ...
        public double getBalance() {
              return balance;
        }
        public void changeAmount(double amount) throws AccountException {
               // Zu viele Schulden verboten
               if (balance + amount < maxDebt)
                     throw new AccountException(amount);
               // Zu hohe Einzahlung verboten
               if (amount > MAX_AMOUNT)
                     throw new AccountException(amount);
               balance += amount;
        }
}
```

```
public class AccountException extends Exception {
        // positiv bei Einzahlungen
        // negativ bei Abhebungen
        private double invalidAmount;
        public AccountException(double invalidAmount) {
              this.invalidAmount = invalidAmount;
        }
        public double getInvalidAmount() {
              return invalidAmount;
        }
}
```

Ausnahmedeklaration und Überschreiben

- Wenn eine überschriebene Methode den Ausnahmetyp X deklariert, dann kann die überschreibende Methode
  - ganz auf die Deklaration des Ausnahmetyps X verzichten,
  - ebenfalls eine Ausnahme vom Typ X deklarieren oder
  - eine Ausnahme von einem Untertyp von X deklarieren.
- Die überschreibende Methode darf keine neuen Ausnahmetypen oder Obertypen hinzufügen.

```
public abstract class A {
        public abstract void m() throws Exception;
}
abstract class B extends A {
       public abstract void m(); // OK
}
abstract class C extends A {
       public abstract void m() throws NullPointerException; // OK
}
abstract class D extends A {
       public abstract void m() throws Throwable; // nicht OK
}
```

Funktionstest (Black-Box-Test)

- Testen aufgrund externer Betrachtung:
  - Beobachtbares Verhalten (z. B. Ausgaben) des Testobjekts wird bei bestimmten Eingaben betrachtet (innere Abläufe werden ignoriert).
  - Verhalten wird mit dem nach Spezifikation erwarteten verglichen.
  - Grundlage für die Wahl der Testfälle ist einzig die Aufgabenbeschreibg.
- Vorteile:
  - Testfälle können unabhängig von der Implementierung erstellt werden.
- Nachteile:
  - Mögliche kritische Pfade in der Implementierung werden nicht erkannt und werden daher vielleicht nicht getestet.
- Häufig: Überprüfung spezifizierter Funktionen mit typischen Werten (Äquivalenzklassentest) und Randwerten (Grenzwerttest)
  - z. B. sehr kleine oder große double-Werte („gerade noch gültig“), leere Benutzereingaben, …

Strukturtest (White-Box-Test)

- Testen aufgrund interner Betrachtung:
  - Programmablauf wird bei Eingaben betrachtet, die unter Kenntnis des inneren Ablaufs so gewählt sind, dass sie best. Testkriterien erfüllen.
  - Sinnvolles Vorgehen: wähle Testfallmenge, welche z. B. jede Anweisung, jeden Zweig (then/else), jeden Pfad etc. mind. einmal durchläuft.
  - Grundlage für die Wahl der Testfälle ist einzig der Programmcode (die Spezifikation dient höchstens noch der Ergebnisüberprüfung).
- Vorteile:
  - Alle (vorhandenen) Programmteile können getestet werden.
- Nachteile
  - Fehlende Pfade können nicht getestet werden (Fehlen wird im Allg. nur d. externe Betrachtung aufgedeckt).
  - Anzahl aller erforderlichen Testziele (z. B. Pfade durch ein Programm) ist im Allg. zu hoch → kein vollständiges Testen möglich.

Integrationstest, Regressionstest

Integrationstest (oft auch: „Gray-Box-Test“):

- dient zur Überprüfung der Schnittstellen zwischen den Modulen bzw. Komponenten und testet somit die Interaktion zwischen zwei oder mehreren Komponenten (welche einzeln für sich schon im vorangehenden Modultest geprüft worden sind).

Regressionstest:

- bei Änderungen (z. B. Fehlerkorrektur, Erweiterung) eines Programms sollten alle alten, von der Änderung nicht betroffenen Tests weiterhin funktionieren.
  - Idealerweise automatisiert: alle Tests mit deren Ergebnissen werden gespeichert; bei Weiterentwicklung des Programms werden sie wieder durchgeführt und mit den alten Ergebnissen verglichen.
  - Ziel: Test, ob durch die Änderungen neue Fehler eingeführt wurden.

Zusicherungen (engl. assertions)

- Während der Programmentwicklung ist es i.d.R. möglich, Bedingungen zu formulieren, die am Beginn oder am Ende einer Methode oder während der Ausführung einer Schleife gelten müssen, damit die Methode korrekt arbeitet (sog. Vor- bzw. Nachbedingung, Schleifeninvariante).
- Solche Bedingungen können als Behauptungen während der Programmentwicklung (genauer: in der Testphase) automatisiert mittels sog. Zusicherungen (engl. assertions) geprüft werden.
- Zusicherungen
  - boolesche Funktionen für Vor- und Nachbedingungen sowie Invarianten,
  - werden zur Laufzeit ausgewertet und machen sich im Fehlerfall durch eine Ausnahme bemerkbar,
  - stehen meist am Anfang oder Ende von Methoden,
  - sollten unbedingt das eigentliche Programm nicht beeinflussen,
  - decken Fehler vielleicht in der Testphase auf, werden vorher konzipiert.

Syntax und Semantik von Zusicherungen

- AssertStatement
  - assert Expression1
  - assert Expression1 : Expression2
- Expression1 ist dabei ein boolescher Ausdruck und Expression2 ein Fehlermeldungsausdruck (z. B. eine Fehlermeldung (String)).
- Semantik:
  - Falls Expression1 zu true ausgewertet wird, erfolgt keine weitere Aktion.
  - Sonst Auslösen einer Exception vom Typ java.lang.AssertionError; Fehlermeldungsausdruck Expression2 wird Ausnahmeobjekt übergeben und kann mittels getMessage() ausgelesen werden.

Aktivier- und Deaktivierbarkeit von Zusicherungen

- Zusicherungen sind prinzipiell mit if-Anweisungen nachbildbar.
- Aber: benötigter Code ist umfangreicher, nicht auf den ersten Blick erkennbar, dass es sich um eine Bedingungsprüfung handelt, if-Anweisungen sind nicht deaktivierbar.
- Verwendung von Zusicherungen ist aktivier- und deaktivierbar.
  - Ausführung ohne Zusicherungen (schneller, Standardeinstellung):
    - java -disableassertions MyProgram
    - java -da MyProgram
  - Ausführung mit Zusicherungen (langsamer, zum Testen):
    - java -enableassertions MyProgram
    - java –ea MyProgram

Vermeidung von Seiteneffekten

- Zusicherungen dienen während der Programmentwicklung zur Überprüfung, ob die verwendeten Methoden mit den korrekten Datenwerten arbeiten.
- Bei der Ausführung beim Anwender sollten sie i. d. R. nicht eingesetzt und korrekter Programmablauf sollte dort anders sichergestellt werden.
- Deshalb:
  - Zusicherungen sind aktivierbar und deaktivierbar.
  - Zusicherungen sollten niemals Seiteneffekte besitzen, die die normale Programmausführung beeinflussen

Benutzung von Zusicherungen

- Test von Nachbedingungen
  - Vor jeder return-Anweisung, über die eine Methode verlassen werden kann, bzw. vor letzter Anweisung einer Methode erfolgt Überprüfung von Nachbedingungen.
  - Beispiele:
    - Prüfung, ob berechneter Wert innerhalb eines definierten Intervalls liegt.
    - Ist eine Wertfolge wirklich sortiert?
    - Liefert eine Methode leereFeld das gewünschte Ergebnis?

```
private void leereFeld(ArrayList al) {
     // Operationen zum Leeren eines Feldes
     // ...
     assert al.isEmpty(): "Das Feld ist nicht leer";
}
```

- Prüfung des Kontrollflusses
  - Prüfung bei if-Anweisungen, ob bestimmter Zweig ausgeführt wird, der eigentlich nicht zur Ausführung kommen darf.
- bei switch-Anweisungen über default-Zweig testen, ob die verwendeten case-Zweige alle Fälle abdecken.
- Prüfung der Parameter
  - von öffentlichen Methoden
    - Falscher Aufruf ist nie unerwartet: rechne mit dem Schlimmsten!
    - Statt Zusicherungen sollte hier IllegalArgumentException (Unterklasse von RuntimeException) verwendet werden, damit Aufrufer angemessen reagieren können.
  - von geschützten Methoden (private, protected)
    - Falsche Parameter sind Fehler; daher während der Entwicklung Parameterprüfung oder Zustandstests mittels Zusicherungen.
    - Integrierte Überprüfung der übergebenen Parameterwerte und Auslösen von Exceptions im Fehlerfall während des echten Betriebs „verzichtbar“

Formale Verifikation mittels wp-Kalkül Grundidee

- Durch jeweils eine konkrete Belegung der in einem Programm(stück) enthaltenen Variablen ist ein konkreter Zustand dieses Programm(stück)s gegeben.
- Im Allgemeinen interessiert man sich für einen oder mehrere Zielzustände, in denen bestimmte Regeln bzgl. der Variablenwerte gelten (sog. Nachbedingung, engl. post-condition).
- Die Anweisungen des Programmstück(s) werden von der Nachbedingung ausgehend rückwärts analysiert und je nach Typ der Anweisung wird gefolgert, welche Bedingungen bzgl. der Variablenwerte vor der Anweisung mindestens gegolten haben müssen, damit die jeweilige Bedingung nach der Anweisung gerade noch gilt.
- Dieses Vorgehen wird schrittweise bis vor die erste Anweisung des Programm(stück)s wiederholt.
- Folgt die dort geltende Bedingung aus der sog. Vorbedingung (engl. pre-condition), dann arbeitet das Programm korrekt.

Programmzustand

- Ein konkrete Belegung dieser Variablen mit Werten bildet einen sog. Zustand dieses Programm(stück)s.
- Das kartesische Produkt der Wertemengen der Variablen bildet damit den Zustandsraum des Programm(stück)s.

Verifikation, partielle und totale Korrektheit

- Eine Verifikation prüft, ob ein Programm bestimmte sog. Zusicherungen erfüllt, d. h. ob die Nachbedingung Q einer Anweisungsfolge A nach Abarbeitung der Anweisungen aus der Vorbedingung P ableitbar ist.
- Eine Anweisungsfolge A heißt partiell korrekt, wenn die Ausführung von A in einem Zustand, der P genügt, beginnt und falls die Ausführung von A nach endlich vielen Schritten terminiert, dann ein Zustand erreicht wird, der Q erfüllt (Schreibweise: P {A} Q).
- Hinweis: der ganze Ausdruck P {A} Q ist eine logische Formel, die wahr oder falsch sein kann.
- Eine Anweisungsfolge A heißt total korrekt, wenn sie partiell korrekt ist und zusätzlich nachgewiesen ist, dass A immer terminiert (Schreibweise: {P} A {Q}).

Verifikation, partielle und totale Korrektheit

- Es seien I_D die Menge der möglichen Eingabedaten und O_D die Menge der möglichen Ausgabedaten eines Programms.
- Die Vorbedingung ist ein Prädikat P auf I_D
- Die Nachbedingung ist ein Prädikat Q auf $O_D \times I_D$
- Es sei A : $I_D \rightarrow O_D$ die Abbildung, die die Eingabedaten des Programms in die Ausgabedaten überführt. Annahme dabei: Algorithmus A terminiert!
- Notation:
  - Falls gilt: $\forall x \in I_D$: P(x) = true $\Rightarrow$ Q(A(x), x) = true dann schreibt man: {P} A {Q} (für totale Korrektheit v. A)
  - Das bedeutet: Für alle Startzustände x für die P gilt, gilt Q nach Ausführung von A.
- P und Q bilden die sog. prädikatenlogische Spezifikation.

Vor- und Nachbedingung am Beispiel

```
// x ist ungerade
// {P: (x % 2) == 1}
x = x + 1;
// x ist gerade
// {Q: (x % 2) == 0}
```

- Nachbedingung Q (von uns gesetzt): x soll gerade sein.
- Aus Q und der Anweisung folgt, dass (x – 1) % 2 = 0 vor der Anweisung gelten muss, woraus direkt P folgt.
- Resultierende Vorbedingung P: x muss vor der Anweisung ungerade gewesen sein.
- P , Q : Beispiele für sog. Prädikate; Prädikate sind hier Aussagen bzgl. im Programmstück enthaltener Variablen, die erfüllt oder nicht erfüllt sein können.

Vor- und Nachbedingung am Beispiel (II)

- $\{P: i \text{ beliebig}\}$ i=7; $\{Q: i = 7\}$
  - Nach endlich vielen (hier: einem) Schritt terminiert die Anweisung.
  - Wenn vor Ausführung der Zustand P erfüllt war, dann ist nach Ausführung der Zustand Q erfüllt.
- $\{P: i \text{ beliebig}, j \text{ beliebig}\}$ j=2; i=7; $\{Q: i = 7 \wedge j = 2\}$
- $\{P: i \text{ beliebig}, j \text{ beliebig}\}$ i=2; i=7; $\{Q: i = 7 \wedge j \text{ beliebig}\}$
  - j hier in P und Q beliebig: will man etwas über j wissen?
  - falls nein: $\{P: \text{wahr}\}$ i=2; i=7; $\{Q: i = 7\}$
- Also: man muss nicht nur bedenken, was die einzelne Anweisung bewirkt, sondern auch, welche Teile der Vorbedingung in die Nachbedingung übernommen bzw. darin verändert werden.

Formalisierung: Verifikation, wp-Kalkül

- wp-Kalkül (wp: weakest precondition = schwächste Vorbedingung):
- Seien A und Q gegeben. Das schwächste Prädikat P (also das Prädikat, das bei möglichst vielen Eingabedaten x wahr ergibt),für das P { A} Q gilt, heißt $wp\ (A,Q) := P$.
- Das bedeutet:
  - Wähle also die Eingabewerte so, dass sie P erfüllen und wähle P so, dass es die schwächste Vorbedingung zu Q ist, die sich aus der Anweisungsfolge A ergibt.
  - Mit diesem Prädikat kann man also feststellen, für welche Eingabewerte das Programm richtig rechnet
- P heißt schwächer als P genau dann, wenn gilt: $P \Rightarrow P'$
- Idee: Untersuche statt vieler Vorbedingungen nur die schwächste Vorbedingung (weakest precondition), denn ausgehend von dieser wird die Nachbedingung „gerade noch“ erfüllt.

Einschub: einige logische Äquivalenzen

De Morgan‘sche Gesetze

- $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
- $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$

Distributivgesetze

- $A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$

Äquivalenz

- $(A \Leftrightarrow B) \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$

Implikation

- $(A \Rightarrow B) \Leftrightarrow \neg A \vee B$
- $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$

Vorgehen bei der Programmverifikation mit wp-Kalkül

- Das wp-Kalkül ist ein sog. Verifikationskalkül
- Verifikationskalküle
  - verwenden (rein syntaktische) Regelwerke zur Ableitung zu verifizierender Aussagen.
  - Regeln sind rein mechanisch anwendbar.
- Also:
  - Wir benötigen für jeden Anweisungstyp (z. B. Zuweisung, Fallunterscheidung, Schleife) eine Regel zur Herleitung der jeweils schwächsten Vorbedingung aus einer gegebenen Nachbedingung.
  - Dann lässt sich anhand dieser Regeln aus einer Nachbedingung einer Anweisungsfolge deren schwächste Vorbedingung bestimmen.
  - Da imperative Programme aus einer linearen Abfolge von Kontrollstrukturen bestehen, folgt aus der Korrektheit von aufeinander folgend ausgeführten Teilen die Korrektheit der Sequenz dieser Teile.

Zuweisungsregel

$wp(\text{"x := e"}, Q) = Q[x/e]$

- Vorgehensweise:
  - Zur Konstruktion der Vorbedingung wird mit der Nachbedingung Q begonnen, darin werden alle Vorkommen der Variablen x durch den Ausdruck e substituiert.
  - Man simuliert sozusagen den Effekt symbolisch.
- Hinweis: Die Nachbedingung Q kann nur erreicht werden, wenn sowohl x als auch e fehlerfrei ausgewertet werden können und deren Auswertung jeweils seiteneffektfrei bzgl. Q ist.

Beispiele zur Zuweisungsregel

- wp(" a := 7", a = 7) = (**7** = 7) = true
  - im Prädikat aa = 7 wird die Variable aa gemäß der Zuweisung a: = 7 durch 7 ersetzt
- wp(" a := 7" , a = 6) = (**7** = 6) = false
  - wenn das Prädikat a = 6 gelten soll, zuvor aber aa den Wert 7 zugewiesen bekommt, dann kann das Prädikat a = 6 nicht erfüllt werden
- wp(" a := 7" , b = 12) = (b = 12)
  - gilt auch allgemeiner: wp("a := 7", Q) = Q
  - Voraussetzung: das Prädikat Q enthält die Variable a nicht
  - das Prädikat Q wird durch die Zuweisung also nicht verändert
- wp(" a := b" , a > 1) = **b** > 1
  - wenn nach Ausführung von a: = b gelten soll: a > 1, dann muss vorher mindestens b > 1 gegolten haben.

```
public static int ganzzahligeQuadratWurzel(int r) {
        // {P: r >= 1}
        int s = 0, i = 0, x = 0;
        while (s <= r) {
               i++;
               s = s + (2*i - 1);
        }
        // {Q': (i-1)*(i-1) <= r && r < i*i}
        x = i - 1;
        // {Q: x*x <= r && r < (x+1)*(x+1)}
        return x;
}
```

Beispiel zur Verifikation: Zuweisung

- Wenn Vorbedingung P die Terminierung von A garantiert und wenn P { A } Q gilt, dann gilt auch: $P \Rightarrow wp(A,Q)$ und somit: {wp (A,Q) } A { Q }
- Beispiel:
  - zu beweisen: {P: 10 < y < 100} x := y {Q: x > 1}
  - Es gilt: wp("x := y", x > 1) = ($\boldsymbol{y}$ > 1)
    - Wenn nach Ausführung von x := y gelten soll: x > 1, dann muss vorher mindestens y > 1 gegolten haben.
    - Anders ausgedrückt: { wp("x := y", x > 1) } x := y { x > 1 }
  - Wegen (10 < y < 100) ⇒ (y > 1) gilt: $P \Rightarrow wp(A,Q)$

Sequenzenregel

$wp(\text{"}\mathtt{s_1;\ s_2}\text{"}, Q) = wp(\text{"}\mathtt{s_1}\text{"}, wp(\text{"}\mathtt{s_2}\text{"}, Q))$

- Vorgehensweise:
  - s1; s2 bedeutet: führe die Anweisung $s1$ aus und nach deren Terminierung führe die Anweisung $s2$ aus.
  - Q wird als Nachbedingung der Anweisung $s2$ betrachtet und deren schwächste Vorbedingung $wp(s2,Q)$ bestimmt.
  - Das dabei entstehende Prädikat wird dann zur Nachbedingung der Anweisung $s1$ und daraus deren schwächste Vorbedingung $wp(s1, wp(s2,Q))$ bestimmt.
- Hinweis: die Sequenzenregel gilt analog für längere Sequenzen
- Also: zwei (korrekte) Programmstücke s1 und s2 können zu einem (korrekten) Programmstück s1;s2 zusammengesetzt werden, wenn die Vorbedingung von s2 aus der Nachbedingung von s1 folgt

Beispiele zur Sequenzenregel

- Sei s1; s2 gegeben durch " a := a + b; b := a * b" und sei die Nachbedingung irgendein Prädikat $Q(a, b)$

$\square\ wp(s_2, Q(a,b))$
$= wp(\text{"}\mathtt{b := a * b}\text{"}, Q(a,b))$ — Zuweisungsregel
$= Q(a, a \cdot b)$
$\square\ wp(\text{"}\mathtt{s_1; s_2}\text{"}, Q(a,b))$ — Sequenzenregel
$= wp(s_1, wp(s_2, Q(a,b)))$
$= wp(s_1, Q(a, a \cdot b))$
$= wp(\text{"}\mathtt{a := a + b}\text{"}, Q(a, a \cdot b))$ — Zuweisungsregel
$= Q(a+b, (a+b) \cdot b)$

- Das bedeutet: wenn als Nachbedingung $Q(a, b)$ gelten soll, dann muss als Vorbedingung mindestens dieselbe Relation Q zwischen $a + b$ und $a + b \cdot b$ gegolten haben.

Beispiel zur Verifikation: Wertetausch von zwei Variablen

Vertauschen der Werte der Variablen x und y zu verifizieren: {P: $y = X \wedge x = Y$} h := x; x := y; y := h; {Q: $x = X \wedge y = Y$}

$wp(\text{"}\mathtt{h := x;\ x := y;\ y := h;}\text{"},\ x = X \wedge y = Y)$
$= wp(\text{"}\mathtt{h := x}\text{"}, wp(\text{"}\mathtt{x := y}\text{"}, wp(\text{"}\mathtt{y := h}\text{"}, x = X \wedge y = Y)))$
$= wp(\text{"}\mathtt{h := x}\text{"}, wp(\text{"}\mathtt{x := y}\text{"}, x = X \wedge h = Y))$
$= wp(\text{"}\mathtt{h := x}\text{"}, y = X \wedge h = Y)$
$= (y = X \wedge x = Y)$

Fallunterscheidung: if-Regel

- Die if-Regel lautet:
  - bei einer Fallunterscheidung mit then- und else-Teil:
    - wp("if b then s1 else s2",Q) = $b \wedge$ wp($s1$,Q) $\vee\ \neg b \wedge$ wp($s2$,Q)
  - bei einer Fallunterscheidung nur mit then-Teil:
    - wp("if b then s1", Q) = [$b \wedge$ wp($s1$,Q)] $\vee$[$\neg b \wedge$ Q]
- Erläuterung
  - Damit der then-Teil der if-Anweisung ausgeführt wird, muss jeweils vorher b true gewesen sein und damit nach der Anweisung $s1$ im then-Teil Q gilt, muss vorher mindestens $wp(s1,Q)$ gegolten haben.
  - Für den else-Teil muss b false gewesen sein und für die Anweisung $s2$ gilt Analoges wie für $s1$.
  - Bei der Fallunterscheidung nur mit then-Teil ist $s2$ die leere Anweisung.
  - b muss seiteneffektfrei bzgl. Q sein, z. B. nicht seiteneffektfrei wäre wp("if (--a > 0) then ..." , $a > 0$)

Beispiele zur if-Regel

Bestimmung des Maximums zweier Zahlen
Was muss vor der folgenden if-Anweisung mindestens gelten, damit nachher $\mathrm{max} = x$ erfüllt ist?

$wp(\text{"}\mathtt{if\ (x>y)\ then\ max := x\ else\ max := y}\text{"}.\ \mathrm{max} = x)$
$= (x > y \wedge wp(\text{"}\mathtt{max := x}\text{"}, max = x))$
$\quad \vee\ (x \leq y \wedge wp(\text{"}\mathtt{max := y}\text{"}, max = x))$
$= (x > y \wedge x = x) \vee (x \leq y \wedge y = x)$
$= (x > y \vee x = y)$
$= (x \geq y)$

Schleifeninvariante

- Ziel: Prüfung der Korrektheit einer Schleife der Form while b { A } (mit Bedingung b und Schleifenrumpf A)
- Gegeben:
  - Prädikate P,Q zur Spezifikation dessen, was die Schleife leisten soll
- Gesucht:
  - wp("while b { A }",Q) =?
  - Falls dieses Prädikat schwächer als P ist, dann ist das Programm korrekt.
- Definition:
  - Eine Invariante der Schleife: while (b) { A } ist ein Prädikat I, für das gilt: { $I \wedge b$ } A { I }
- Eigenschaften:
  - Wenn b erfüllt ist, dann muss unmittelbar vor dem Schleifenrumpf A auch b gelten; nach A (vor der nächsten Iteration) kann b gelten oder nicht.
  - Wenn vor dem Schleifenrumpf $I \wedge b$ gilt, dann ist I nach der Ausführung von A erfüllt; I ist also vor und nach dem Schleifenrumpf erfüllt, daher auch Bezeichnung als Invariante.
  - In wp-Terminologie muss man zeigen: $I \wedge b \Rightarrow wp(A,I)$ d. h.: $I \wedge b$ muss also stärker sein, als die schwächste Vorbedingung des Schleifenrumpfs.

Vorgehensweise

- Vier Schritte zum Nachweis der Korrektheit:
  - Beweise, dass I vor Eintritt in die Schleife gilt.
  - Beweise, dass { $I \wedge b$ } A { I } gilt, dass also I tatsächlich eine Invariante ist.
  - Beweise, dass ($I \wedge \neg\, b$) $\Rightarrow Q$, so dass bei Terminierung das gewünschte Ergebnis erreicht wird.
  - Beweise, dass die Schleife terminiert (totale Korrektheit) (mittels Terminierungsfunktion (auch: (sic!) Schleifenvariante)).

Hinweis zum Finden von Schleifeninvarianten

- Das Finden geeigneter Schleifeninvarianten ist ein kreativer und erfahrungsbasierter Prozess, für den es kein algorithmisches Verfahren gibt.
- Das Finden geeigneter Schleifeninvarianten ist daher ein Prozess, den man nur durch das Bearbeiten vieler Beispiele wirklich erlernen kann.
- Je genauer man verstanden hat, was eine Schleife wirklich leisten soll, desto leichter fällt es, eine geeignete Schleifeninvariante anzugeben.
- Oft hilfreiche Heuristik:
- Man betrachtet die Variablen, die in der Schleife verändert werden, und setzt diese in Relation zueinander bzw. zu den anderen Variablen (insbes. Eingaben) im Programm.
- Die Invariante muss stark genug sein, um die Nachbedingung daraus ableiten zu können, daher liefert die Nachbedingung gute Hinweise.

Schleifeninvariante finden

```
public static int ganzzahligeQuadratWurzel(int r) {
        int s = 0, i = 0, x = 0;
        while (s <= r) {
              i++;
              s = s + (2*i - 1);
        }
        x = i - 1;
        return x;
}
```

- Offenbar gilt in jeder Schleifeniteration: $i^2 = s$.
- Oben hatten wir bereits gezeigt, dass nach der Schleife gelten muss: $(i - 1)^2 \leq r \wedge r < i^2$
- Dies muss aus $I \wedge \neg b$ folgen: wählt man I zu schwach, gelingt der Nachweis $I \wedge \neg b \Rightarrow Q$ (folgt gleich) nicht.
- Verwende also: $I :\equiv i^2 = s \wedge (i - 1)^2 \leq r$

1. Beweise, dass I vor Eintritt in die Schleife gilt

$wp(\texttt{"s = 0; i = 0; x = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r))$ — Sequenzenregel

$= wp(\texttt{"s = 0"}, wp(\texttt{"i = 0"}, wp(\texttt{"x = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r))))$

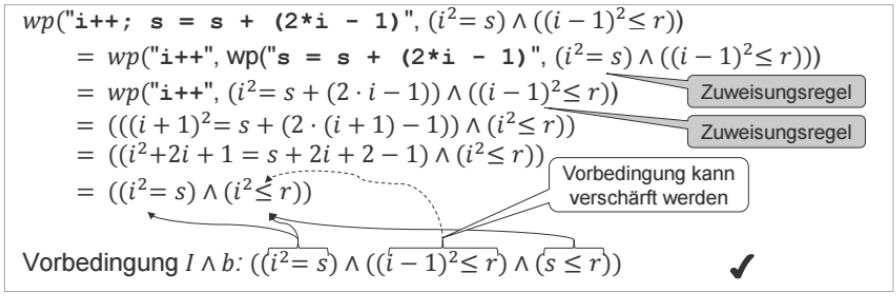
$= wp(\texttt{"s = 0"}, wp(\texttt{"i = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r)))$ — Zuweisungsregel

$= wp(\texttt{"s = 0"}, (0 = s) \wedge (1 \leq r))$ — Zuweisungsregel

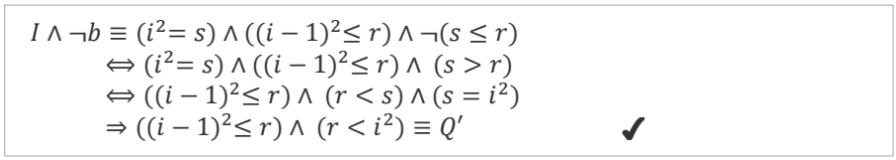
$= ((0 = 0) \wedge (1 \leq r))$ — Zuweisungsregel

$= (r \geq 1)$ ✓

2. Beweise, dass $\{ I \wedge b \}$ A $\{ I \}$ gilt



3.Beweise, dass $(I \wedge \neg b) \Rightarrow Q$



Letzter Schritt nur Implikation (⇒), da Q‘ keine Aussagen über s macht und daher der Rückwärtsschluss (⇐) von Q‘ (ohne s) auf die vorletzte Zeile (mit s) nicht zulässig ist.

Schleifenvariante (auch: Terminierungsfunktion)

- Die relevanten Eigenschaften der Schleifenvariante sind also:
  - V ist eine geeignete Funktion auf den in der Schleife verfügbaren Variablen.
  - V ist ganzzahlig.
  - V ist streng monoton fallend.
  - V ist nach unten beschränkt (meistens durch Konstante c = 0).
- Das Finden einer passenden Schleifenvariante ist ebenso wie das Finden einer Schleifeninvariante ein kreativer und erfahrungsbasierter Prozess, der erst durch viele erarbeitete Beispiele oder „trial-and-fail“ wirklich verstanden wird.

4. Beweis, dass die Schleife terminiert

- Mögliche Schleifenvariante: $V := r - (i - 1)^2$
- Zeige $I \Rightarrow V \geq 0$:
  - es gilt: $V := r - (i - 1)^2 \in \mathbb{Z}$ (VV ist also ganzzahlig), da r und i int-Variablen sind
  - aus I folgt, dass $(i - 1)^2 \leq r$ (für $r \geq 1$) und damit $V := r - (i - 1)^2 \geq 0$
  - also: $I \Rightarrow V \geq 0$:
- Zeige $\{I \wedge b \wedge V = z\}$ A $\{0 \leq V < z\}$ mit $V, z \in \mathbb{N}_0$:
  - Nachweisen, dass V streng monoton fallend ist: wenn V nach dem k-ten Schleifendurchlauf den Wert z hat, folgt aus der geforderten Monotonie, dass V nach dem nächsten $((k + 1)$-ten) Schleifendurchlauf kleiner als z sein muss.
  - Halte den Wert von V_k in z fest: $V_k := r - (i - 1^2) = r - i^2 + 2i - 1 =: z$
  - Für $k \geq 1$ ist $i \geq 1$ und damit auch $2i - 1 > 0$
  - Nach einmaliger Ausführung des Schleifenrumpfes i++; s = s + (2*i - 1);
  - ergibt sich $V_{k+1} := r - ((i + 1) - 1^2) = r - i^2 = z - (2i - 1) < z$
  - Also: V ist streng monoton fallend

Checklisten zur Fehleridentifizierung und -vermeidung

- Variablen- und Konstantendeklarationen
  - Werden beschreibende (sog. mnemonische) Variablen- und Konstantennamen verwendet, die mit den Codier-Regeln übereinstimmen?
  - Gibt es Variablen mit ähnlichen Namen, die verwechselt werden könnten?
  - Ist jede Variable sauber initialisiert?
  - Gibt es Instanzvariablen, die besser als lokale Variablen von Methoden deklariert werden sollten?
  - Gibt es Literale, die lieber als Konstanten deklariert werden sollten?
  - Gibt es Konstanten, die noch nicht als final deklariert sind?
  - Sind Laufvariablen von for-Schleifen im Schleifenkopf deklariert?
  - Haben alle Instanz- und Klassenvariablen eine möglichst restriktive Sichtbarkeit?
  - Gibt es Instanzvariablen (Attribute), die besser Klassenvariablen (static) wären oder umgekehrt?
- Methodendeklaration
  - Entsprechen die Methodennamen der Codier-Regel und sind sie mnemonisch?
  - Werden die Argumente vor ihrer Verwendung auf Plausibilität geprüft?
  - Gibt jede Methode bei jedem Rückkehrpunkt den beabsichtigten Wert zurück?

- Haben alle Methoden eine möglichst restriktive Sichtbarkeit?
- Gibt es Objektmethoden, die besser Klassenmethoden (static) wären oder umgekehrt?

- Klassendeklaration
  - Gibt es Konstruktoren, die alle Klassenvariablen direkt initialisieren?
  - Kann die Vererbungshierarchie vereinfacht werden?
  - Kann man (weitere) Variablen oder Methoden aus Unterklassen in Oberklassen verschieben?
- Datenzugriff
  - Liegen Zugriffe auf Arrays (Reihungen) im gültigen Bereich und werden auch die Randwerte bearbeitet?
  - Ist für jeden Objekt-Zugriff gesichert, dass die Referenz „!= null“ ist?
  - Können Zahlenbereiche über- oder unterlaufen?
  - Stimmen bei Ausdrücken mit mehreren Operatoren die Annahmen über die Präzedenzregeln (Vorrangregeln)?
  - Stimmt die Klammerung?
- Vergleiche, boolean-Operatoren
  - Gibt es unbeabsichtigte Seiteneffekte?
  - Lässt sich der Ausdruck vereinfachen?
- Kontrollfluss
  - Ist die jeweils beste Schleifenanweisung gewählt?
  - Terminiert jede Schleife?
  - Wenn es mehrere Schleifenausgänge gibt, ist jeder korrekt?
  - Hat jede switch-Anweisung einen default-Fall?
  - Sind fehlende break-Anweisungen im switch gewollt und korrekt kommentiert?
  - Bei break und continue mit Label: stimmt das Ziel?
  - Übertriebene Verschachtelungstiefe (Fallunterscheidung, Schleife)?
  - Könnte verschachtelte if-Anweisung durch switch ersetzt werden?
  - Sind leere Rümpfe korrekt und wenn ja, sind sie mit {} markiert?
- Ein-/Ausgabe
  - Werden alle Dateien vor der Verwendung geöffnet?
  - Werden alle Dateien nach der Verwendung wieder geschlossen?
  - Gibt es Rechtschreib- oder Grammatikfehler in angezeigtem oder gedrucktem Text?
  - Werden alle Ausnahmen sinnvoll verarbeitet?
- Schnittstellen
  - Stimmen die Anzahl, Reihenfolge, Typen und Werte von Methodenargumenten mit der Deklaration überein?
  - Stimmen die Einheiten (Meter, Zentimeter, …)?
  - Bei Übergabe von Objekten: gibt es Seiteneffekte? Werden diese beim Aufrufer bedacht?
- Kommentare
  - Hat jede Quelldatei, jede Klasse und jede Methode, jede Konstante einen Kommentar?
  - Wird das Verhalten (gegeben, gesucht, prinzipieller Ansatz) im Kommentar beschrieben?
  - Passen die Kommentare (noch) zum Quelltext?
  - Sind die Kommentare hilfreich für das Verstehen des Quelltextes?
  - Sind die richtigen Kommentare im javadoc-Kommentar für die htmlSeiten und die richtigen Kommentare intern?
- Umfang von Quelldateien und Methoden
  - Sind die Quelldateien kürzer als (z.B.) 2.000 Zeilen?
  - Sind Methoden kürzer als (z.B.) 60 Zeilen?
  - Hinweis: konkrete Zahlenangaben sind projekt-/problemabhängig
- Modularität
  - Hängen die Klassen eines Pakets inhaltlich eng zusammen?
  - Gibt es replizierten Code, den man entfernen könnte?
  - Gibt es mehrere Stellen im Code, an denen dasselbe getan wird, so dass diese Funktionalität zu einer Methode zusammengefasst werden könnte?
  - Wurde aus Unkenntnis der Standard-Bibliothek Funktionalität selbst implementiert?
- Speichereffizienz
  - Sind Arrays (Reihungen) übertrieben groß deklariert?
  - Werden Referenzen auf nicht mehr benötigte Objekte auf null gesetzt, damit der Speicherbereiniger (Garbage Collector) den Platz frei geben kann?
- Leistungsdefekte
  - Können effizientere Datenstrukturen und/oder Algorithmen benutzt werden?
  - Sind logische Tests im Code so angeordnet, dass die oft erfolgreicheren und billigeren Tests vor den teureren und weniger häufigen Tests durchgeführt werden?
  - Können die Kosten für die Wiederberechnung eines Wertes reduziert werden, indem die Berechnung nur einmal ausgeführt und der Wert zwischengespeichert wird?
  - Wird jedes berechnete Ergebnis auch tatsächlich benötigt?

Grundlegende Datentypen

Motivation

- Szenario: arbeitsteilige Entwicklung großer Software-Systeme im Team.
- Zerlegung solcher Software-Systeme in logisch und funktional zusammengehörige Module (oder: Komponenten).
- Module besitzen Schnittstellen bestehend aus Signaturen der Operationen und Zusicherungen (nicht im Java interface) über diese Operationen, Zweck:
  - Verwendungssicht
    - Andere können Modul allein bei Kenntnis der Schnittstelle benutzen.
    - Kenntnis der Implementierung ist nicht erforderlich.
  - Implementierungssicht
    - Modul kann allein bei Kenntnis der Schnittstelle implementiert werden.
    - Kenntnis seiner Verwendungen nicht erforderlich.
- Frage: Wie kann die Schnittstelle eines (zu entwickelnden) Moduls (eines Typs) ohne Angabe einer konkreten Implementierung vorab so spezifiziert werden, dass dessen korrekte Verwendung bzw. Implementierung möglich sind?
- Lösung: Spezifikation als sog. Abstrakter Datentyp (ADT)
  - Festlegung der auf den Typ anwendbaren Operationen (Schnittstelle)
  - Festlegung der Wirkung der Operationen (Zusicherungen)
- Vorteile:
  - Nutzende Komponenten können sich auf die Spezifikation der Wirkung verlassen, ohne die Implementierung zu kennen, sog. Design by Contract.
  - Die Implementierung einer Komponente ist für die Nutzer geheim. Sie kann ausgetauscht/verbessert werden, ohne dass die Nutzer betroffen sind, solange die spezifizierte Wirkung der Operationen erhalten bleibt, sog. Geheimnisprinzip

Ein erstes Beispiel: Menge ganzer Zahlen

- Aufgabe: Verwalte eine Menge ganzer Zahlen so, dass Zahlen eingefügt und gelöscht werden können und der Test auf Enthaltensein durchgeführt werden kann.
- Ebene der Spezifikation:
  - Abstrakteste Sicht eines Moduls: es gibt Objekte bestimmter Sorte, auf die Operationen anwendbar sind.
  - Ggfs. können sogar mehrere Sorten von Objekten eine Rolle spielen.
  - Operationen erzeugen aus gegebenen Objekten dieser Sorten neue Objekte, die ebenfalls zu einer der Sorten gehören.
- Im Beispiel:
  - Sorten: Mengen ganzer Zahlen, ganze Zahlen selbst, Wahrheitswerte.
  - Operationen: Zahlen einfügen und löschen, Test auf Enthaltensein.
- System bestehend aus Objektsorten mit dazugehörigen Operationen bezeichnet man als Datentyp.

Festlegung der Syntax: Signatur

- Um Datentyp zu beschreiben, muss man festlegen:
  - wie die Objektsorten und Operationen heißen,
  - wie viele und was für Objekte die Operationen als Argumente benötigen und
  - welche Sorte von Objekt sie als Ergebnis liefern.
- Rein syntaktischer Aspekt, wird festgelegt durch sog. Signatur:

adt $IntSet$

sorts $IntSet, Int, Boolean$

ops

$create$:		$\rightarrow IntSet$
$insert$:	$IntSet \times Int$	$\rightarrow IntSet$
$delete$:	$IntSet \times Int$	$\rightarrow IntSet$
$contains$:	$IntSet \times Int$	$\rightarrow Boolean$
$isEmpty$:	$IntSet$	$\rightarrow Boolean$

Kein Argument, liefert stets ein leeres $IntSet$; 0-stellige Funktion wird auch als ***Konstante*** bezeichnet.

Festlegung der Semantik/Wirkung von Operationen: Axiome

- Spezifikation als Abstrakter Datentyp (ADT) Beschreibung der Semantik/Wirkung von Operationen durch Angabe interessierender Aspekte der Wirkungsweise, Gesetze oder Axiome genannt.
- Gesetze: Gleichungen über Ausdrücken, die mit Hilfe der Operationssymbole gebildet werden; darin vorkommende Variablen sind implizit allquantifiziert
-

axs

$$
\begin{aligned}
&isEmpty(create) && = true\\
&isEmpty(insert(x,i)) && = false\\
&insert(insert(x,i),i) && = insert(x,i)\\
&contains(create,i) && = false\\
&contains(insert(x,j),i) && = \begin{cases} true & falls\ i = j \\ contains(x,i) & sonst \end{cases}
\end{aligned}
$$

end $IntSet$

Für alle x aus $IntSet$ und für alle i aus Int stehen links und rechts vom = gleiche Objekte. Intuitiv: mehrfaches Einfügen von i verändert die Menge x nicht.

Bewertung der Spezifikation als Abstrakter Datentyp

- Vorteile
  - Man kann Datentyp genau soweit festlegen, wie gewünscht, ohne z. B. Implementierungsdetails vorwegzunehmen.
  - Die Sprache, in der Axiome beschrieben werden, folgt formalen Regeln. Daher sind Entwurfswerkzeuge konstruierbar, die Spezifikation prüfen oder Prototyp erzeugen können.
- Nachteile
  - Bei komplexen Anwendungen wird die Zahl der Gesetze sehr groß.
  - Es ist nicht immer leicht,
  - anhand der Gesetze die intuitive Bedeutung eines Datentyps zu erkennen,
  - eine Datenstruktur anhand von Gesetzen zu charakterisieren; insbesondere ist es schwierig zu prüfen, ob die Menge der Gesetze vollständig ist.

Weitere ADT-Begriffe

- Konstruktoren: mit create und insert lassen sich alle möglichen Objekte des Typs *I*ntSet erzeugen. Beide sind zwingend erforderlich.
- Hilfskonstruktoren: erzeugen auch Datenobjekte des Typs; für Konstruktion der Objekte aber nicht zwingend erforderlich.
- Normalform eines Datenobjekts: konstruiert durch eine minimale Zahl von Konstruktoraufrufen (keine Hilfskonstruktoren).

Implementierung des ADTs IntSet

Interface der öffentlichen Operationen des ADTs *I*ntSet

```
import static java.lang.System.arraycopy;
import static java.util.Arrays.copyOf;
public interface IntSet {
        public void insert(int elem);
        public void delete(int elem);
        public boolean contains(int elem);
        public boolean isEmpty();
}
```

- Eine mögliche Implementierung als Klasse MyIntSet:
  - Verwendung von Arrays, arraycopy und copyOf.
  - create wird durch Konstruktor realisiert.
  - insert erzeugt größeres Array und fügt Element hinten an.
  - delete lässt weg und „klebt“ Teil-Arrays (vorher/danach) zusammen

```
class MyIntSet implements IntSet {
        private int[] is = new int[0];
        //public MyIntSet() { }
        public void insert(int elem) {
                if (!contains(elem)) {
                        is = copyOf(is, is.length + 1);
                        is[is.length - 1] = elem;
                }
        }
        public boolean contains(int elem) {
                for (int i = 0; i < is.length; i++)
                        if (is[i] == elem) return true;
                return false;
 }
```

- Konstruktor realisiert zusammen mit leerem Array is die im ADT spezifizierte empty-Funktionalität.
- Mengen enthalten keine Duplikate, deshalb nur einfügen, wenn elem nicht schon enthalten.

```
        public boolean isEmpty() {
                return is.length == 0;
        }
        public void delete(int elem) {
                if (contains(elem)) {
                        int[] isNew = new int[is.length - 1];
                        for (int p = 0; p < is.length; p++) {
                                if (is[p] == elem) {
                                        arraycopy(is, 0, isNew, 0, p);
                                        arraycopy(is, p+1, isNew, p, is.length-1-p);
                                        is = isNew;
                                        return;
} } } } }
```

Behälterdatentypen

- Aufgrund ihrer großer praktischer Relevanz betrachten wir im Folgenden Behälterdatentypen (engl. container types)
  - Datentypen zur Speicherung, Organisation und Verwaltung von Objekten eines Elementdatentyps.
  - Typ der Elemente im Behälter wird zum Parameter des ADTs.
  - Beispiele: Verwaltung von Daten in beliebigen Anwendungskontexten

Statische Datenstrukturen

- Um mit Mengen von Datenelementen umzugehen, haben wir bislang ein- oder mehrdimensionale Arrays benutzt.
- Array ist ein Beispiel für sog. statische Datenstrukturen, d. h. Größe der Struktur wird zum Erstellungszeitpunkt festgelegt und kann dann zur Laufzeit nicht mehr geändert werden.
- Nachteile:
  - Falls das Array zu klein ist, muss man wie in obigem Beispiel Code zum Anlegen größerer Arrays und zum Kopieren in diese schreiben. Das verlangsamt die Laufzeit.
  - Falls das Array zu groß ist, wird unnötig Platz verschwendet; dadurch werden evtl. andere Programme behindert, d. h. diese können nicht laufen oder laufen langsam. (Oben: Erzeugung eines kleineren Arrays und langsames Umkopieren.)
  - Falls Elemente in bestimmter Reihenfolge gehalten werden sollen, dann erfordert das Einfügen oder Löschen von Elementen in einem Array Kopierarbeit. (Oben: unsortierte Menge.)

Dynamische Datenstrukturen

- Anderer Ansatz:
  - Verkettung von Datenelementen, die jeweils ihren direkten Nachfolger (und ggf. Vorgänger) kennen (z. B. sog. Liste).
  - Dann ist an beliebiger Stelle Einfügen oder Entfernen von Elementen möglich, ohne dabei andere Elemente verlagern (kopieren) zu müssen.
  - Direkter Zugriff auf das i-te Element ist nicht möglich (im Unterschied zu Arrays, dort Direktzugriff in O(1) möglich). Die Liste muss von vorne beginnend bis zum i-ten Element linear durchlaufen werden, O(n).
- Liste ist Beispiel für sog. dynamische Datenstrukturen
  - können während des Programmablaufs beliebig wachsen oder schrumpfen,
  - enthalten nicht mehr Elemente als notwendig und
  - werden in Java mit Referenztypen implementiert.

Motivation für generische/parametrisierte Klassen

Beispiel: (rudimentärer) Container, der nur einen Wert speichern kann

```
public class Container {
        private int value;
        public int getValue() {
              return value;
        }
        public void setValue(int n){
              value = n;
} }
```

```
public class Container<T> {
        private T value;
        public T getValue() {
              return value;
        }
        public void setValue(T n) {
               value = n;
} }
```

- Im Code wurden alle Vorkommen von int, die Datentyp des zu verwaltenden Wertes angeben, durch T ersetzt.
- Durch Angabe des Typplatzhalters T in spitzen Klammern wird Klasse zur generischen bzw. parametrisierten Klasse

Verwendung generischer/parametrisierter Klassen

- Verwendungsbeispiele:

```
// Containter fuer Integer-Zahlen
Container<Integer> i =  new Container<Integer>();
Integer j = new Integer(3);
i.setValue(j);
// Container fuer Gleitkommazahlen
Container<Double> d =  new Container<Double>();
d.setValue(new Double(0.2));
Double d2 = d.getValue();
System.out.println(d2);
```

- Java verlangt, dass bei der „Instanziierung“ für den Platzhalter ein Klassentyp eingesetzt wird.
- Also für primitive Datentypen Wrapper-Klassen verwenden

Deklaration generischer/parametrisierter Klassen

```
public class MyClass<T, U> {
        private T variable1;
        public U calc(T t, U u) {
              ...
              return u;
        }
}
```

- Mehrere Typparameter werden durch Kommata getrennt angegeben.
- Typparameter können innerhalb der Klasse als Typen von Variablen oder Rückgabewerten oder als Parameter von Methoden genutzt werden
- Typparameter dürfen nicht in statischen Elementen verwendet werden, da verschiedene „Instanzen“ einer Klasse mit unterschiedlichen generischen Typen dennoch dieselben statischen Elemente verwenden.

```
public class MySpecialClass<E> {
        private E f;
        ...

        public void doThis(E h) {
              f = h;
              MySpecialClass<E> g = new MySpecialClass<E>();

              E e = new E();
              f.doThat();
              f.toString();
        }
}
```

- Referenzen auf Objekte vom Typ E erlaubt
- E darf als Typparameter für eine Instanzierung der parametrisierten Klasse verwendet werden
- Konstruktion eines Objekts nicht erlaubt, da Laufzeittyp unbekannt
- Aufruf spezieller Methoden der generischen Klasse nicht erlaubt, da Laufzeittyp unbekannt

```
public class ChildClass<E> extends MySpecialClass<E> {
       MySpecialClass<MyInt> c = new ChildClass<MyInt>();
       MySpecialClass<Object> o = c;
 …
```

- Polymorphie gilt auch für getypte Klassen,aber nicht für den Typparameter selbst
- Konvention für Namen der Typparameter:
    - Einzelne Großbuchstaben
    - T – Typ, K – Key, N – Number, E – Element, V – Value, S, U, V, …

Methoden und Typparameter

```
public class MethodsAndTypes {
        public static <T> Pair<T,T> duplicate(T elem) {
               return new Pair<T,T>(elem, elem);
        }
        public static void method(Container<String> elem) { … }
        public static void method(Container<Double> elem) { … }
}
```

- Typparameter können auch bei Methoden angegeben werden
- Überladen von Methoden, deren Signaturen sich nur im generischen Typ unterscheiden ist nicht erlaubt, da der Typparameter nur zur Übersetzungsaber nicht zur Laufzeit bekannt ist.

Erzeugung generischer Objekte

```
public static void main(String[] args) {
        Container<Integer> c = new Container<Integer>();
        Container<Integer> c2 = new Container<>();
        ComplexContainer<Container<Double>, Integer> c3 =
               new ComplexContainer<>();
}
```

- Sogenannter DiamondOperator <> erlaubt verkürzte Schreibweise

```
public class MyArray<T> {
        private T[] elements;
        public MyArray() {
               elements = new T[1000];
        }
        public void add(T e) { … }
}
```

- MyArray.java:5: error: generic array creation Erzeugung eines generischen Arrays nicht ohne Fehler/Warnungen möglich, da der Typparameter nur zur Übersetzungs- aber nicht zur Laufzeit bekannt ist.

Platzhalter / Beschränkungen

```
static void print(Container<?> c) {
 Object o = c.getValue();
 System.out.println(o);
}
```

- ? ist Platzhalter für beliebige Typen

```
static <T extends Number> double sum(T t1, T t2) {
       return t1.doubleValue() + t2.doubleValue();
}
```

- Number ist Typ-Obergrenze. T ist Unterklasse von Number und stellt damit auf jeden Fall doubleValue() bereit..

```
class Car { … }
class Audi extends Car { … }
class A6 extends Audi { … }
insertAudi(new MyArray<Audi>); //ok
insertAudi(new MyArray<Car>); //ok
insertAudi(new MyArray<A6>); //zu speziell
static void insertAudi(MyArray<? super Audi> array) {
        array.add(new A6());
        array.add(new Car());
}
```

- Da insertAudi als Typ-Untergrenze mit Audi aufgerufen werden könnte, kann nicht sicher in jedem Fall ein Car hinzugefügt werden

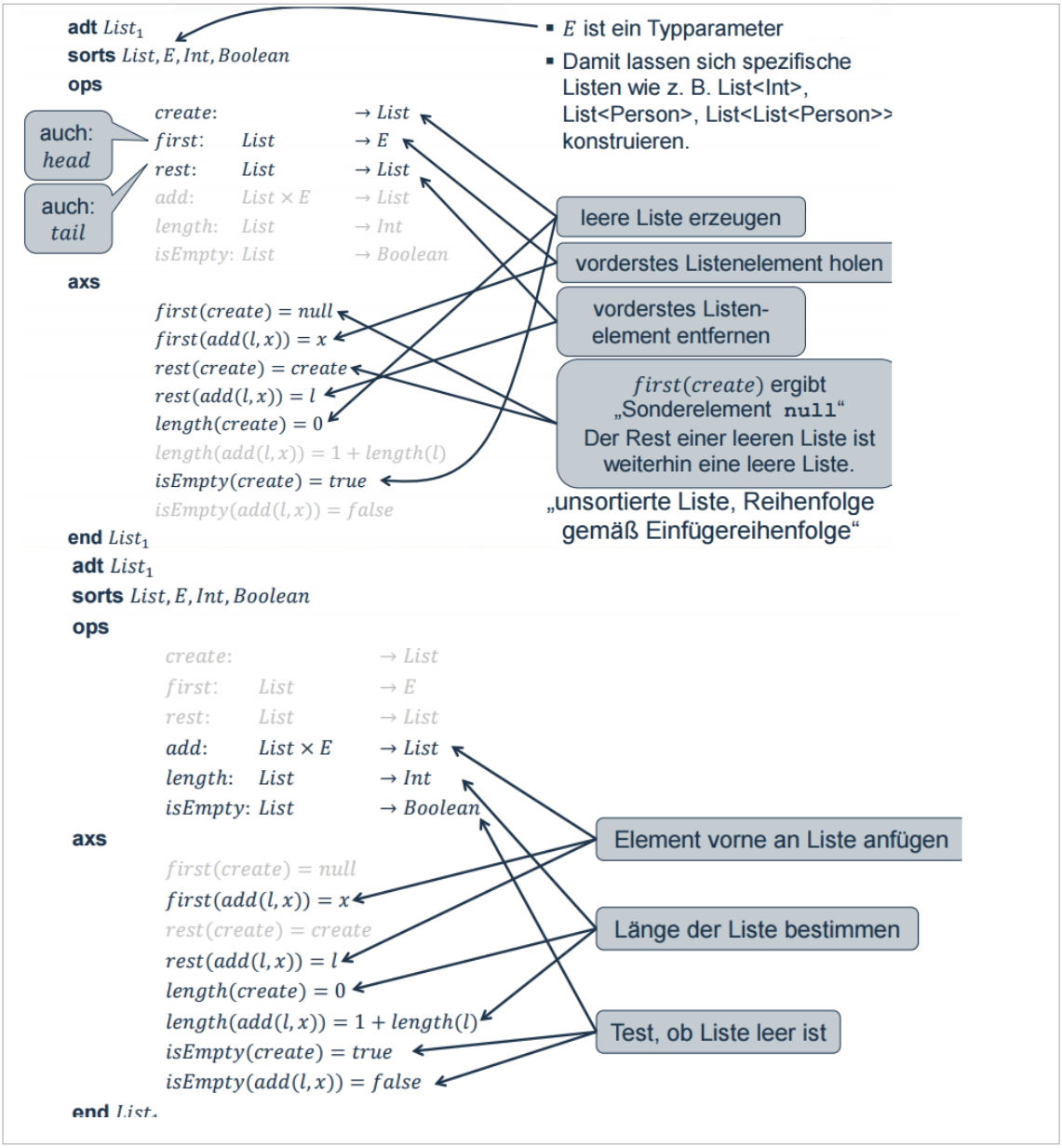
Eigenschaften generischer/parametrisierter Klassen

- Generische bzw. parametrisierte Klassen bieten sich an, wenn man feststellt, dass man mehrere Klassen für verschiedene Typen, aber mit ansonsten identischem Code benötigt.
- Vorteile:
  - Code muss nur einmal geschrieben werden.
  - Platzhalter kann durch einen beliebigen Typ ersetzt werde,der so verwendet werden kann, wie der Platzhalter im Code der generischen/parametrisierten Klasse.

Listen

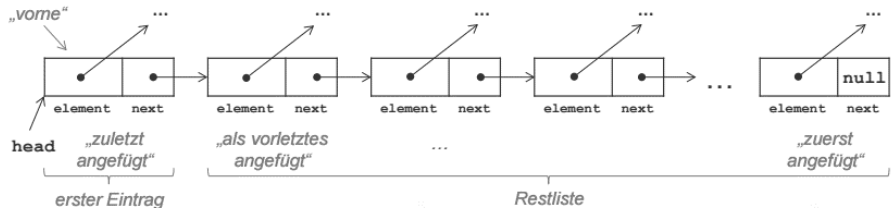
- Listen
  - enthalten endliche Folgen von Objekten eines gegebenen Grundtyps,
  - in einer definierten Ordnung: erstes, zweites … Element.
- Jedes Element hat Vorgänger und Nachfolger (außer dem ersten und letzten Element).
- Listen dürfen Duplikate enthalten.
- Man unterscheidet unsortierte und sortierte Listen (bzgl. einer Ordnung des Grundtyps).
- Listen unterscheiden sich bzgl. der angebotenen Operationen:
  - Hier zunächst elementare (unsortierte) Liste bei der Anfügen und Entfernen nur an einem Ende der Liste möglich ist (ADT $List1$).
  - Später andere Listentypen mit mehr Komfort. (Einfügen und Entfernen an beliebiger Stelle (am Anfang, in der Mitte, am Ende), Sortierung der Listenelemente, und/oder Verketten zweier Listen, Zugriff auf Anfangs- /End-/Teillisten, …)

Elementare Listen mit first, rest, add



Implementierung des ADTs *List*1 mit einfach verketteter Liste

Liste besteht aus verketteten Listeneintrags-Objekten, die jeweils „vorne“ angefügt werden:



Vorderstes Listenelement holen

```
public class List<E> {
        Entry<E> head;
        List() {…}

        public E first() {
              if (head != null)
                      return head.element;
              else
                     return null;
        }
        ...
```

```
class Entry<E> {
        E element;
        Entry<E> next;
}
```

Vorderstes Listenelement entfernen

```
public void rest() {
        if (head != null) {
              head = head.next;
        }
 }
 ...
```

Element vorne an Liste anfügen

```
public List<E> add(E newEl) {
        Entry<E> newEntr = new Entry<E>();
        newEntr.element = newEl;
        newEntr.next = head;
        head = newEntr;
        return this;
 }
 public boolean isEmpty() {
        return (head == null);
 }
```

Länge der Liste bestimmen (nach Entrekursivierung iterativ)

```
//iterative Fassung
 public int length() {
        int len = 0;
        Entry<E> e = head;
        while (e != null) {
              e = e.next;
              len++;
        }
        return len;
 }
}
```

Stapel/Keller (engl. Stack)

- Ein Stapel/Keller (engl. Stack) ist ein Spezialfall der elementaren Liste (die dem Datentyp *List*1 entspricht).
- Operationen werden hier traditionell anders benannt: top = first, push = add, pop = rest, stack = List
- Stapelprinzip (bekannt z.B. von einigen Kartenspielen und von den Methodenschachteln (siehe LE 3)):
  - Man kann nur oben etwas drauflegen oder herunternehmen. Nur oberstes Element ist sichtbar.
  - Zuletzt eingefügtes Element wird als erstes entnommen. Stapel sind sog. last-in-first-out (LIFO) Datenstrukturen.

Abstrakter Datentyp **Stack**

adt *Stack*
sorts *Stack*, *E*, *Boolean*
ops

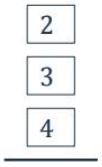
create:		→ *Stack*	← Konstruktoren
push:	*Stack* × *E*	→ *Stack*	← Konstruktoren
pop:	*Stack*	→ *Stack*	← Hilfskonstruktor
top:	*Stack*	→ *E*	← Projektionen
isEmpty:	*Stack*	→ *Boolean*	← Projektionen

axs

$isEmpty(create)$	$= true$
$isEmpty(push(s,e))$	$= false$
$pop(create)$	$= create$
$pop(push(s,e))$	$= s$
$top(create)$	$= null$
$top(push(s,e))$	$= e$

end *Stack*

Beispiel:
$push(push(push(create, 4), 3), 2)$



Implementierung des ADTs Stack mit Liste

- ADT Stack kann leicht mit Implementierungen des Datentyps $\mathit{List}1$ realisiert werden.
- Gleich lautende Operationen der Klasse List werden direkt verwendet.
- Gemäß der Namenskonventionen: top = first, push = add, pop = rest, stack = List werden die zugeordneten Listenoperationen verwendet, um die Stapel-Operationen zu realisieren.
- Grundsätzlich kann man mit jeder korrekt implementierten Liste einen Stapel implementieren.

Klasse java.util.Stack<E> in Java-Standardbibliothek

- Java stellt eine Stack-Implementierung bereit mit geringfügigen Unterschieden in der Benennung und Funktionalität der Methoden:
  - boolean empty() prüft, ob dieser Stack leer ist.
  - E peek() schaut auf das Objekt, das oben auf dem Stapel liegt, ohne es davon zu entfernen (entspricht top() in unserem ADT).
  - E pop() entfernt das oben liegende Objekt vom Stapel und gibt es als Ergebnis zurück (entspricht top() und pop() in unserem ADT).
  - E push(E item) legt das Element auf den Stapel.
  - int search(Object o) liefert die Position des Objekts auf dem Stapel (nicht Bestandteil einer reinen Stapel-Semantik).

(Warte-) Schlangen (engl. Queues)

- (Warte-) Schlangen (engl. Queues) sind spezielle Listen,
  - bei denen Elemente nur an einem Ende („vorne“) entnommen
  - und nur am anderen Ende („hinten“) angehängt werden.
- Schlangen entsprechen unserem Datentyp $\mathit{List}1$
  - ohne die Operation add
  - dafür mit einer Operation append (append),die ein Element ans Ende einer Liste hängt.
- Traditionell gibt es auch hier wieder andere Bezeichnungen:
  - Front = first, enqueue = append, dequeue = rest, Queue = List
- Schlangen sind sog. First-In-First-Out (FIFO) Datenstrukturen.

Abstrakter Datentyp *Queue*

adt *Queue*
sorts *Queue, E, Boolean*
ops

create:		$\rightarrow Queue$
front:	*Queue*	$\rightarrow E$
enqueue:	*Queue* × *E*	$\rightarrow Queue$
dequeue:	*Queue*	$\rightarrow Queue$
isEmpty:	*Queue*	$\rightarrow Boolean$

axs

$$dequeue(create) = create$$

$$dequeue(enqueue(q,x)) = \begin{cases} q & \text{falls } isEmpty(q) \\ enqueue(dequeue(q),x) & \text{sonst} \end{cases}$$

$$front(create) = null$$

$$front(enqueue(q,x)) = \begin{cases} x & \text{falls } isEmpty(q) \\ front(q) & \text{sonst} \end{cases}$$

$$isEmpty(create) = true$$

$$isEmpty(enqueue(q,x)) = false$$

$$\ldots$$

end *Queue*

Normalform:
- Nur *create* und *enqueue* sind Konstruktoren.
- Jede Schlange lässt sich nur mit diesen 2 Operationen erzeugen.

Für nicht-leere Schlange ist es unerheblich, ob man zuerst ein Element anhängt und dann eines wegnimmt oder umgekehrt.

Das vorderste (älteste) Element einer Schlange ist unabhängig davon, ob weitere Elemente mittels *enqueue* angefügt sind.

Implementierung des ADTs Queue mit Liste

- Schlangen werden mit zusätzlichem Zeiger auf das Schlusselement implementiert (für effizientes Anhängen am Ende).
- Leere Schlange: Zeiger auf erstes und letztes Element gleich null.
- Einelementige Schlange: Zeiger auf erstes und letztes Element zeigen auf dasselbe einzige Schlangenelement.
- enqueue:
  - next-Verweis des bisherigen Ende-Elements auf das neue Element zeigen lassen.
  - Dann Ende-Verweis der Schlange auf das neue Element setzen.
- dequeue:
  - Kopf-Verweis auf den Nachfolger des aktuellen Kopfs versetzen.
  - Achtung: Wenn die Schlange nur ein Element hatte, dann zeigt der
    - Ende-Verweis danach auf das entfernte Element.
    - Ende-Verweis muss ebenfalls auf null gesetzt werden

Implementierung des ADTs Queue mit Array

- Array-Implementierung ist eine Alternative bei bekannter maximaler Schlangenlänge.
- Durch Einfügen und Entfernen an verschiedenen Enden „durchwandert“ die Schlange das Array und stößt bald an einem Ende an, obwohl Gesamtgröße des Arrays durchaus ausreicht (wenn chlange höchstens so viele Elemente enthält, wie Array lang ist).
- Trick:
  - Array als zyklisch auffassen.
  - Wenn Ende erreicht, vorne wieder anfangen, sofern dort noch Plätze frei sind.
- Auf das Array-Element mit Index n-1 folgt logisch das Array-Element mit Index 0.
- Die Implementierung muss sicherstellen, dass die Schlange nicht länger als die maximale Schlangenlänge wird, damit das Ende der Schlange im zyklischen Array nicht den Kopf der Schlange überschreib

Prioritätswarteschlange (engl. Priority Queue)

- Die Elemente der Schlange erhalten zusätzlich eine Priorität.
- Beim Auslesen wird nicht das am längsten in der Schlange befindliche Element (FIFO) geliefert, sondern das Element höchster Priorität, das am längsten in der Schlange war.
- Beispiele:
  - Warteraum in Krankenhaus-Ambulanz: Patienten in akuter Gefahr werden vorgezogen.
  - Betriebssysteme: Prozesse (laufende Programme) besitzen unterschiedliche Priorität und erhalten entsprechend Ressourcen (z. B. CPU-Zeit).
- Signatur des abstrakten Datentyps $PQueue$

create:		$\rightarrow PQueue$
front:	*PQueue*	$\rightarrow (E \times Int)$
enqueue:	*PQueue* × $(E \times Int)$	$\rightarrow PQueue$
dequeue:	*PQueue*	$\rightarrow PQueue$
isEmpty:	*PQueue*	$\rightarrow Boolean$

Hilfsoperation maxprio

- Zusätzlich erforderliche Operation: Maxprio: PQueue → Int
- maxprio gehört nicht zur Signatur der Prioritätswarteschlange, weil kein Benutzer der Prioritätswarteschlange den Wert von maxprio kennen muss.
- Implementierung in Java: maxprio als private Methode vereinbaren.
- Zugehöriges Axiom:

$$maxprio(enqueue(q,(x,g))) = \begin{cases} g & \text{falls } isEmpty(q) \\ g & \text{falls } maxprio(q) < g \\ maxprio(q) & \text{sonst} \end{cases}$$

- Axiom der Schlange
  - Für eine nicht-leere Schlange ist es unerheblich, ob man zuerst ein Element anfügt und dann eines entfernt oder umgekehrt.

$$dequeue(enqueue(q,x)) = \begin{cases} q & \text{falls } isEmpty(q) \\ enqueue(dequeue(q),x) & \text{sonst} \end{cases}$$

- Axiom der Prioritätswarteschlange

$$dequeue(enqueue(q,(x,g))) = \begin{cases} q & \text{falls } isEmpty(q) \\ q & \text{falls } maxprio(q) < g \\ enqueue(dequeue(q),(x,g)) & \text{sonst} \end{cases}$$

- Axiom der Schlange
  - Das vorderste (älteste) Element einer Schlange ist unabhängig davon, ob noch weitere Elemente mittels **enqueue** angehängt wurden

$$front(enqueue(q,x)) = \begin{cases} x & \text{falls } isEmpty(q) \\ front(q) & \text{sonst} \end{cases}$$

- Axiom der Prioritätswarteschlange

$$front(enqueue(q,(x,g))) = \begin{cases} (x,g) & \text{falls } isEmpty(q) \\ (x,g) & \text{falls } maxprio(q) < g \\ front(q) & \text{sonst} \end{cases}$$

Implementierung des ADTs *P*Queue mit einer Liste

- Strategie: Einfügen sortiert nach Priorität, keine Hilfsoperation maxprio
- Abbildung:

$$create_{PQueue} = create_{List_1}$$

$$dequeue(add_{List_1}(q,(x,g))) = q$$

$$front(add_{List_1}(q,(x,g))) = (x,g)$$

$$enqueue(q,(x,g)) = \begin{cases} add_{List_1}(q,(x,g)) & \text{falls } isEmpty(q) \\ add_{List_1}(q,(x,g)) & \text{falls } q = add_{List_1}(r,(y,h)) \wedge h < g \\ add_{List_1}(enqueue(l,(x,g)),(y,h)) & \text{sonst, mit } q = add_{List_1}(l,(y,h)) \end{cases}$$

Falls (x,g) höchste Priorität hat, einfach vorne an Liste anhängen.

Sonst Listenkopf überspringen und in q gemäß Sortierung einfügen.

$$isEmpty_{PQueue} = isEmpty_{List_1}$$

Tafelübung

Einschränkung des Typparameters (Generics)

- mögliche Typen für Typparameter können eingeschränkt werden (Bounds)
  - nur Klassen, die von bestimmter Klasse erben
  - nur Klassen, die bestimmtes Interface implementieren
- Nutzen? Nutzen!
  - stellt „Mindest-Anforderung“ an Typen dar
  - konkrete Typen haben „mindestens“ die entsprechende Schnittstelle
  - erlaubt den Aufruf dieser Methoden, ohne konkreten Typ zu kennen
- in Java: Einschränkung des Typparameters mittels extends auch bei Interfaces...
- Beispiele
  - class Foo < T extends Exception > { /* ... */ }
  - class Bar < T extends Comparable <T > > { /* ... */ }

Statische generische Methoden

Folgender Code kompiliert nicht

```
public class Foo <T > {
        public Foo ( T param ) { /* ... */ }
        public static Foo <T > getFoo ( T param ) {
                if ( param == null ) { /* Fehlerbehandlung */ }
                return new Foo <T >( param );
        }
}
```

Was ist da schiefgegangen?

- bei der Instanziierung wird ein konkreter Typ für T eingesetzt
- Klassenmethoden sind nicht an ein Objekt gebunden
  - keine Instanziierung hat stattgefunden
  - für T existiert kein konkreter Typ

Folgender Code kompiliert nicht

```
public static Foo <T > getFoo ( T param ) { /* ... */ }
```

Lösung

- bei der Methodendeklaration den Typparameter mit angeben
- Compiler kann dann beim Methodenaufruf einen passenden Typ einsetzen

... im Beispiel:

```
public static <T > Foo <T > getFoo ( T param ) { /* ... */ }
// ================================================
Foo < Integer > intFoo = Foo . getFoo (4711);
```

Typlöschung

- Generics wurden erst nachträglich zu Java hinzugefügt
  - dabei wurde Augenmerk auf (Bytecode-)Kompatibilität gelegt
- Java-Laufzeitsystem kennt keine Generics im Typsystem
- deswegen notwendig: Typlöschung (Type Erasure)
  - Typparameter werden durch Object ersetzt („ausradiert“) bzw. bei Bounds durch spezifischere Typen
- explizite Typkonvertierungen werden in den Code eingefügt
- Also...
  - Bei der Übersetzung wird der Code generiert, den wir auch selbst hätten schreiben können („Schlechte Lösung (2)“). Die Typlöschung passiert aber erst nach der Typprüfung im Compiler, deswegen kann Typsicherheit zur Übersetzungszeit garantiert werden

Beispiel für Typlöschung

Code vor Typlöschung

```
class Foo <T > {
        T value ;
        public Foo ( T value ) {
                this . value = value ;
        }
        public T get () {
                return this . value
        }
}
// ===================================
Foo < Integer > f = new Foo < Integer >(13);
Integer v = f . get ();
```

Code nach Typlöschung

```
class Foo {
        Object value ;
        public Foo ( Object value ) {
                this . value = value ;
        }
        public Object get () {
                return this . value
        }
}
// ===================================
Foo f = new Foo (13);
Integer v = ( Integer ) f . get ();
```

Konsequenzen der Typlöschung

- Auf Grund der Typlöschung ist Überladen mit Typparametern nicht möglich.
- Zur Laufzeit stehen keine Typinformationen über Typparameter zur Verfügung. Folgender Code führt deshalb zu einem Fehler zur Übersetzungszeit:

```
class Foo <T > {
        /* ... */
}
// ========================
if ( f instanceof Foo < Integer >) {
        /* ... */
} else if ( f instanceof Foo < String >) {
        /* ... */
} else {
        /* ... */
}
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo <T > { /* ... */ }
// ===================================
Foo < Integer >[] foos = new Foo < Integer >[1];
```

- Arrays mit generischen Typen sind in Java problematisch...
  - Verwendung würde zu unerwartetem Verhalten führen (s.u.)
  - Grund dafür ist wieder die Typlöschung
- mögliche Lösungen sind eher Hacks als saubere Lösungen...
  - Compiler-Warnungen ignorieren/unterdrücken
  - „Hilfsklassen“ ohne Typparameter erstellen und verwenden

Unerwartetes Verhalten

Beispiel ohne Generics

```
String [] strings = new String [1];
Integer [] ints = new Integer [1];
ints [0] = 42; // Autoboxing von int nach Integer
Object [] objects = strings ;
objects [0] = ints [0]; // -> ArrayStoreException
String s = strings [0];
```

Beispiel mit Generics (Annahme: Code würde so übersetzt werden können)

```
ArrayList < String >[] strings = new ArrayList < String >[1];
ArrayList < Integer >[] ints = new ArrayList < Integer >[1];
ints [0] = new ArrayList < Integer >();
ints [0]. add (42); // Autoboxing von int nach Integer
Object [] objects = strings ;
objects [0] = ints [0];
String s = strings [0]. get (0); // -> ClassCastException
```

Erläuterung

- Der Code würde in einer ClassCastException resultieren, sobald man lesend auf ein „falsches“ Element zugreift. Eigentlich erwartet man aber schon beim Abspeichern eine ArrayStoreException.

Mögliche Lösungen

Lösung 1: Compiler-Warnungen unterdrücken

```
@SuppressWarnings (" unchecked ")
Foo < Integer >[] foos = new Foo [1];
```

Lösung 2: „Hilfsklasse“ erzeugen und verwenden

```
class Foo_Integer extends Foo < Integer > {
        /* intentionally left blank */
};
Foo_Integer [] foos = new Foo_Integer [1];
```

Generische Arrays

Folgender Code compiliert nicht

```
class Foo <T > {
        public void bar () {
                T [] array = new T [1];
        }
}
```

- Grund für den Fehler ist erneut die Typlöschung
- T ist zur Laufzeit nicht bekannt
- Java benötigt allerdings Basistyp zum Erzeugen eines Arrays

Mögliche Lösungen

Lösung 1: Compiler-Warnungen beim Zugriff/Cast unterdrücken

```
Object [] array = new Object [1];
@SuppressWarnings (" unchecked ")
T elem = ( T ) array [0];
```

Lösung 2: Compiler-Warnungen bei der Erzeugung unterdrücken (besser!)

```
@SuppressWarnings (" unchecked ")
T [] array = ( T []) new Object [1];
```

Wildcards – Motivation

Szenario

Eine Funktion soll einen Parameter eines generischen Typs haben. Der Funktion ist es aber „egal“, welcher Typparameter tatsächlich verwendet wurde; insbesondere soll der Parameter auf beliebige Typparameter passen.

Wildcards – Motivation

Folgender Code compiliert nicht

```
class Bar <T > { /* ... */ }
class Foo {
        void func1 () {
                Bar < Integer > bInt = new Bar < Integer >();
                func2 ( bInt );
                // func2 ([...]) [...] cannot be applied to func2 ([...])
        }
        void func2 ( Bar < Object > p ) { /* ... */ }
}
```

Wildcards

- das Wildcard-Symbol ? dient als Platzhalter für beliebige Typen
  - damit ist Foo<?> Basistyp „aller Foos“
  - es gehen damit aber auch Typinformationen verloren…
  - ganz grob: Lesen erlaubt, Schreiben verboten
- eine Wildcard kann mit Hilfe von Bounds eingeschränkt werden
  - ? extends Foo
    - beliebiger Typ, aber „mindestens Foo“
  - ? super Foo
    - beliebiger Typ, aber nur Oberklassen von Foo (inklusive Foo)

Lösung für unser Problem

Verwendung einer Wildcard

```
class Bar <T > { /* ... */ }
class Foo {
        void func1 () {
                Bar < Integer > bInt = new Bar < Integer >();
                func2 ( bInt );
        }
        void func2 ( Bar <? > p ) { /* ... */ }
}
```

Verkettete Listen, dynamische Arrays, Mengen, Streutabellen

Java Collection Framework

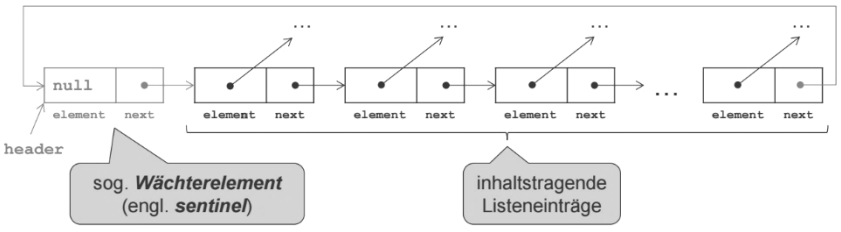
- Im Folgenden: Betrachtung von Behälter-Interfaces und -klassen, die direkt von der Java-Standard-Bibliothek (im sog. Java Collection Framework) bereitgestellt werden.
- Zweck:
  - Vertiefung des Wissens über die Implementierung von Behälterklassen
  - Anwendung standardisierter Behälter-Interfaces und -klassen mit dem Ziel der direkten Verwendbarkeit in eigenen Programmen
- Viele dieser Interfaces und Klassen verwenden die zuvor betrachtete Parametrisierungstechnik und ermöglichen es damit, konkrete Inhaltstypen erst bei der Deklaration festzulegen.

Interface java.util.List<E>

```
public interface List<E> extends Collection<E> {
        // Element elem an die Liste anhaengen;
        // liefert true, falls die Liste geaendert wurde
        boolean add(E elem);

        // Element elem an der Stelle index einfuegen;
        // erstes Listenelement: index = 0
        void add(int index, E elem);
        // alle Elemente aus der Liste entfernen
        void clear();

        // liefert true, wenn das Objekt o in der Liste enthalten ist
        boolean contains(Object o);
        // liefert das Element an der Stelle index
        E get(int index);
       // liefert true, wenn die Liste leer ist
       boolean isEmpty();
       // liefert einen Iterator fuer die Elemente der Liste in einer
       // bestimmten Reihenfolge
       Iterator<E> iterator();
       // entfernt das erste Vorkommnis des Objekts o, sofern vorhanden;
       // liefert false, wenn das Objekt o in der Liste nicht vorkommt
       boolean remove(Object o);
        // entfernt das Element an der Stelle index
        // und gibt das entfernte Element zurueck
        E remove(int index);

        // ersetzt das Element an der Stelle index mit dem spezifizierten
        // Element und gibt das ersetzte Element zurueck
        E set(int index, E elem);
        // liefert die Anzahl der Elemente der Liste
        int size();
        ...
}
```

Einfach verkettete Listen Listenimplementierung mit Wächterelement

- next d. Wächterelements zeigt auf d. vordersten Listeneintrag. Letzter Listeneintrag verweist auf Wächer → Zyklus.
- Vorteil: Operationen sind leichter implementierbar, da weniger Sonderfälle.
- Z.B. ist Einfügen am Anfang genau wie Einfügen in der Mitte oder am Ende



Private innere Klasse für Listeneintrags-Objekte

```
public class LinkedList<E> extends AbstractSequentialList<E>  implements List<E> {
        // Klasse fuer Listeneintraege:
        // jeweils mit einem Elementverweis und
        // einem Verweis auf das nachfolgende Element
        private class Entry {
                E element;
                Entry next;
                Entry(E elem, Entry next) {
                        this.element = elem;
                        this.next = next;
                }
        }
        // Waechterelement erzeugen, Zahl der Listenelemente auf 0 setzen
        private Entry header = new Entry(null, null);
        private int size = 0;
        ...
        public LinkedList() {
               // kein Typ-Parameter!
                header.next = header; // Zyklus hat nur Wächterelement
        }
```

Einschub: Innere, geschachtelte und lokale Klassen

Es ist möglich, innerhalb eines beliebigen Anweisungsblocks weitere Klassen zu definieren. Es gibt folgende Varianten:

- innere Klasse (engl. inner class): nicht-statische Klasse innerhalb einer anderen Klasse
- geschachtelte Klasse (engl. nested class): statische Klasse innerhalb einer anderen Klasse
- lokale Klasse (engl. local class): nicht-statische Klasse innerhalb einer Methode

Innere Klasse als Hilfsklasse der äußeren Klasse

- Methoden der inneren Klassen können auf alle Elemente der äußeren Klasse zugreifen, auch auf private.
- Ein Objekt der inneren Klasse ist immer abhängig von einen Objekt der äußeren Klasse, d. h. es muss ein Objekt der äußeren Klasse existieren, um eines der inneren instanziieren zu können.
    - Deshalb können innere Klassen keine statischen Elemente besitzen.
- Innere Klassen können mit den Sichtbarkeitsmodifikatoren public, private und protected spezifiziert werden.

Geschachtelte Klasse als Hilfsklasse der äußeren Klasse

- Genau wie statische Methoden können (statische) geschachtelte Klassen auf statische Elemente der sie enthaltenden Klasse zugreifen, nicht aber auf deren Instanzvariablen.
- Objekte der geschachtelten Klasse
    - sind nicht von einem Objekt der äußeren Klasse abhängig,
    - können statische Elemente besitzen.
    - Beispiel: Objekterzeugung außerhalb der Klasse Outer Outer.Nested in = new Outer.Nested();
- Geschachtelte Klassen können mit den Sichtbarkeitsmodifikatoren public, private und protected spezifiziert werden.

Einschub: Kontext-Trennung am Beispiel

```
class Outer {
        int i = 100;
        static void classMethod() {
               final int l = 200;
               class LocalInStaticContext {
                       int k = i; // Fehler (i nicht statisch)
                       int m = l; // ok (l Konstante und initialisiert)
        }
                }
        void instanceMethod() {
               int l = 200;
                class Local {
                       int k = i; // ok (i Instanzvariable)
                       int m = l; // Fehler (l nicht Konstante im aeusseren Block)
               }
        }
}
```

Hinter einem Listeneintrag einfügen O(1)

```
...
 // Hilfsmethode zum Einfuegen des Elements
 // elem nach dem Listeneintrag e
 private Entry addAfter(E elem, Entry e) {
         assert (elem != null) && (e != null);
         Entry newEntry = new Entry(elem, e.next); // 1
         e.next = newEntry; // 2
         size++;
         return newEntry;
 }
..
```

Listeneintrag/Element an der Stelle index holen O(n)

```
// Hilfsmethode, die den Eintrag an der Stelle index liefert
 private Entry entry(int index) {
         if (index < 0 || index >= size) {
               throw new IndexOutOfBoundsException();
         }
         Entry e = header;
         for (int i = 0; i <= index; i++) { e = e.next; }
         return e;
 }
 // Liefert das Element an der Stelle index
 public E get(int index) {
        return entry(index).element; //
 }
 // set(..) benutzt auch entry() □ gleicher Aufwand wie get.
 ...
```

Element am Anfang/an der Stelle index einfügen O(n)

```
// Element elem am Anfang in die Liste einfuegen
 public boolean add(E elem) {
        addAfter(elem, header);
         return true;
 }
 // Element elem an der Stelle index einfuegen
 public void add(int index, E elem) {
        addAfter(elem,
        (index == 0 ? header : entry(index - 1)));
 }
 ...
```

Test, ob ein Objekt enthalten ist (Strategie: lineare Suche) O(n)

```
...
 // ueberprueft, ob Objekt o enthalten ist
 public boolean contains(Object o) {
        // o == null wird hier nicht behandelt
        for (Entry e = header.next; e != header; e = e.next) {
                if (o.equals(e.element)) {
                        return true;
                }
        }
        return false;
 }
 ...
```

Die Suche nach dem Element erfolgt per linearer o. sequentieller Suche, weil d. Liste Element für Element durchlaufen wird.

- Bester Fall O(1)
- Schlechtester Fall O(n)
- Durchschnitt erfolgreich O(n/2)
- Durchschritt erfolglos O(n)

Einen Listeneintrag löschen – mit Schleppzeiger O(n)

```
public E remove(int index) { //remove(Object o) analog
        if (index < 0 || index >= size) {
              throw new IndexOutOfBoundsException();
        }
        Entry drag = header; //Schleppzeiger
        Entry p = header.next;
        for (int i = 1; i <= index; i++) {
              drag = p;
              p = p.next;
        }
        drag.next = p.next;
        size--;
        return p.element;
 }
```

Länge der Liste, Liste leer?, Liste leeren O(1)

```
...
 // liefert die Anzahl der Elemente in der Liste
 public int size() {
       return size;
 }

 // ueberprueft, ob die Liste leer ist
 public boolean isEmpty() {
       return size == 0;
 }
 // alle Elemente aus der Liste entfernen
 public void clear() {
       header.next = header;
       size = 0;
 }
 ...
```

Durchlaufen einer verketteten Liste – Basistechniken

Variante 1: per klassischer for-Schleife

```
// alle Medien ausgeben
Medium elem;
for (int i = 0; i < medList.size(); i++) {
       elem = medList.get(i);
       elem.print();
}
```

Variante 2: per sog. for-each-Schleife

```
// alle Bibliotheksmitglieder ausgeben
for (BibMitglied bm : bmList) {
       bm.print();
}
```

Durchlaufen einer verketteten Liste – Iterator

Fortsetzung der Implementierung von LinkedList<E>:

```
...
 // konkrete Implementierung des Iterators in LinkedList
 private class ListItr implements Iterator<E> {
 ...
 }

 // liefert einen Iterator fuer die Liste
 public Iterator<E> iterator() {
        return new ListItr();
 }
 ...
}
```

Iterator

- Möglichkeit zur Navigation durch eine beliebige Datenstruktur.
- Iterator ist eine Art fortschaltbarer Zeiger, der immer das nächste Element der Datenstruktur liefert.
- Zustand der Traversierung wird damit unabhängig von der Datenstruktur.
- Mit Iteratoren ist Datenstruktur gleichzeitig mehrfach durchlaufbar

Interface java.util.Iterator<E>

```
public interface Iterator<E> {
        // liefert true, wenn die Aufzaehlung weitere Elemente enthaelt
        // oder: liefert true, wenn next() ein weiteres Element liefert und keine
        // Ausnahme vom Typ NoSuchElementException ausloest
        boolean hasNext();
        // liefert das naechste Element der Aufzaehlung (vom Typ E) oder loest
        // eine Ausnahme vom Typ NoSuchElementException aus, wenn kein naechstes
        // Element existiert
        E next();
        // entfernt das Objekt, das der letzte Aufruf von next() geliefert hat,
        // aus der zugeordneten Sammlung
        // pro next()-Aufruf ist nur ein remove()-Aufruf zugelassen
        // Achtung: das Verhalten des Iterators ist unspezifiziert, wenn die
        // zugrunde Sammlung zwischenzeitlich auf andere Weise als durch Aufruf
        // dieser Methode modifiziert wurde
        void remove();
}
```

Durchlaufen einer Liste mit einem Listen-Iterator

```
// alle Medien ausgeben
Iterator<Medium> iter = medList.iterator();
Medium elem;
while (iter.hasNext()) {
       elem = iter.next();
       elem.print(); // mache etwas mit dem Listenelement
}
```

Implementierung des Listen-Iterators mit innerer Klasse

```
...
 private class ListItr implements Iterator<E> {
        int currentIdx = 0;
        Entry currentElement = header; // initial
        public boolean hasNext() {
              return (currentIdx < size);
        }

        public E next() {
               if (currentIdx < size) {
                       currentElement = currentElement.next;
                       currentIdx++;
                      return currentElement.element;
               } else {
                     throw new NoSuchElementException();
               }
        }

        public void remove() { ... }
 }
 ...
```

Dynamische Arrays

Einfach verkettete Listen vs. Standard-Arrays

	Einfach verkettete Listen	Standard-Arrays
Speicherbedarf	▪ ändert sich dynamisch mit der Anzahl der enthaltenen Objekte ▪ immer nur so viel, wie nötig ▪ Speicherbedarf für Zeiger	▪ wird einmal festgelegt und ist dann nicht veränderbar ▪ ggfs. Speicherverschwendung bei nicht voll belegtem Array
Element suchen	▪ lineare Suche	▪ binäre Suche (wenn sortiert)
i-tes Element zugreifen	▪ Liste von vorne bis zur betreffenden Stelle durchlaufen	▪ direkter Zugriff möglich

Dynamisches Array erzeugen

```
public class ArrayList<E> extends java.util.AbstractList<E>
                 implements java.util.List<E>,  java.util.RandomAccess {
        protected int size; // Anzahl der enthaltenen Elemente
        protected E[] list;
        // Konstruktor fuer dynamisches Array
        // mit Kapazitaet capacity
        public ArrayList(int capacity) {
               list = (E[])new Object[capacity];
               size = 0;
        }
        // weiterer Konstruktor fuer dynamisches Array,
        // das alle Elemente der Collection c aufnimmt
        public ArrayList(java.util.Collection<E> c) {
                list = (E[])new Object[c.size()];
               size = 0;
               for (E e : c) {
                      list[size++] = e;
               }
        }
...
```

Element ersetzen und lesen, Anzahl der enthaltenen Elemente

```
// Ersetzen des Wertes an Position index durch element;
// gibt ersetztes Element zurueck
public E set(int index, E element) {
        if (index >= size) throw new IndexOutOfBoundsException();
        E oldValue = list[index];
        list[index] = element;
        return oldValue;
}

// Lesen des Wertes an der Position index
public E get(int index) {
       if (index >= size) throw new IndexOutOfBoundsException();
       return list[index];
}

// Ist das dynamische Array leer?
public boolean isEmpty() {
       return (size == 0);
}

// gibt die Anzahl der im dynamischen Array
// enthaltenen Elemente zurueck
public int size() {
       return size;
}
```

Elemente tauschen, Element in der Mitte einfügen

```
...
// vertauscht die Elemente an den Positionen i und j
public void swap(int i, int j) {
        if ((i >= size) || (j >= size)) {
               throw new IndexOutOfBoundsException();
       }
        E h = list[i];
        list[i] = list[j];
        list[j] = h;
        }
// Wert element an Position index einfuegen
public void add(int index, E element) {
        // sicherstellen, dass das interne Array gross genug ist;
        // ggfs. groesseres internes Array erzeugen
        ensureCapacity(size + 1);
        // Elemente ab index um eins nach oben schieben
        for (int i = (size - 1); i >= index; i--) {
                list[i + 1] = list[i];
        }
        list[index] = element; // neues Element einfuegen
        size++;
}
//add(element) wird auf add(0, element) zurueck gefuehrt
...
```

Element in der Mitte einfügen

Einfügen in der Mitte und Löschen aus der Mitte eines Arrays haben einen Aufwand $O(n)$, weil Elemente „rechts“ von der Einfüge- bzw. Löschstelle verschoben werden müssen.

Verschieben (= Kopieren von aufeinander folgenden Elementen) ist auf heutigen Rechnern schneller als unsortiertes Kopieren von Einzelwerten (Cache-Speicher unterstützen räumliche Lokalität, DMA-Bausteine).

Deshalb anstatt

```
for (int i = (size - 1); i >= index; i--) {
        list[i + 1] = list[i];
}
```

besser so:

```
System.arraycopy(list, index, list,  index + 1, size - index);
```

Vergleiche: Bei einer Liste haben diese Operationen den Aufwand $O(1)$, wenn die Einfüge- bzw. Löschstelle schon gefunden ist.

Kapazität des dynamischen Arrays sicherstellen

```
...
 // sicherstellen, dass internes Array gross genug ist
 private void ensureCapacity(int minCapacity) {
         int oldCapacity = list.length;
         // ueberschreitet benoetigte Kapazitaet die vorhandene, ...
         if (minCapacity > oldCapacity) {
                 // ... dann neuen Verweis auf das alte interne Array
                 // erzeugen
                 E oldData[] = list;
                 // ... neues 50% groesseres Array erzeugen ...
                 int newCapacity = (oldCapacity * 3)/2 + 1;
                 // (falls Vergroesserung nicht ausreichend,
                 // benoetigte Groesse verwenden)
                 if (newCapacity < minCapacity) {
                        newCapacity = minCapacity;
                 }
                 list = (E[]) new Object[newCapacity];
                 // ... und umkopieren
                 System.arraycopy(oldData, 0, list, 0, oldCapacity);
         }
 }
 ...
}
```

Vergleich einfach verkettete Liste und dynamisches Array

	Einfach verkettete Liste `LinkedList`	Dynamisches Array `ArrayList`
`boolean add(E elem)`	*O(1)*	*O(n)*
`void add(int index, E elem)`	*O(n)*	*O(n)*
`void clear()`	*O(1)*	*O(1)**
`boolean contains(Object o)`	*O(n)*	*O(n)**
`E get(int index)`	*O(n)*	*O(1)*
`boolean isEmpty()`	*O(1)*	*O(1)*
`boolean remove(Object o)`	*O(n)*	*O(n)**
`E remove(int index)`	*O(n)*	*O(n)**
`E set(int index, E elem)`	*O(n)*	*O(1)*
`int size()`	*O(1)*	*O(1)*

Kann bei sortierter Speicherung in `ArrayList` ohne Steigerung der Komplexität bei den anderen Operationen auf *O(log n)* verbessert werden. Siehe unten.

Mengen Motivation

- Darstellung von Mengen ist eine der grundlegendsten Aufgaben überhaupt.
- Bislang können wir zur Darstellung von Mengen Listen und dynamische Arrays einsetzen.
- Im Folgenden: verschiedene Implementierungsmöglichkeiten von Mengen mit Hilfe von Listen und dynamischen Arrays. (Implementierungen mit Bäumen folgen in der LE „Bäume“).
- Ziele:
  - Mengen so darstellen, dass die klassischen Operationen Hinzufügen und Entnahme von Elementen, Vereinigung, Durchschnitt und Differenz effizient ausgeführt werden können.
  - Mengen enthalten keine Duplikate von Elementen.
  - Elemente haben keine feste Reihenfolge.

Abstrakter Datentyp Set

adt *Set*
sorts *Set, E, Boolean*
ops

empty:		→ *Set*
add:	*Set* × *E*	→ *Set*
isIn:	*Set* × *E*	→ *Boolean*
del:	*Set* × *E*	→ *Set*
isEmpty:	*Set*	→ *Boolean*
single:	*E*	→ *Set*
union:	*Set* × *Set*	→ *Set*
intersect:	*Set* × *Set*	→ *Set*
diff:	*Set* × *Set*	→ *Set*
equals:	*Set* × *Set*	→ *Boolean*
containsAll:	*Set* × *Set*	→ *Boolean*

axs

...

end *Set*

Interface java.util.Set

```
public interface Set<E> extends Collection<E> {
         // liefert true und fuegt das Element elem zur Menge hinzu, wenn es
         // nicht bereits vorhanden war
         boolean add(E elem);
         // fuegt alle Elemente der Collection c zur Menge hinzu, wenn diese
         // nicht bereits enthalten sind (entspricht Vereinigung der Mengen)
         boolean addAll(Collection<? extends E> c);
         // entfernt alle Elemente aus Menge
         void clear();
         // liefert true, wenn die Menge das spezifizierte Objekt o enthaelt
         boolean contains(Object o);
         // liefert true, wenn die Menge alle Elemente der Collection c enthaelt
         // (entspricht Teilmengenbeziehung)
         boolean containsAll(Collection<?> c);

         // liefert true, wenn das spezifizierte Objekt o mit der Menge
         // uebereinstimmt (entspricht Gleichheit zweier Mengen)
         boolean equals(Object o);
         // liefert true, wenn die Menge leer ist
         boolean isEmpty();
         // liefert einen Iterator fuer die Elemente der Menge
         Iterator<E> iterator();
         // liefert true und entfernt das spezifizierte Objekt o aus der Menge,
         // wenn es darin vorhanden war
         boolean remove(Object o);
         // entfernt alle Elemente der Collection c aus der Menge, sofern diese
         // darin enthalten sind (entspricht Differenz der Mengen);
         // liefert true, wenn die Menge durch
         // die Operation geaendert wurde
         boolean removeAll(Collection<?> c);
         // behaelt diejenigen Elemente der Menge, die auch in der Collection c
         // enthalten sind (entspricht Schnittmenge);
         // liefert true, wenn die Menge durch
         // die Operation geaendert wurde
         boolean retainAll(Collection<?> c);
         // liefert die Anzahl der Elemente in der Menge (ihre Kardinalitaet)
         int size();
}
```

Implementierungen des Abstrakten Datentyps *Set*

- Wir betrachten im Folgenden drei Implementierungsvarianten:
  - Implementierung mit einfach verketteter Liste ohne Sortierung der Elemente
  - Implementierung mit einfach verketteter Liste mit Sortierung der Elemente
  - Implementierung mit dynamischem Array mit Sortierung der Elemente
  - Implementierung als Bitvektor ohne Sortierung der Elemente
- Zu beachten bei den Implementierungsvarianten 1-3:
  - Einfach verkettete Listen bzw. dynamische Arrays erlauben Duplikate, Mengen aber nicht. Die Mengenoperationen sind also so zu gestalten, dass Duplikate vermieden werden.
  - Eingaben der Mengenoperationen werden transformiert in Eingaben für die jeweils zugrunde liegende Datenstruktur und deren Ausgaben in Ausgaben entsprechender Mengenoperationen.

Menge ohne Sortierung (als einfach verkettete Liste)

- Grundidee:
  - Mengenelemente werden in nicht-sortierter, einfach verketteter Liste gehalten.
  - Neue Elemente werden vorne an der zugrundeliegenden Liste angefügt, sofern sie nicht bereits enthalten sind (Test erfordert Durchlaufen der Liste per linearer Suche).
- Aufwände:
  - Alle Operationen, die auf ein bestimmtes Element zugreifen müssen (Suchen, Löschen) erfordern Durchlaufen der Liste bis zum jeweiligen Element (Aufwand $O(n)$).
  - Alle Operationen, die auf mehrere Elemente zugreifen (mehrere Elemente suchen, hinzufügen, löschen), erfordern dementsprechend mehrfaches Durchlaufen der Liste (Aufwand $O(n^2)$)

Menge anlegen, Elementanzahl, Mengen-Iterator

```
public class SetAsList<E> extends AbstractSet<E> implements Set<E> {
        // zugrunde liegende Liste
        List<E> list;

        // Erzeugen einer leeren Liste
        public SetAsList() {
               list = new LinkedList<E>();
        }
        // liefert die Kardinalitaet der Menge O(1)
        public int size() {
               return list.size();
        }
        // prueft, ob die Menge leer ist O(1)
        public boolean isEmpty() {
               return list.isEmpty();
        }
        // liefert einen Iterator fuer die Menge O(1)
        public Iterator<E> iterator() {
               return list.iterator();
        }
 ...
```

Enthaltensein eines Elements, Teilmenge, Element hinzufügen

```
// prueft, ob die Menge das Objekt o enthaelt O(n)
 public boolean contains(Object o) {
       return list.contains(o);
 }
 // prueft, ob die Menge alle Elemente aus c O(n^2)

 // enthaelt (Teilmengenbeziehung)
 public boolean containsAll(Collection<?> c) {
        Iterator<?> i = c.iterator();
        while (i.hasNext()) {
               if (!contains(i.next())) {
                      return false;
                }
       }
       return true;
 }
 // fuegt das angegebene Element zur Menge hinzu, O(n)
 // sofern nicht schon enthalten
 public boolean add(E elem) {
       if (contains(elem))
               return false;
       return list.add(elem);
 }
 ...
```

Vereinigungsmenge, Element entfernen, Differenzmenge

```
// fuegt alle Elemente von c zur Menge hinzu O(n²)
 // (Vereinigungsmenge)
 public boolean addAll(Collection<? extends E> c) {
        Iterator<? extends E> i = c.iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               setChanged |= add(i.next());
        }
        return setChanged;
 }
 // entfernt Objekt o aus der Menge, sofern enthalten O(n)
 public boolean remove(Object o) {
       return list.remove(o);
 }
 // entfernt alle Elemente von c aus der Menge (Differenz) O(n²)
 public boolean removeAll(Collection<?> c) {
        Iterator<?> i = c.iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               setChanged |= remove(i.next());
        }
        return setChanged;
 }
 ...
```

Schnittmenge, Menge leeren

```
// behaelt die Elemente der Menge,
 // die auch in c enthalten sind
 // (entspricht Schnittmenge) O(n²)
 public boolean retainAll(Collection<?> c) {
        Iterator<E> i = iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               E a = i.next();
               if (!c.contains(a)) {
                      i.remove();
                      setChanged = true;
               }
        }
        return setChanged;
 }
 // leert die Menge
 public void clear(){
       list.clear();
 }
 ...
```

Gleichheit zweier Mengen

```
// prueft die Gleichheit zweier Mengen O(n2)
 public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Set)) return false;
        Collection c = (Collection) o;
        if (c.size() != size())
               return false;
        return containsAll(c);
 }
}
```

Fazit: Aufwände von SetAsList

- Beobachtung: Verkettete Liste hatte für Hinzufügen von Objekt Aufwand $O(1)$ bzw. $O(nn)$ bei mehreren Objekten.
- Da keine Duplikate erlaubt sind, muss this für jedes einzufügende Objekt durchlaufen werden.

Test auf leere Menge	`boolean isEmpty()`	$O(1)$
Zahl der Elemente der Menge	`int size()`	$O(1)$
Test auf Enthaltensein eines Objekts	`boolean contains(Object o)`	$O(n)$
Test auf Teilmengenbeziehung	`boolean containsAll(Collection<?> c)`	$O(n^2)$
ein Objekt hinzufügen	*`boolean add(E elem)`*	$O(n)$
Vereinigungsmenge bilden	*`boolean addAll(Collection<? extends E> c)`*	$O(n^2)$
ein Element entfernen	`boolean remove(Object o)`	$O(n)$
Differenzmenge bilden	`boolean removeAll(Collection<?> c)`	$O(n^2)$
Schnittmenge bilden	`boolean retainAll(SetAsList<E> c)`	$O(n^2)$
Test der Mengen auf Gleichheit	`boolean equals(Object o)`	$O(n^2)$

Menge mit Sortierung (als einfach verkettete Liste)

Grundidee: wenn die Elemente in der Menge aufsteigend sortiert sind, dann …

- … muss man nicht die ganze verkettete Liste durchlaufen, um nach Duplikaten zu suchen und:
  - contains endet im Durchschnitt schneller,
  - add endet im Durchschnitt schneller (weil contains benutzt wird) und
  - remove endet im Durchschnitt schneller.
- … sind die Mengenoperationen schneller (Aufwand in $O(n)$ statt in $O(n^2)$), weil ein Reißverschlussverfahren verwendet werden kann.

Aufwände:

- Objekt einfügen: richtige Position suchen, dort einfügen. Fällt mit Enthaltenseinstest zusammen. Aufwand bleibt: $O(n)$
- Vereinigungs-, Schnitt- und Differenzmenge: „paralleler" Durchlauf durch beide beteiligte Listen mit Reißverschlussverfahren. Aufwand reduziert sich zu: $O(n)$

Vergleich von Objekten

- Um die schnelleren Mengenoperationen anwenden zu können, ist es Grundvoraussetzung, dass die beteiligten Mengen bereits sortiert sind.
- Wir haben bereits das Sortieren durch Auswählen (SelectionSort) kennen gelernt.
- In einer späteren LE lernen wir weitere Sortierverfahren und deren Implementierung mit einfach verketteten Listen bzw. dynamischen Arrays kennen.
- Wichtige Voraussetzung zur Herbeiführung bzw. Aufrechterhaltung einer Sortierung:
  - Die Objekte müssen vergleichbar sein.
  - Es muss eine Ordnung für diese Objekte definiert sein.

Vergleich von Objekten in Java: java.lang.Comparable

- Zu vergleichende Objekte sollen in Java das Interface java.lang.Comparable<T> implementieren.
- Dieses definiert nur die Methode:public int compareTo(T o);
- Alle Implementierungen der Schnittstelle Comparable sollten
  - einen negativen Wert zurückgeben, wenn gemäß der gegebenen Ordnung das gegebene Objekt „kleiner" als das übergebene Objekt ist, also: this$t$ < o
  - 0 zurückgeben, wenn die beiden Objekte „gleich groß" sind, also: this = o
  - einen positiven Wert zurückgeben, wenn das gegebene Objekt „größer als das übergebene Objekt ist: $t$his > o
- Wir gehen im Folgenden davon aus, dass Elemente von sortierten Listen dieses Interface implementieren: LinkedList<E extends Comparable<E>>

Implementierung der Vereinigung zweier Mengen mit sortierten verketteten Listen

```
public class LinkedList<E extends Comparable<E>> {
 ...
        public boolean addAll(LinkedList<E> c) {
                boolean setChanged = false;

                Entry currentElement = header.next; // initial
                Entry drag = header;
                for (E cElem : c) {
                        while (currentElement != header &&
                           currentElement.element.compareTo(cElem) <= 0) {
                               drag = currentElement;
                               currentElement = currentElement.next;
                       }
                        drag = addAfter(cElem, drag);
                       setChanged = true;
                }
                return setChanged;
        }
 ...
}
```

Menge mit Sortierung (als dynamisches Array)

- Bei der Implementierung einer Menge mit Hilfe einer einfach verketteten Liste profitierten allein die Mengenoperationen von der Sortierung.
- Bei der Suche nach einem Element mit contains(…)und beim Einfügen mit add(…) musste die ganze Liste per linearer Suche durchlaufen werden.
- Vom Suchaufwand her ist eine Implementierung mithilfe eines Arrays eine günstigere Lösung, da direkt auf die einzelnen Elemente zugegriffen werden kann.
- Die Speicherung als Array spart Platz, weil keine Verzeigerung nötig ist. Sie kostet Laufzeit, weil beim Änderung der Größe Kopierarbeit anfällt

Menge mit Sortierung (als dynamisches Array)

- Zuvor: Implementierung einer Menge (Interface Set) als SetAsList<E> mithilfe einer verketteten Liste LinkedList<E>
- Nun: Implementierung einer Menge (Interface Set) als SetAsArrayList<E> mithilfe eines dynamischen Arrays ArrayList<E>
- Zusätzlich:
  - Sortierung der Elemente (SortedSetAsArrayList)
  - Mengenelemente implementieren java.lang.Comparable<T>

```
public class SortedSetAsArrayList<E extends Comparable<E>>
                extends java.util.AbstractSet<E>  implements java.util.Set<E> {
        // zugrunde liegendes dynamisches Array
        protected ArrayList<E> list;
        // leere Menge erzeugen
        public SortedSetAsArrayList() {
              list = new ArrayList<E>(10);
 }
```

Elementanzahl, Mengen-Iterator, Enthaltensein eines Elements

- Zur Erinnerung: Für den Test auf Enthaltensein (contains) eines bestimmten Elements in einem Array haben wir bereits zwei Strategien kennengelernt und untersucht:
  - Lineare Suche $O(n)$ (in unsortiertem Array)
  - Binäre Suche O (log n) (in sortiertem Array)
- Wenn ArrayList Elemente sortiert speichert, dann hat contains den Aufwand O(log n).

```
// liefert die Zahl der Elemente der Menge
 public int size() {
       return list.size();
 }
 // prueft, ob die Menge leer ist
 public boolean isEmpty() {
        return list.isEmpty();
 }
 // liefert einen Iterator fuer die Menge
 public java.util.Iterator<E> iterator() {
        return list.iterator();
 }
 // prueft, ob Menge das Element o enthaelt
 public boolean contains(Object o) {
       return (index(o) != -1);
 }
```

Aufwand binäre Suche vs. lineare Suche

	binäre Suche	lineare Suche
bester Fall	$O(1)$	$O(1)$
schlechtester Fall	$O(\log n)$	$O(n)$
Durchschnitt (erfolgreich)	$O(\log n)$	$O(n)$
Durchschnitt (erfolglos)	$O(\log n)$	$O(n)$

Binäre Suche: rekursive Implementierung

```
private boolean searchRec(int x, int[] a, int u, int o) {
        // {P: 0 <= u && o < a.length}
        int m = (u + o) / 2;
        if (u > o) return false;
        else if (x == a[m]) return true;
        else if (x < a[m]) return searchRec(x, a, u, m - 1);
        else return searchRec(x, a, m + 1, o);
        // {Q: x in a[u]...a[o]?}
}
```

rekursive Methode wird nach Außen verdeckt:

```
public boolean search(int x, int[] a) {
       return searchRec(x, a, 0, a.length - 1);
}
```

Binäre Suche: Entrekursivierung, dann Einbau in Außenmethode

```
public boolean search(int x, int[] a) {
        int u, o, m;
         u = 0;
         o = a.length - 1;
         m = (u + o) / 2;
         while ((u <= o) && (x != a[m])) {
                 // {0 <= u && o < a.length}
                 if (x < a[m]) {
                        o = m - 1;
                 } else {
                         u = m + 1;
                 }
                 m = (u + o) / 2;
         }
         return (u <= o);
         // {x in a[0]...a[length-1]?}
}
```

Binäre Suche: Methode index(…)

Damit ist index(…) für SortedSetAsArrayList<E> eine leichte Modifikation von boolean search(int x, int[] a):

```
private int index(Object p) {
         // sicherstellen, dass p vom Typ Comparable<E> ist, um
         // dann compareTo ohne Typwandlungen verwenden zu koennen
         Comparable<E> x;
         try {
                 x = (Comparable<E>) p;
         } catch (ClassCastException e) {
                return -1;
         }
         // eigentliche Suche
         int u = 0, o = size() - 1, m = (u + o) / 2;
         while ((u <= o) && (x.compareTo(list.get(m))) != 0) {
                 if (x.compareTo(list.get(m)) < 0) {
                        o = m - 1;
                 } else {
                         u = m + 1;
                 }
                 m = (u + o) / 2;
         }
         return ((u <= o) ? m : -1);
}
```

Element hinzufügen, Element entfernen

```
...
 // fuegt das angegebene Element o zur Menge hinzu O(n)
 public boolean add(E o) {
        if (index(o) == -1) {
                list.add(indexneu(o), o);
                return true;
        } else {
                 return false;
         }
 }
 // entfernt das angegebene Element aus der Menge O(n)
 public boolean remove(Object o) {
         int index = index(o);
         if (index != -1) {
                list.remove(index);
                return true;
         } else {
                return false;
         }
 }
 ...
}
```

Menge ohne Sortierung (als Bitvektor)

- Falls die darzustellende Menge S Teilmenge eines genügend kleinen, endlichen Wertebereichs $U = \{a_1 ..... a_N\}$ ist, so eignet sich eine Bitvektor-Darstellung (U: Universum).
- In der Java-API ist diese Idee realisiert durch die Klasse BitSet.

Menge ohne Sortierung (als Bitvektor)

```
class Set {
        boolean set[] = new boolean[10000000];
        ...
        void add(int key) {
              set[key] = true;
        }
        boolean contains(int key) {
              return set[key];
        }
}
```

- Einfügen: $O(1)$ Entfernen: $O(1)$ Vereinigung, Schnitt, Differenz: $O(N)$
- Achtung: Aufwand ist proportional zur Größe N des Universums, nicht zur Größe n der dargestellten Mengen

Vergleich der Mengen-Implementierungen

	Verkette Liste (ohne Sortierung)	Verkettete Liste (mit Sortierung)	Dynamisches Array (mit Sortierung)	Bitvektor
`boolean isEmpty()`	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	O(N)
`int size()`	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	O(N)
`boolean contains(Object o)`	$O(n)$	$O(n)$	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(1)}$
`boolean containsAll(Collection<?> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean add(E elem)`	$O(n)$	$O(n)$	$O(n)$	$\boldsymbol{O(1)}$
`boolean addAll(Collection<? extends E> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean remove(Object o)`	$O(n)$	$O(n)$	$O(n)$	$\boldsymbol{O(1)}$
`boolean removeAll(Collection<?> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean retainAll(SetAsList<E> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean equals(Object o)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)

Wörterbuch (engl. dictionary)

- Viele Anwendungen von Mengendarstellungen benötigen die Operationen Durchschnitt, Vereinigung und Differenz nicht.
- Der am häufigsten benötigte Satz von Operationen enthält Einfüge-, Lösch- und Nachschlage-Operationen.
- Wörterbuch (engl. dictionary) ist ein Datentyp für im Wesentlichen genau diese Operationen.
- Beispiel: Lieferservice
  - speichert Daten seiner Kunden: Name, Vorname, Adresse, TelNr
  - Bestellung: Kunde gibt TelNr an und wird darüber eindeutig identifiziert, ermöglicht Zugriff auf gespeicherte Adressdaten.
- Abstraktion: in einem Wörterbuch ist über eindeutigen Schlüssel Zugriff auf zugeordneten Wert möglich.
- Wörterbuch verwaltet solche Schlüssel-Wert-Paare.
- Frage: Wie lassen sich solche Strukturen effizient implementieren, wenn man besonders an schnellem Nachschlagen interessiert ist?

Grundidee von Streuverfahren/Hash-Verfahren

- Gesucht: effiziente Speichermöglichkeit für eine Teilmenge von Elementen eines (beliebig) großen Wertebereichs.
- Beispiel: Telefonnummern der aktuellen Kunden eines Lieferservices vs. alle vergebenen Telefonnummern der Region.
- Strategie: aus dem Wert eines zu speichernden Mengenelements wird seine Adresse im Speicher berechnet.
- Speicher zur Aufnahme der Mengenelemente:
  - Tabelle (Array) der Größe m (sog. Streutabelle, auch: Hash-Tabelle)
  - m ist i. d. R. sehr viel kleiner als der Wertebereich aus dem die Mengenelemente stammen.
- Annahme: zu speichernde Mengenelemente haben eindeutigen (numerischen) Schlüssel(wert) (engl. key), z. B.
  - ISBN-Nr. als Schlüssel von Buch-Objekten
  - MatrikelNr. als Schlüssel von Student-Objekten

Streutabellen (auch: Hash-Tabellen)

Prinzip: Statt bei key wird ein Eintrag bei h(key) in der Tabelle vermerkt

```
class MyType {
 String key;
 ...
}

class HashTable {
        MyType table[] = new MyType[200];
        void add(MyType elem) {
              table[h(elem.key)] = elem;
        }
        boolean contains(MyType elem) {
              return table[h(elem.key)] == elem;
        }
        static int h(String k) {
        ...
        }
        ...
}
```

Streufunktion (auch: Hash-Funktion, Schlüsseltransformation)

Erforderlich ist sog. Streufunktion (auch: Hash-Funktion, Schlüsseltransformation), das ist eine Abbildung h: $G \rightarrow S$

- wobei G Grundmenge für die Schlüssel der Mengenelemente ist
- und S = {0, … , mm− 1} ∈ ℕ Indizes eines Arrays (der Streutabelle) sind.

Eigenschaften der Streufunktion h

- h wird im Allgemeinen nicht injektiv gewählt (injektiv: jedes Element der Zielmenge wird höchstens einmal als Funktionswert angenommen).
  - Wenn h injektiv wäre,
  - dann müsste für die Größe m der Streutabelle gelten $m \geq G$.
  - Die tatsächlich in die Tabelle eingetragene Schlüsselmenge G' ist im Allgemeinen eine Teilmenge von G, mit $G' \ll G$
  - Daher bedingt $m \geq GG$Speicherplatzverschwendung.
  - Daher erfolgt keine Forderung nach Injektivität von h.
  - Folge: sog. Kollisionen, d. h. es gibt $x, y \in G$ mit $x \neq y$, für die $h(x) = h(y)$.
- h sollte möglichst surjektiv sein, also alle Tabellenindizes erfassen (surjektiv: jedes Element der Zielmenge wird mindestens einmal als Funktionswert angenommen).
  - Die meisten Streufunktionen verursachen nicht besetzte Arrayelemente, sog. Lücken (in der Tabelle).
  - Der Belegungsfaktor (auch: Lastfaktor) $BF := \frac{G'}{S} g$ibt an, wie gut die Tabelle besetzt ist.
  - h sollte so gewählt werden, dass für gegebenes G und moderaten bis hohen Lastfaktor (→ sparsamer Umgang mit begrenztem Speicherplatz) die Zahl der Kollisionen gering ist. (Achtung: das sind widersprüchliche Ziele.)
- h sollte die zu speichernden Schlüssel möglichst gleichmäßig auf den Bereich der Tabellenindizes verteilen.
- h sollte effizient zu berechnen sein (also in $O(1)$ oder $O(\log n)$).

Weitere Eigenschaften von Streutabellen

- Streutabellen erfordern nicht, dass zwei Werte miteinander vergleichbar sind (keine totale Ordnung notwendig). Dies war Voraussetzung der (meisten) bisherigen Verfahren.
- Streutabellen ermöglichen weder eine sortierte Ausgabe der Elemente noch einen schnellen Zugriff auf ein Maximum oder Minimum.
- Streutabellen haben im Allgemeinen eine feste Größe und passen sich nicht automatisch dynamisch wachsenden Datenmengen an

Beispiel: Monatsnamen verteilen

- Ziel: Monatsnamen über 17 Behälter verteilen.
- Ansatz Streufunktion:
  - Betrachte Zeichenketten c_0, … , c_{l-1}
  - Bestimme Zahlenwert der Binärdarstellung jedes Zeichens: $N(c_i)$
  - Dann wähle z.B.: $h(c_0 \dots \dots c_{l-1}) = \Sigma_{l-0}^{l-1} N(c_i)) \; mod \; m$
- Strategie anwendbar für allgemeine Zeichenketten

Beispiel: Monatsnamen verteilen

- Vereinfachung des Beispiels:
  - Nur erste drei Zeichen betrachten
  - Zusätzliche Annahme:
    - $N(A) = N(a) = 1$
    - $N(B) = N(b) = 2$
  - Umlaute werden als zwei Zeichen
    - $h(\text{"Apr"}) = (1 + 16 + 18) \bmod 17 = 1$
    - $h(\text{"Dez"}) = (4 + 5 + 26) \bmod 17 = 1$

Kollisionen, offene und geschlossene Streuverfahren

- Offenes Hashing:
  - Behälter kann beliebig viele kollidierende Schlüssel aufnehmen.
  - Behälter wird z. B. durch verkettete Liste realisiert.
- Geschlossenes Hashing:
  - Behälter kann nur kleine konstante Anzahl b von kollidierenden Schlüsseln aufnehmen.
  - Falls mehr als b Schlüssel auf Behälter fallen, entsteht sog. Überlauf (engl. overflow).
  - Gewöhnlich betrachteter Spezialfall: $b = 1$
    - Kollision im engeren Sinn.
    - Behälter wird im Allgemeinen als Zelle bezeichnet.
- Wie wahrscheinlich sind Kollisionen?
- Umgang mit Kollisionen?

Kollisionswahrscheinlichkeit

- Annahme: „ideale" Streufunktion, die n Schlüsselwerte völlig gleichmäßig auf m Behälter verteilt, $n < m$.
- P_X: Wahrscheinlichkeit, dass Ereignis X eintritt.
- $P_{\text{Kollison}} = 1 - P_{\text{keine Kollision}}$
- $P_{\text{keine Kollision}} = P(1) * P(2) * \ldots * P(n)$
- $P(i)$: Wahrscheinlichkeit, dass der i-te Schlüssel auf freien Behälter abgebildet wird, wenn alle vorherigen Schlüssel ebenfalls auf freie Behälter abgebildet wurden.
- $P_{\text{keine Kollision}}$: Wahrscheinlichkeit, dass alle Schlüssel auf freie Behälter abgebildet werden.
- P(1) = 1, P(2) = (m-1)/(m), P(i) = (m-i+1)(m)
- Damit $P_{\text{Kollision}}$ = 1- (m(m-1)(m-2)….. (m-n+1))/(m^n)
- Zahlenbeispiel: mm = 365, sog. Geburtstagsparadoxon

n	$P_{Kollision}$
22	0,475
23	0,507
50	0,970

Gute Streufunktionen für die Praxis

- Einfach aber effektiv: Wähle Primzahl m und setze die Größe der Streutabelle auf m fest. Dann ist gute Streufunktion:
  - $h(k) = k \bmod m$
- Kann die Größe der Streutabelle keine Primzahl sein (oft ist $m = 2^i$, 2er-Potenz), dann ist gute Streufunktion:
  - $h(k) = (k \bmod p) \bmod m$, wobei $\boldsymbol{p}$ Primzahl und $\boldsymbol{m < p \ll G}$.
- Sollten die zu speichernden Werte (doch) einen Zusammenhang zur Primzahl p haben, dann wähle zufällig $0 \neq a < p$ und $b < p$:
  - $h(k) = ((ak + b) \bmod p) \bmod m$, wobei $\boldsymbol{p}$ Primzahl und $\boldsymbol{m < p \ll GG}$.

Umgang mit Kollisionen beim offenen Hashing

- Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt.
- Array von Zeigern verwaltet die Behälter
- Elemente, die auf den gleichen Index abgebildet werden, werden als Elemente einer verketteten Liste (Überlaufbereich) an den entspr. Tabelleneintrag angehängt.
- Implementierungstechnik mit getrennten Listen für jeden Behälter wird auch als separate chaining bezeichnet.

Eigenschaften des offenen Hashings

- Vorteile:
  - Streutabelle funktioniert auch noch (obwohl langsam), wenn ihre Größe zu klein gewählt war.
  - Löschen ist möglich.
- Nachteile:
  - Zusätzlicher Speicherplatz wird für die Zeiger benötigt.
  - Es können lange Listen entstehen.
- Aufwandsabschätzung:
  - Schlimmster Fall: h liefert immer d. selben Wert, alle Elemente befinden sich in einer Liste Aufwand $O(n)$.
  - Bester Fall: h streut perfekt, keine Kollisionen, $O(1)$.
  - Durchschnittlicher Fall: Faustregel etwa „1 plus Belegungsfaktor", O(1+(n/m))

Geschlossenes Hashing

- Zahl der Einträge n ist begrenzt durch Kapazität der Tabelle $m \cdot b$
- Im Folgenden: Spezialfall $b = 1$; für $b > 1$ analoge Techniken.
- Jede Zelle (jeder Behälter) der Streutabelle kann also genau ein Element aufnehmen; Implementierung durch Array.
  - `E[] hashtable = (E[])new Object[m];`
- Annahmen:
  - Elemente haben Schlüssel(-wert) key.
  - Spezieller Wert empty der Schlüssel-Domäne zeigt an, dass Zelle unbesetzt ist.
  - Spezieller Wert deleted der Schlüssel-Domäne zeigt an, dass Zelle in der Vergangenheit besetzt war, dass aber das damals enthaltene Element inzwischen gelöscht wurde.

Kollisionsbehandlung für geschlossenes Hashing

- Methoden der Kollisionsbehandlung haben für das geschlossene Hashing große Bedeutung.
- Idee (offene Adressierung auch Sondierung bzw. re-hashing):
  - Neben $h = h_0$ weitere Streufunktionen h_1, … , h_{m-1} benutzen, die für einen gegebenen Schlüssel x die Zellen $h(x)$, $h_1(x)$ … inspizieren („sondieren“).
  - Sobald freie oder als deleted markierte Zelle gefunden: x dort eintragen.
  - Bei Suche nach xxwird die gleiche Folge von Zellen betrachtet, bis entweder x gefunden wird oder das erste Auftreten einer freien Zelle in der Folge anzeigt, dass xxnicht vorhanden ist.
  - Konsequenz:
    - Element y darf nicht einfach gelöscht und Zelle als frei markiert werden da Element x, das beim Einfügen durch Kollisionen an y vorbei geleitet worden ist, nicht mehr gefunden werden würde.
    - Also: betreffende Zelle als gelöscht markieren (deleted-Markierung)
    - Benutzter Speicherplatz wird also nicht wieder freigegeben.
    - Geschlossenes Hashing eignet sich nicht für sehr „dynamische“ Anwendungen mit vielen Einfügungen und Lösch-Operationen.
- Generelles Problem
  - Schrittweite = $hi(x) - h_{i+1}(x)$ muss relativ zur Tabellengröße so gewählt werden, dass jede Zelle erreicht wird (keine Lücken).
    - Bei Schrittweite +1 ist das trivialerweise erfüllt.
    - Bei gerader Tabellengröße und gerader Schrittweite werden die Hälfte der möglichen freien Plätze ignoriert.
    - Schrittweite und Tabellengröße müssen im Allgemeinen teilerfremd sein.

In der Praxis übliche Sondierungsverfahren

- Lineares Sondieren
  - Betrachte nacheinander Folgezellen von $h(x)$
  - Allgemeiner:
  - $h_i(x) = (h(x) + c \cdot i) \bmod m$ $(1 \le i \le m - 1)$
  - Bei annähernd voller Tabelle hat lineares Sondieren die Tendenz, lange Ketten von besetzten Zellen im Abstand cc zu bilden.
- Quadratisches Sondieren
  - $h_{i(}x) = (h(x) + i^2) \bmod m$
  - Weniger Clusterbildung, d.h. wegen kürzerer Ketten schnelleres Auffinden

Primär- und Sekundärkollisionen

- Primärkollision: $h(x) = h(y)$. Zwei Schlüsselwerte x und y konkurrieren um die selbe Zelle.
- Sekundärkollision: $h(x) = h_i(y)$ für $i > 1$. Schlüsselwert x findet seinen „rechtmäßigen“ Tabelleneintrag vor als bereits durch ein „Überlaufelement“ besetzt.

Doppelte Streuadressierung (engl. double hashing)

- Wähle zwei Streufunktionen h, h', die voneinander unabhängig sind.
- Für jede der beiden Streufunktionen tritt eine Kollision mit Wahrscheinlichkeit $1/m$ auf, für zwei Schlüssel x und y gilt also:
  - $P(h(x) = h(y)) = 1/m$
  - $P(h'(x) = h'(y)) = 1/m$
- h und h' sind unabhängig, wenn Doppelkollision nur mit Wahrscheinlichkeit $1/m^2$ auftritt:
  - $P(h(x) = h(y) \wedge h'(x) = h'(y)) = 1/m^2$
- In der Praxis ist dann eine gute Folge von Streufunktionen:
  - $h_i(x) = (h(x) + h'(x) \cdot i^2) \bmod m$

Eigenschaften des geschlossenen Hashings

- Vorteile:
  - Mehraufwand der Listen wird vermieden.
  - Weniger leere Stellen (Lücken) in Hashtabelle, dadurch sparsamerer Umgang mit Speicher.
- Nachteile:
  - Löschen ist nicht direkt möglich.
    - Statt Löschen muss deleted-Markierung an Einträgen stehen.
    - Sondieren wird ab markierten Behältern fortgesetzt.
  - Cluster-Bildung/Ballungen durch Kollisionen.
  - Im Allgemeinen wesentlich langsamer als offenes Hashing.
- Faustregel:
  - Der Belegungsfaktor BF sollte nicht größer als 80% sein, denn:
  - Bei BF = 50% sind im Mittel nur 2 Zugriffe nötig.
  - Bei BF = 90% sind im Mittel bereits 10 Zugriffe nötig.

Reorganisation von Streutabellen

- Dilemma:
  - Hoher Lastfaktor (> 80%) bedingt hohe Kollisionshäufigkeit und damit von $O(1)$ nach $O(n)$ steigenden Suchaufwand → Notwendigkeit der Anpassung der Tabellengröße mit aufwändiger Reorganisation.
  - Niedriger Lastfaktor (< 50%) verschwendet Speicherplatz.
- Reorganisation:
  - Globale Reorganisation
    - Anlegen einer neuen Streutabelle mit besser passender Größe.
    - Umspeichern aller Einträge mit neuer Streufunktion.
    - Nachteile: Zeitaufwand des Umspeicherns und temporär doppelter Speicherplatzbedarf.
  - Dynamische Reorganisation
    - Sobald der Lastfaktor einen Schwellwert erreicht, wird zusätzliches Array angelegt. Die Elemente der längsten Liste werden mit neuer Streufunktion auf das alte und neue Array umgespeichert.
    - Problem: Wie findet man die neue Streufunktion? → Literatur

Streutabellen in der Java-Standardbibliothek

- Streutabellen in der Java-Standardbibliothek implementieren (u. a.) das Interface Map.
- Eine Abbildung (engl. map) stellt eine Beziehung zwischen einem eindeutigen Schlüssel (Namen) und einem Wert her.

Interface java.util.Map

```
public interface Map<K,V> {
         // leert eine Map
         void clear();
         // prueft, ob ein bestimmter Schluessel enthalten ist
         boolean containsKey(Object key);
         // prueft, ob ein bestimmter Wert enthalten ist
         boolean containsValue(Object key);
        // liefert true, wenn die Map leer ist
        boolean isEmpty();
        // bestimmt die Anzahl der Eintraege
        int size();
        // liefert zu einem Schluessel den entsprechenden Wert
        V get(Object key);
        // fuegt einen neuen Eintrag hinzu; existiert bereits ein solcher
        // Schluessel, wird sein Wert aktualisiert
        V put(K key, V value);
        // alle Eintraege der uebergebenen Map werden uebernommen
         void putAll(Map<? extends K, ? extends V> t);

         // entfernt einen Eintrag mit dem angegebenen Schluessel
         V remove(Object key);
         ...
}
```

Klasse java.util.HashMap<K,V>

Diese Klasse der Java-Bibliothek ist eine Streutabelle (Typparameter K(ey) und V(alue)), die Kollisionen mit offenem Hashing (Verkettung) behandelt.

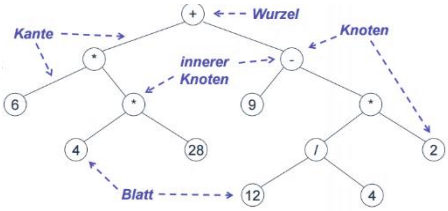
Klasse java.util.HashMap<K,V>

```
HashMap<String, Double> preisliste
 = new HashMap<String, Double>(151); // Initialgroesse
// Eintragen
preisliste.put("Mixer", new Double( 79.95));
preisliste.put("Mikrowelle", new Double( 230.90));
preisliste.put("Ferrari", new Double(400399.99));
...
// Auslesen
double preis = preisliste.get("Mixer").doubleValue();
```

java.lang.Object definiert eine Methode hashCode(), die einen eindeutigen int-Schlüssel für Objekte liefert.

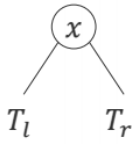
Bäume

Bäume



- Listen: jedes Element hat einen Nachfolger
- Baum: jedes Element hat mehrere Nachfolger
- weitere Beispiele für Baumstrukturen:
  - Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte
  - Klammerstrukturen
- Höhe eines Knotens:
  - Die Höhe h eines Knotens entspricht der Anzahl der Kanten des längsten Pfades zu einem von diesem Knoten aus erreichbaren Blatt (= Knoten ohne Nachfolger).
- Die Höhe eines Baumes ist die Höhe der Wurzel
- Bäume enthalten keine Zyklen.
- Verzweigungsgrad = Zahl der möglichen Nachfolger pro Element
- Ein Baum mit n Knoten hat n-1 Kanten

Binärbäume



- Jeder Knoten hat nur zwei Nachfolger (Verzweigungsgrad: 2).
- Definition:
  - Der leere Baum ist ein Binärbaum. (Bezeichnung: ⋄; graphisch: ■)
  - Wenn x ein Knoten ist und T_l u. T_r Binärbäume sind, dann ist auch das Tripel (T_l, x, T_r) ein Binärbaum T.
- x heißt Wurzel, T_l linker Teilbaum, T_r rechter Teilbaum von T.
- Die Wurzeln von T_l und T_r heißen Kinder/Söhne von x, x ist ihr Elternknoten/Vaterknoten.
- Ein Knoten, dessen beide Kinder leere Bäume sind, heißt Blatt.
  - Teilbaum in linearer Notation: ((⋄, 12,⋄ ,/, ⋄, 4,⋄) ,∗, ⋄, 2,⋄)
- Die maximale Höhe eines Binärbaums mit n Knoten ist $n - 1$.
  - Dieser Fall tritt auf, wenn der Baum zu einer Liste entartet.
- Die minimale Höhe eines Binärbaums mit n Knoten ist $\lfloor \log_2 n \rfloor$.
- Falls alle inneren Knoten zwei Kinder haben, so hat ein Binärbaum mit n + 1 Blättern genau n innere Knoten.

Abstrakter Datentyp *Bintree*

adt $BinTree$
sorts $BinTree, E, Boolean$
ops

$create$:		$\rightarrow BinTree$
$maketree$:	$Bintree \times E \times BinTree$	$\rightarrow BinTree$
$value$:	$BinTree$	$\rightarrow E$
$left, right$:	$BinTree$	$\rightarrow BinTree$
$leaf$	E	$\rightarrow BinTree$
$isEmpty$:	$BinTree$	$\rightarrow Boolean$

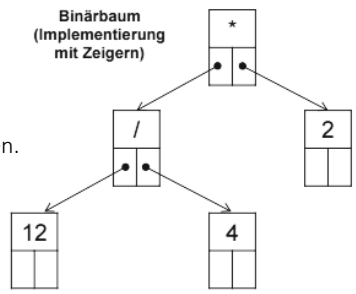
axs

$left(maketree(l,e,r)) = l$
$right(maketree(l,e,r)) = r$
$value(maketree(l,e,r)) = e$
$leaf(v) = maketree(create, v, create)$
$isEmpty(create) = true$
$isEmpty(maketree(l,e,r)) = false$
…

end $BinTree$

Anwendungsbeispiel

- Hilfsfunktion zum Erzeugen eines Blatts:
  - leaf(v) = maketree(create,v,create)



Implementierung des ADTs *BinTree* mit Referenzen

- Implementierung ist ähnlich der von einfach oder doppelt verketteten Listen.
- Jeder Knoten besitzt zwei Referenzen auf die beiden Teilbäume.

Implementierung des ADTs *B*inTree mit Referenzen

```
class Entry<E> {
        E element;
        Entry left, right;
        public Entry(Entry<E> l, E x, Entry<E> r) {
                left = l;
                element = x;
                right = r;
        }
        …
}
class BinTree<E> {
        Entry<E> root;
        …
        // Konstruktor und Methoden
        // der Klasse BinTree
}
```

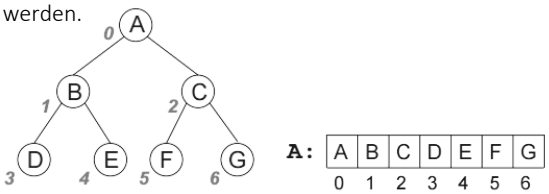
Implementierung des ADTs *B*inTree mit Array-Indizes als Zeigern

Implementierung mit mehrdimensionalem Array und Indizes, ohne explizite Verweise

A	B	C	D	E	F	G	
0	1	2	3	4	5	6	zugeordneter Index
1	3	-1	-1	5	-1	-1	Index linkes Kind
2	4	-1	-1	6	-1	-1	Index rechtes Kind

Implementierung des ADTs *B*inTree durch Einbettung in Array

Betrachte vollständigen Binärbaum der Höhe h: Strategie: durch die Position in einem Feld kann die Position der Nachfolger berechnet werden.



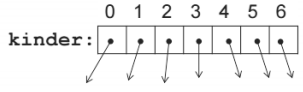
- Baum wird eingebettet in Array A der Länge $2^{h+1} - 1$ (hier: = 7)
  - speichere Wurzel in A[0]
  - linker Nachfolger von Knoten ii steht in A[2*i+1]
  - rechter Nachfolger von Knoten ii steht in A[2*i+2]
  - Vorgänger eines Knotens k steht in A[(k-1)/2]

(Allgemeine) Bäume

- Anzahl der Kinder eines Knotens ist beliebig.
- Keine „festen Positionen“ für Kinder (wie left, right).
- Rekursive Definition:
  - Der leere Baum ist ein (allgemeiner) Baum.
  - Wenn x ein Knoten ist und $T1, \dots, Tk$ Bäume sind, dann ist auch das $(k + 1)$-Tupel $(x, T1, \dots, Tk)$ ein Baum.
- Begriffe der Binärbäume lassen sich übertragen (bis auf die positionsbezogenen, z. B. linker Teilbaum)
- Grad eines Knotens: Anzahl seiner Kinder (für Knoten x: k)
- Grad eines Baumes: maximaler Grad aller Knoten im Baum)

Implementierung allgemeiner Bäume mit Hilfe von ArrayLists

Jeder Knoten speichert eine ArrayList von Referenzen auf seine Kinder.



`kinder.get(i)` liefert Referenz auf das i-te Kind.

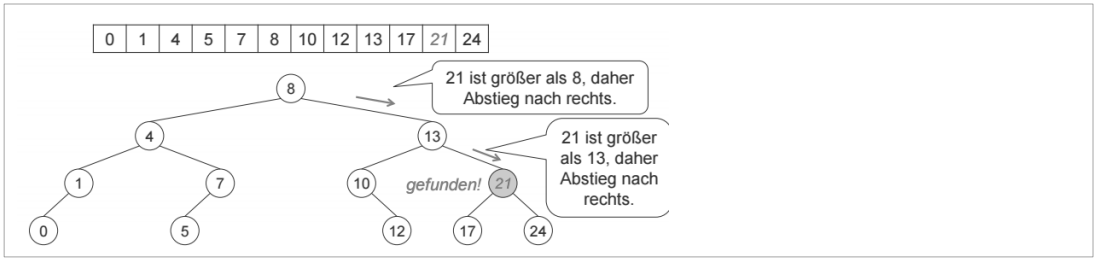
Implementierung allgemeiner Bäume über Binärbäume

- Struktur eines Knotens ist gleich der in einem Binärbaum.
- Aber andere Interpretation der Referenzen:
  - Statt Referenz auf den linken Teilbaum:Referenz auf das am weitesten links stehende Kind.
  - Statt Referenz auf den rechten Teilbaum: Referenz auf das rechte Geschwisterkind.
  - Es ist kein direkter Zugriff auf das i-te Kind möglich.

Suchbäume

Bäume kann man sich als strukturelles Abbild des Teile-und Herrsche-Prinzips vorstellen, wenn man beim Zugriff jeweils pro Knoten nur ein Kind weiterverfolgt.

Eine solche Datenstruktur in Form eines Baums wird Suchbaum genannt. Ein binärer Suchbaum hat max. zwei Kinder je Knoten.



Binäre Suchbäume

- Sei T<E> ein Baum. Eine Knotenmarkierung ist eine Abb. μ: $E \rightarrow D$ für irgendeinen geordneten Wertebereich D.
- Anschaulich:
  - Es wird jedem Knoten E von TTein Wert aus D zugeordnet, der es ermöglicht, die Knotenwerte zu vergleichen.
  - Direkter Vergleich möglich z. B. bei numerischen Werten, Strings
  - Was jedoch z. B. bei Symbolen? Mit μ Ordnungsrelation definieren.
- Ein knotenmarkierter Binärbaum T heißt binärer Suchbaum genau dann, wenn für jeden Teilbaum von T‘ von T, T‘= (Tl,y,Tr) gilt:
  - $\forall x \in Tl$: μ (x) < $\mu(y)$
  - $\forall z \in Tr$: μ (z) > $\mu(y)$

Abstrakter Datentyp SBintree

adt $SBinTree$
sorts $SBinTree, E, Boolean$
ops

$create$: $\rightarrow SBinTree$
$insert$: $SBinTree \times E \rightarrow SBinTree$
$find$: $SBinTree \times E \rightarrow Boolean$
$delete$: $SBinTree \times E \rightarrow SBinTree$

nicht-öffentliche Hilfsfunktion $maketree$ (s. ADT $BinTree$)

Ein binärer Suchbaum enthält keine doppelten Einträge.

axs

$$insert(create, x) = maketree(create, x, create)$$

$$insert(maketree(l, e, r), x) = \begin{cases} maketree(l, e, r) & \text{falls } x = e \\ maketree(insert(l, x), e, r) & \text{falls } x < e \\ maketree(l, e, insert(r, x)) & \text{falls } x > e \end{cases}$$

Einfügen im passenden Teilbaum

$$find(create, x) = false$$

$$find(maketree(l, e, r), x) = \begin{cases} true & \text{falls } x = e \\ find(l, x) & \text{falls } x < e \\ find(r, x) & \text{falls } x > e \end{cases}$$

...

end $SBinTree$

Implementierung von binären Suchbäumen

- hier:
  - Implementierung mit doppelter Verzeigerung
  - Nicht für alle Verwendungen erforderlich, aber vieles (z. B. Iterator) wird leichter.

```
public class SBinTree<E extends Comparable<E>> {

        private class Entry {
                Entry left; // linker Nachfolger
                Entry right; // rechter Nachfolger
                Entry parent; // Elternknoten

                E value; // Nutzdaten

                public Entry(E val, Entry parent) {
                        left = null;
                        right = null;
                        value = val;
                        this.parent = parent;
                }
        }
        // Wurzel des Binaerbaums
        Entry root;

        public SBinTree() {
                root = null;
        }
 ...
```

Suche im binären Suchbaum: rekursive Implementierung

äußere Methode contains

```
public boolean contains(E val) {
       return containsRec(val, root);
}
```

innere Methode containsRec

```
private boolean containsRec(E val, Entry node) {
        // Baum leer?
        if (node == null) return false;
        int result = val.compareTo(node.value);
        if (result == 0) return true; //gefunden
        if (result < 0) {
                // Wert des Suchbaumknotens node "groesser" als Wert val,
               // im linken Teilbaum weitersuchen
               return containsRec(val, node.left);
        } else {
               // Wert des Suchbaumknotens node "kleiner" als Wert val,
                // im rechten Teilbaum weitersuchen
               return containsRec(val, node.right);
        }
}
```

Suche im binären Suchbaum: iterative Implementierung

```
public boolean contains(E val) {
        Entry node = root;
        while (node != null) {
                int result = val.compareTo(node.value);
                if (result == 0) return true; //gefunden
                if (result < 0) {
                // Wert des Suchbaumknotens node "groesser" als Wert val, links
               // weitersuchen
        node = node.left;
        } else {
                // Wert des Suchbaumknotens node
                // "kleiner" als Wert val, rechts weitersuchen
                 node = node.right;
        }
        }
        return false;
}
```

Einfügen in einen binären Suchbaum

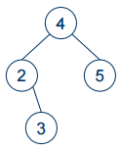
- Aufgabe: Füge 12 in gegebenen Binärbaum ein.
- Grundidee:
  - Suche nach einzufügendem Wert im Binärbaum.
  - Wenn der einzufügende Wert nicht gefunden wird, dann füge ihn als neuen Nachfolger des Knotens ein, an dem die Suche erfolglos endete (und gib true zurück).
  - Sonst: Gib false zurück (Suchbäume enthalten keine Duplikate).

Einfügen in einen binären Suchbaum: iterative Implementierung

```
public boolean add(E val) {
        if (root == null) { // Sonderfall: Einfuegen in leeren Baum
              root = new Entry(val, null);
              return true;
        }
        Entry node = root;
        Entry dragPtr = null; // Schleppzeiger
        int result = 0;
        while (node != null) {
              result = val.compareTo(node.value);
              if (result == 0) return false;
              dragPtr = node; // zeigt auf letzten Knoten
              if (result < 0)
                    node = node.left;
              else
                    node = node.right;
        }
        // dragPtr zeigt auf das Blatt, an das anzu-
        // haengen ist. Neuen Knoten an Blatt anfuegen.
        Entry newNode = new Entry(val, dragPtr);
        if (result < 0) { // result hat Ergebnis des letzten Vergleichs
              dragPtr.left = newNode;
        } else {
              dragPtr.right = newNode;
        }
        return true;
}
```

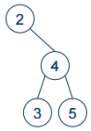
Einfügereihenfolge bedingt das Aussehen

4, 2, 5, 3 nacheinander in zuvor leeren Baum einfügen



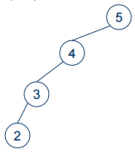
- Höhenunterschiede am Beispiel:
- Knoten 4: ±1, da $h(2) = 1$, $h(5) = 0$
- Knoten 2: ±1, da $h(\text{null}) = -1$, $h(3) = 0$
- ausgewogener/balancierter Baum, da rechte und linke Unterbäume stets etwa die gleiche Höhe (maximaler Unterschied ±1) haben.
- sogar vollständig ausgewogener Baum, da sich die Knotenanzahl in Unterbäumen auch um maximal ±1 unterscheiden.

2, 4, 3, 5 nachHöhenunterschiede am Beispiel:



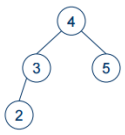
- Knoten 4: 0, da $h(3) = 0$, $h(5) = 0$
- Knoten 2: ±2, da $h(\text{null}) = -1$, $h(4) = 1$
- nicht ausgewogener/unbalancierter Baum.einander in zuvor leeren Baum einfügen

5, 4, 3, 2 nacheinander in zuvor leeren Baum einfügen



- Höhenunterschiede am Beispiel:
- Knoten 5: ±3, da $h(4) = 2$, $h\ (\text{null}) = -1$
- nicht ausgewogener/unbalancierter Baum.
- Extremer Fall, da vorsortierte Eingabe: Suchen, Einfügen haben Aufwand $O(n)$.

4, 3, 2, 5 nacheinander in zuvor leeren Baum einfügen:



- ausgewogener/balancierter Baum.
- vollständig ausgewogener Baum.

Löschen aus einem binären Suchbaum

- Um einen Knoten zu löschen, muss man diesen zuerst finden. Dann sind folgende drei Fälle zu unterscheiden:
  - Löschen eines Blattknotens
    - einfach: Knoten wird entfernt.
  - Löschen eines Knotens mit nur einem Nachfolger
    - einfach: der eine Nachfolger rückt auf.
  - Löschen eines Knotens mit zwei Nachfolgern
    - nicht so einfach: geeignete Strategie erforderlich, um Suchbaumeigenschaft zu erhalten.

Zwei alternative Strategien

- Variante 1
  - Finde im rechten Unterbaum des zu löschenden Knotens den Knoten mit minimalem Wert (= den am weitesten links stehenden Knoten) und merke Dir dessen Wert.
  - Lösche diesen Knoten mit minimalem Wert aus dem Baum.
  - Überschreibe den Wert des ursprünglich zu löschenden Knotens mit dem gemerkten minimalen Wert.
- Variante 2
  - Finde im linken Unterbaum des zu löschenden Knotens den Knoten mit maximalem Wert (= den am weitesten rechts stehenden Knoten) und merke Dir dessen Wert.
  - Lösche diesen Knoten mit maximalem Wert aus dem Baum.
  - Überschreibe den Wert des ursprünglich zu löschenden Knotens mit dem gemerkten maximalen Wert.

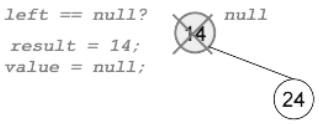
Löschen aus einem binären Suchbaum: rekursive Implementierung

```
...
 private class Entry {
         ...
         private E removeMin() {
                E result;
               if (left == null) {
                       result = value;
                       value = null;
               } else {
                       result = left.removeMin();
                       if (left.value == null) {
                              left = left.right;
                              if (left != null) {
                                      left.parent = this;
                               }
                        }
                }
                return result;
         }
 ..
```

- Hilfsmethode, die minimalen Schlüsselwert in nichtleerem Baum ermittelt und den zugehörigen Knoten löscht.
- Nach rekursivem Abstieg muss im Elternknoten des zu löschenden Knotens Zeiger umgesetzt werden.
- Trick:
  - Im zu löschenden Knoten wird dessen Wert (value) auf null gesetzt.
  - Diese Kennzeichnung kann beim Aufstieg überprüft werden.
- Achtung: Sonderfall „Wurzel des Teilbaums ist zu löschender Min-Knoten“ muss noch separat verarbeitet werden

removeMin am Beispiel

- Achtung: Sonderfall
- zu löschender Knoten ist Wurzel des betrachteten Teilbaums.
- Dieser Sonderfall muss an der aufrufenden Stelle noch separat behandelt werden



Löschen aus einem binären Suchbaum: rekursive Implementierung

```
        ...
         public Entry remove(E val) {
                int result = val.compareTo(value);
                if (result < 0) { // val < value
                       if (left != null) {
                              left = left.remove(val);
                              if (left != null) left.parent = this;
                        }
                        return this;
                } else if (result > 0) { // val > value
                       if (right != null) {
                              right = right.remove(val);
                              if (right != null) right.parent = this;
                       }
                       return this;
                } else { // val == value
                       if ((left == null) && (right == null)) return null;
                        else if (left == null) return right;
                        else if (right == null) return left;
                        else { // der aktuelle Knoten hat zwei Kinder
                              value = right.removeMin();
                               if (right.value == null) {
                                     right = right.right;
                                     if (right != null) right.parent = this;
                               }
                              return this;
                       }
                }
         }
 } // Ende innere Klasse Entry
```

Wurzel muss umgesetzt werden, da sie sich durch die Löschoperation geändert haben könnte.

```
...
 public void remove(E val) {
       root = root.remove(val);
       root.parent = null;
 }
 ..
```

Aufwandsbetrachtung

- Suchen, Einfügen und Löschen folgen jeweils einem einzigen Pfad im Baum von der Wurzel zu einem Blatt oder einem inneren Knoten.
- Aufwand ist proportional zur Länge des Pfades:
  - Balancierte Bäume: Pfadlänge $O(\log n)$
  - (Zur Liste) degenerierte Bäume: Pfadlänge $O(n)$; entstehen
  - insbesondere dann, wenn man die Werte in sortierter Reihenfolge einfügt.
  - Beispiel: 37, 52, 70, 92 in binären Suchbaum einfügen Aufwand für Aufbau des gesamten Baumes $O(n^2)$ also: schlechtes Worst-Case-Verhalten!

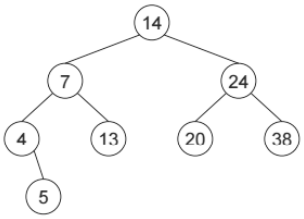
Besuchsreihenfolgen für die Knoten eines Baumes

- Drei grundsätzliche Besuchsreihenfolgen: inorder, preorder, postorder
- Auffassbar als Operationen, die Baum in eine Sequenz überführen

```
private class Entry {
 ...
        public void inorder() {
               if (left != null) left.inorder();
               System.out.println("Value:" + value);
               if (right != null) right.inorder();
        }

        public void preorder() {
               System.out.println(value);
               If (left != null) left.preorder();
               if (right != null) right.preorder();
        }
       public void postorder() {
               if (left != null) left.postorder();
               if (right != null) right.postorder();
               System.out.println(value);
        }
 }
```

Besuchsreihenfolgen für die Knoten eines Baumes

- Inorder
  - Besuche linken Unterbaum
  - Aktion auf aktuellem Knoten
  - Besuche rechten Unterbaum
- Preorder
  - Aktion auf aktuellem Knoten
  - Besuche linken Unterbaum
  - Besuche rechten Unterbaum
- Postorder
  - Besuche linken Unterbaum
  - Besuche rechten Unterbaum
  - Aktion auf aktuellem Knoten



- Beispiel:
  - inorder-Durchlauf („links, mitte, rechts“):
    - 4, 5, 7, 13, 14, 20, 24, 38
  - preorder-Durchlauf („mitte, links, rechts“):
    - 14, 7, 4, 5, 13, 24, 20, 38
  - postorder-Durchlauf („links, rechts, mitte“):
    - 5, 4, 13, 7, 20, 38, 24, 14

Klassifikation von Suchbäumen

- Anzahl m der Kinder pro Knoten (Knotengrad):
  - Binärbaum: Anzahl 0 ≤ mm ≤ 2
  - allgemeiner Baum (auch: Vielwegbaum): Anzahl beliebig
- Speicherort der Nutzdaten:
  - natürlicher Baum: Nutzdaten werden bei jedem Knoten gespeichert.
  - Blattbaum/hohler Baum: Nutzdaten werden nur in den Blattknoten gespeichert; innere Knoten dienen nur der Verzweigung.
- Ausgewogenheit/Ausgeglichenheit/Balanciertheit:
  - ausgewogener/ausgeglichener/balancierter Baum:
    - Für alle Knoten gilt: Die Höhen aller ihrer Unterbäume unterscheiden sich höchstens um Δn (meistens ±1) (sog. Höhenbalance).
    - vollständig ausgewogen: Die Anzahl der Knoten in den Unterbäumen unterscheidet sich höchstens um 1.
  - nicht ausgewogener/unbalancierter Baum: keine derartige Einschränkung

AVL-Bäume

- Binäre Suchbäume
  - gutes Durchschnittsverhalten
  - aber: sehr schlechter Worst Case, wenn Elemente in bereits sortierter Reihenfolge eingefügt werden.
- Alternative: AVL-Bäume
  - ermöglichen Einfügen, Löschen und Suchen in $O(\log n)$ auch im Worst Case,
  - sind balancierte, binäre Suchbäume.
  - Grundidee: sog. Strukturinvariante für binären Suchbaum formulieren und diese bei jeder Aktualisierungen (Einfügen, Löschen) aufrecht erhalten

Strukturinvariante der AVL-Bäume (AVL-Bedingung)

- Definition: Ein AVL-Baum ist ein binärer Suchbaum, in dem sich die Höhen seiner zwei Teilbäume höchstens um 1 unterscheiden.
- oder anders ausgedrückt:
  - AVL-Bedingung: Sei ee ein beliebiger Knoten eines binären Suchbaumes, und $h(e)$ die Höhe des Teilbaums mit Wurzel e, dann gilt für die beiden Kinder e.left und $e.\mathrm{right}$ von e: $|h(e.\mathrm{left}) - h(e.\mathrm{right})| \leq 1$
- Konsequenz: Wird diese Strukturinvariante durch Aktualisierungen verletzt, so muss sie durch eine Rebalancieroperation wieder hergestellt werden.

Herstellung von balancierten/ausgewogenen Bäumen

- Möglichkeit 1: globale Reorganisation
  - Alle Elemente aus einem unbalancierten/unausgewogenen Baum auslesen, diese dann so umordnen, dass die neu sortierte Folge beim Einfügen in einen frischen Baum Balanciertheit/Ausgewogenheit liefert.
  - Nachteil: aufwändig
- Möglichkeit 2: lokale Reorganisation durch Rebalancieroperation
  - Balanciertheit/Ausgewogenheit bei jeder Einfüge-/Löschoperation durch lokalen Umbau des Baums sichern.

- Manipulation jeweils einiger Knoten in der Nähe der Wurzel des aus der Balance geratenen Teilbaums: Teilbaum anheben/absenken, um Höhen anzugleichen.
- Das ist die bessere Strategie

Aktualisierung eines AVL-Baums Einfügen/Löschen

- Einfüge-/Löschoperation wird zunächst genauso ausgeführt wie in gewöhnlichem binärem Suchbaum.
- Einfügeoperation
  - Fügt in jedem Fall neues Blatt hinzu.
  - Das neu erzeugte Blatt (= aktuelle Position) gehört zu allen Teilbäumen von Knoten, die auf dem Pfad von der Wurzel zu diesem Blatt liegen.
  - Durch Einfügung können die Höhen dieser Teilbäume um 1 wachsen.
- Löschoperation
  - Entfernt irgendwo im Baum einen Knoten: Blatt oder innerer Knoten.
  - Der Elternknoten des gelöschten Knotens sei aktuelle Position (Sonderfall Wurzel beachten).
  - Teilbäume auf dem Pfad von der Wurzel zu diesem Elternknoten sind betroffen: Deren Höhe könnte sich um 1 verringert haben.
- Danach folgt in beiden Fällen eine Rebalancierphase falls die AVL-Eigenschaft wieder hergestellt werden muss.

Aktualisierung eines AVL-Baums Rebalancierphase

- Rebalancierphase
  - Man läuft von aktueller Position aus den Pfad zur Wurzel zurück.
  - In jedem Knoten wird dessen Balance geprüft: falls Knoten aus der Balance geraten ist, so wird das durch eine der im Folgenden beschriebenen Rebalancieroperationen korrigiert.
  - Wir können also davon ausgehen, dass alle Knoten unterhalb des gerade betrachteten (aus der Balance geratenen) Knotens selbst balanciert sind.
- Hinweis: Suchen im AVL-Baum erfolgt genau so, wie in gewöhnlichem Suchbaum.

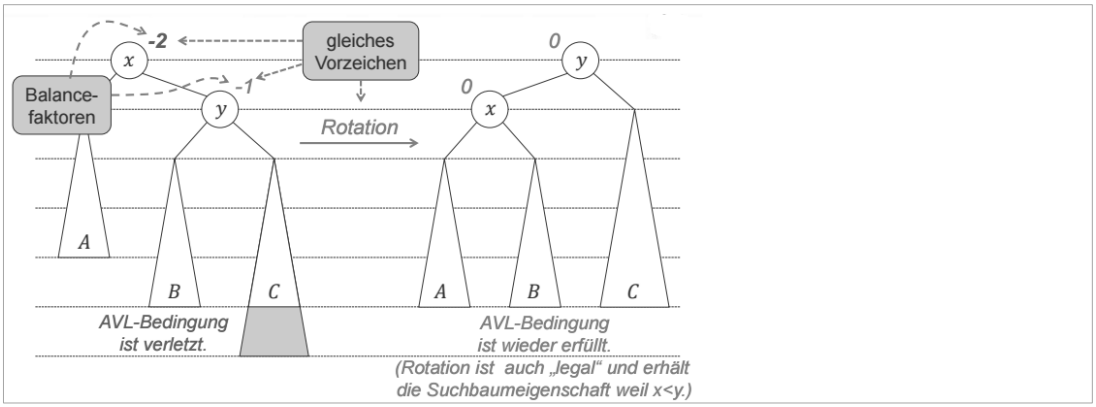
Bestimmung der Höhe und des Balancefaktors eines Knotens

- Balancefaktor bf eines Knotens: Differenz zwischen der Höhe des linken Teilbaums und der Höhe des rechten Teilbaums.
- Induktionsanfang:
  - Höhe h und Balancefaktor bf eines leeren Teilbaums sind -1. (Damit sind Höhe h und Balancefaktor bf eines Blattes 0).
- Induktionshypothese/-annahme:
  - Höhe und Balancefaktoren von Binärbäumen mit weniger als n Knoten können berechnet werden.
- Induktionsschluss:
  - Betrachte die Wurzel eines Baums mit n Knoten.
  - Dessen Teilbäume haben weniger als n Knoten.
  - Laut Induktionshypothese können deren Höhen und Balancefaktoren berechnet werden.
  - Dann gilt für die Höhe h der Wurzel:
  - h (Wurzel) = 1 + max (h (linker Teilbaum) , h (rechter Teilbaum))
  - Und damit für den Balancefaktor bf:
  - bf(Wurzel) = h (linker Teilbaum) – h(rechter Teilbaum)
- Aufwandsbetrachtungen
  - Berechnung von Höhe und Balancefaktor für den ganzen Baum:
    - Die rekursive Implementierung steigt bis zu den Blättern ab und berechnet dann Höhe und Balancefaktoren „von unten nach oben“ im Baum.
    - Jeder Knoten wird dabei einmal besucht: Aufwand: $O(n)$.
  - Neuberechnung bei einer Änderung:
    - Nimmt man im Baum eine Änderung vor (Hinzufügen eines neuen Knotens oder Löschen eines Knotens), dann ändern sich die Höhenangaben und Balancefaktoren nur bei den Knoten, die auf dem Pfad von der Änderungsstelle bis zur Wurzel liegen.
    - Die Korrektur nach einer Änderung hat daher den Aufwand $O(\log 2n)$.

Rebalancieren – Fall a: nach Einfügung in einen AVL-Baum

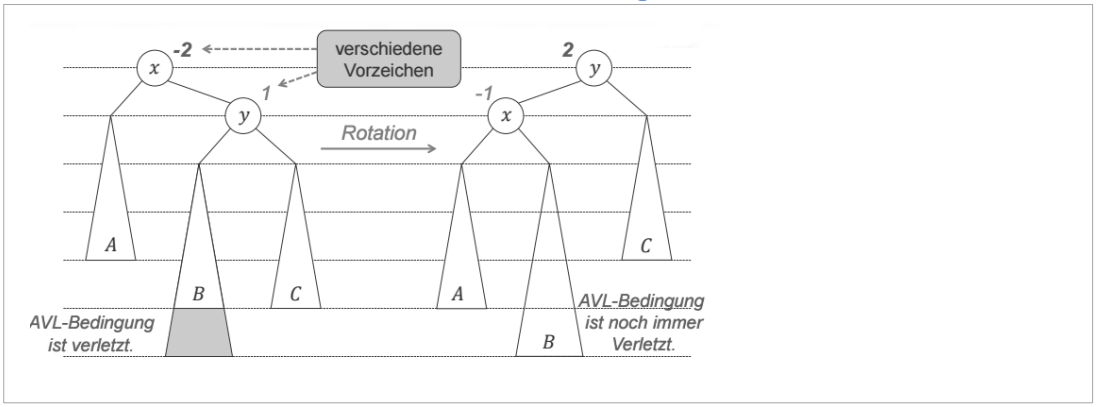
- Annahme:
  - Gerade betrachteter Knoten (Teilbaum) ist durch Einfügung aus der Balance geraten.
  - Also ist die Höhe einer seiner Teilbäume um 1 gewachsen.
  - O. B. d. A. (=ohne Beschränkung der Allgemeinheit) sei der rechte Teilbaum um 2 tiefer (bzw. höher) als der linke.
  - Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.
- Zwei Unterfälle unterscheidbar:
  - a1: Innerhalb des rechten Teilbaums ist der rechte Teilbaum gewachsen.
  - a2: Innerhalb des rechten Teilbaums ist der linke Teilbaum gewachsen.

Fall a1: Innerhalb des rechten Teilbaums ist der rechte Teilbaum gewachsen.



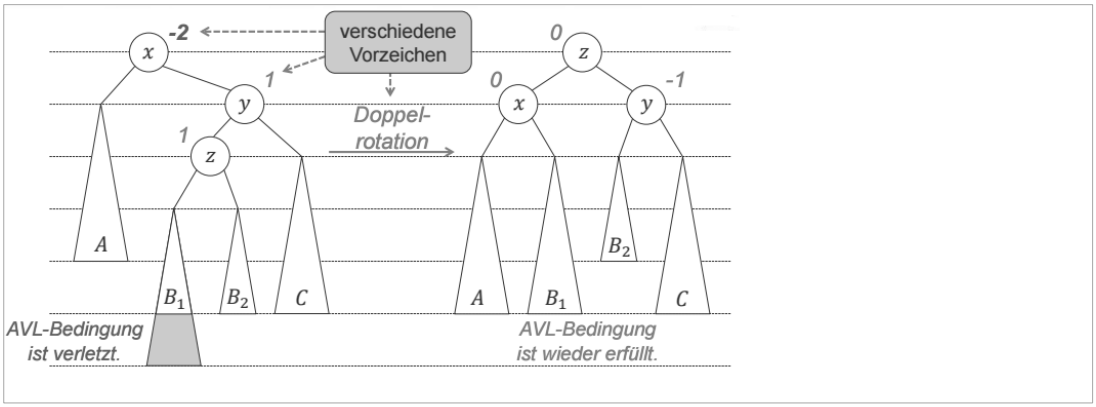
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall a2: Innerhalb des rechten Teilbaums ist der linke Teilbaum gewachsen



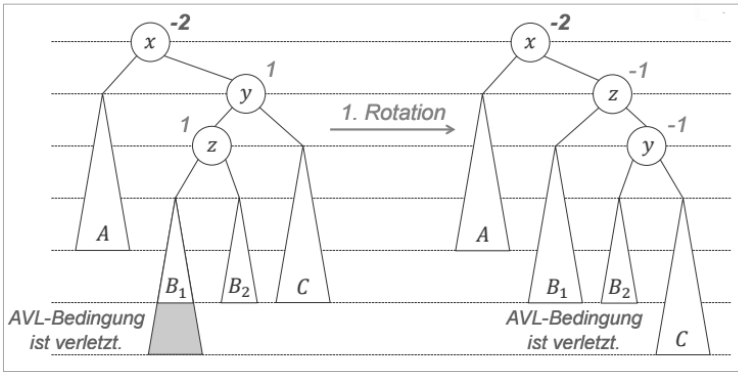
Frage: Was tun? Offenbar genauere Analyse von Fall a2 erforderlich.

Fall a2.1: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums gewachsen



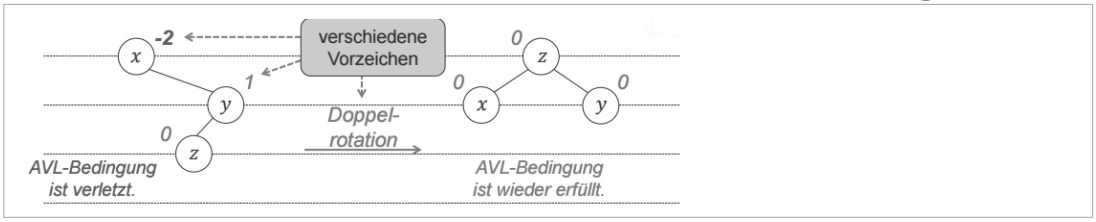
Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Fall a2.1: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums gewachsen. (II)



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

Fall a2.2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum des linken Teilbaums gewachsen.



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren. Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

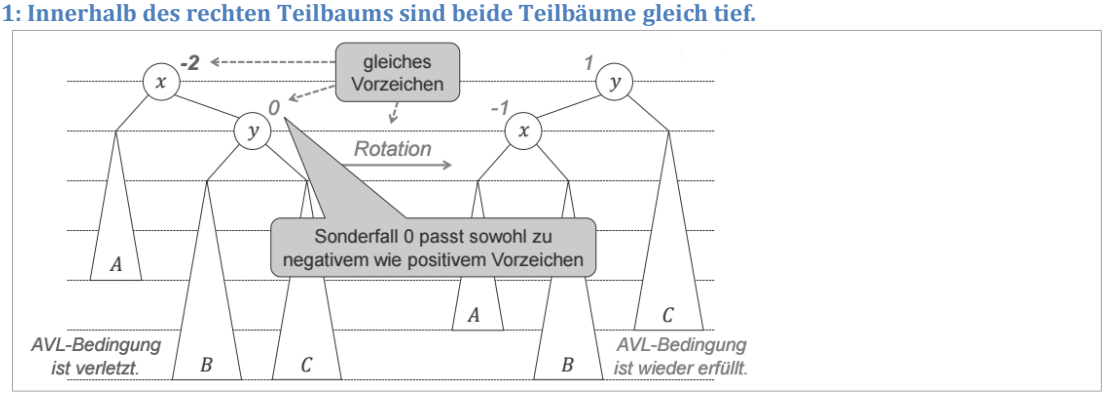
Rebalancieren – Fall a: nach Einfügung in einen AVL-Baum

- also: Nach einer Einfügung genügt eine einzige Rotation oder Doppelrotation, um die Strukturinvariante des AVL-Baums (AVL-Bedingung) wiederherzustellen (und dabei die Suchbaumeigenschaft zu erhalten).
- Beweis (für Interessierte):
  - Eine Rotation oder Doppelrotation im ersten aus der Balance geratenen Knoten e auf dem Pfad von der aktuellen Position zur Wurzel sorgt dafür, dass der Teilbaum mit Wurzel e die gleiche Höhe hat, wie vor der Einfügung.
  - Also haben auch alle Teilbäume, deren Wurzeln Vorfahren von e sind, die gleiche Höhe und keiner dieser Knoten kann jetzt noch unbalanciert sein.

Rebalancieren – Fall b: nach Löschung aus einem AVL-Baum

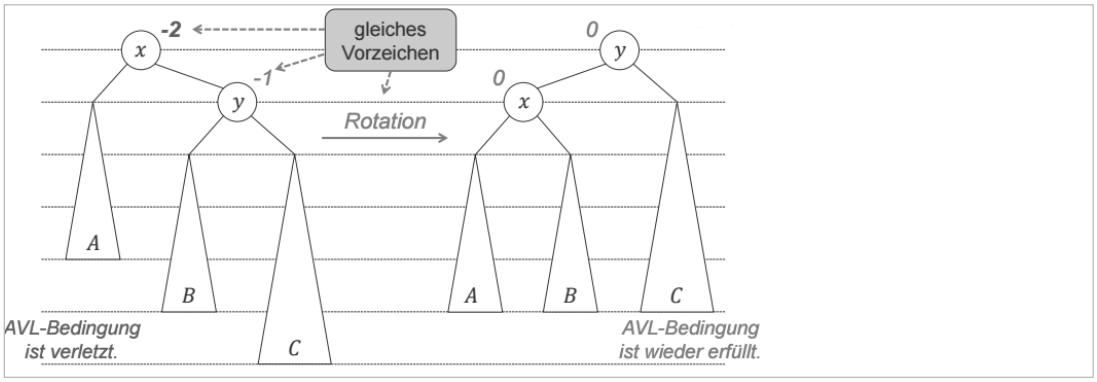
- Annahme:
  - Gerade betrachteter Knoten (Teilbaum) ist durch Löschung aus der Balance geraten.
  - Also ist die Höhe einer seiner Teilbäume um 1 geschrumpft.
  - O. B. d. A. sei der rechte Teilbaum um 2 tiefer (bzw. höher) als der linke.
  - Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.
- Drei Unterfälle (nach Löschung) unterscheidbar:
  - b1: Innerhalb des rechten Teilbaums sind beide Teilbäume gleich tief.
  - b2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum tiefer.
  - b3: Innerhalb des rechten Teilbaums ist der linke Teilbaum tiefer.

Fall b1: Innerhalb des rechten Teilbaums sind beide Teilbäume gleich tief.



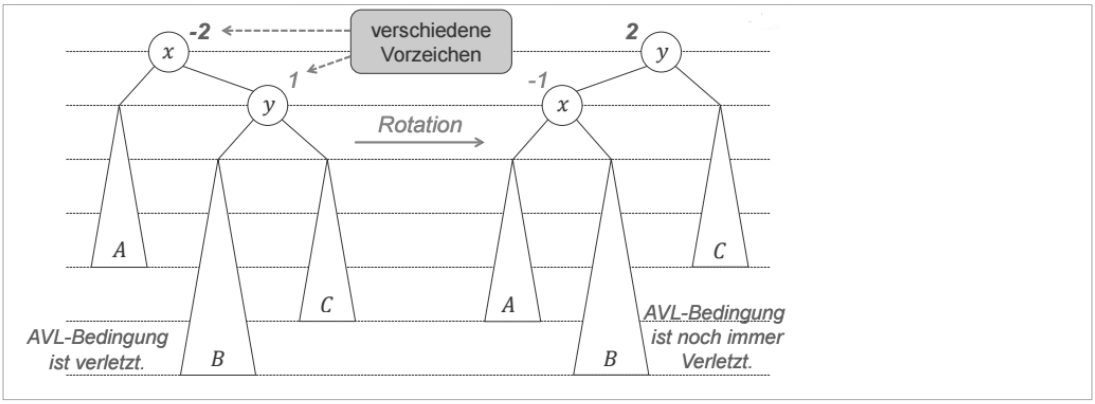
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall b2: Innerhalb des rechten Teilbaums ist der rechte Teilbaum tiefer.



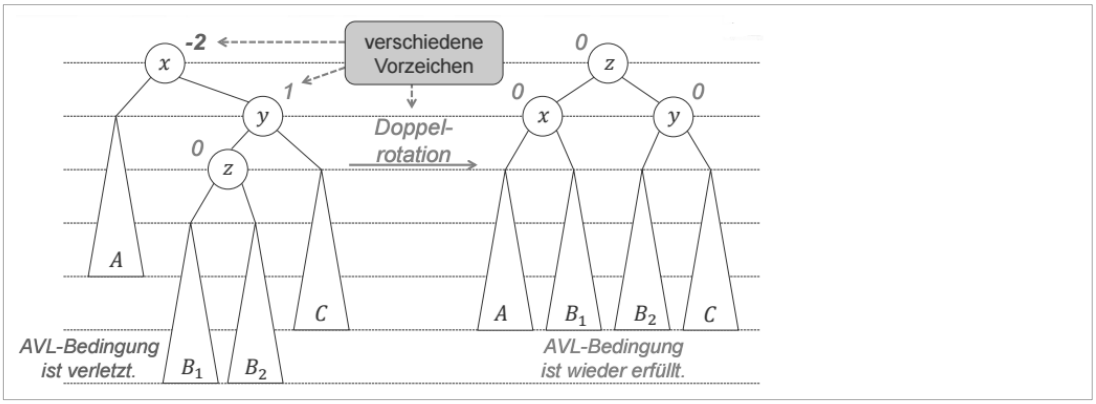
hier: Einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn ausreichend, um Balance wiederherzustellen.

Fall b3: Innerhalb des rechten Teilbaums ist der linke Teilbaum tiefer.



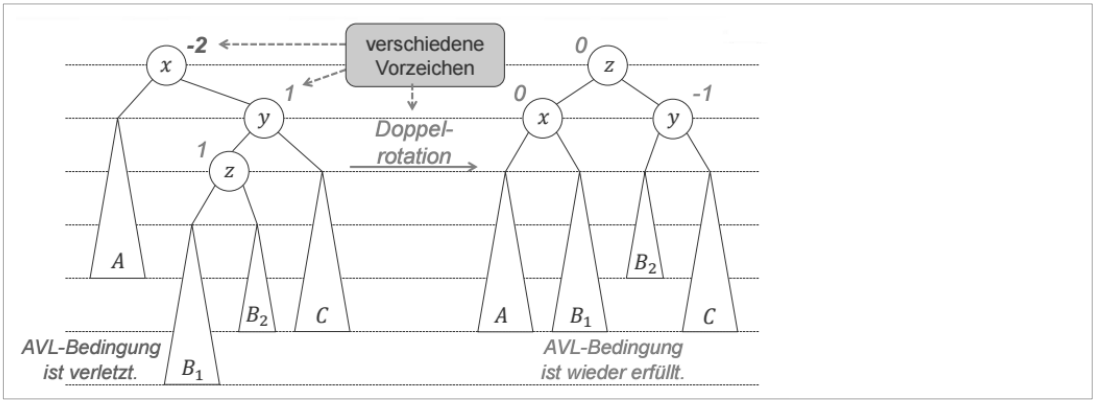
Frage: Was tun? Offenbar wieder genauere Analyse von Fall b3 erforderlich (vgl. a2).

Fall b3.1: Innerhalb des rechten Teilbaums sind beide Teilbäume des linken Teilbaums gleich tief.



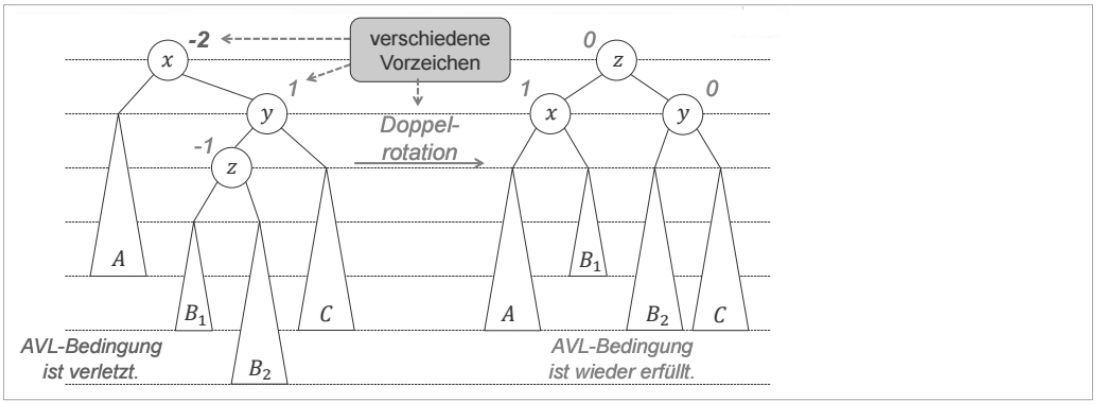
Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren.Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren.

Fall b3.2: Innerhalb des rechten Teilbaums ist der linke Teilbaum des linken Teilbaums tiefer.



Zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren Danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Fall b3.3: Innerhalb des rechten Teilbaums ist der rechte Teilbaum des linken Teilbaums tiefer.



zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren danach gesamten (Teil-) Baum gegen Uhrzeigersinn rotieren

Rebalancieren – Fall b: nach Löschung aus einem AVL-Baum

also: Ein durch eine Löschoperation aus der Balance geratener (Teil-) Baum kann durch eine Rotation oder Doppelrotation wieder ausgeglichen werden. Es kann aber erforderlich sein, auch Vorgängerbäume bis hin zur Wurzel zu rebalancieren. Die

Zwischenfazit und Hinweis zur Symmetrie

Zwei grundlegende Fälle unterscheidbar:

- Balancefaktoren der Wurzel des aus der Balance geratenen (Teil-) Baums und deren Kind y auf dem Pfad in den zu tiefen Teilbaum haben gleiche Vorzeichen; Lösung: einfache Rotation
  - rechter Teilbaum zu tief: einfache Rotation des gesamten (Teil-) Baums gegen den Uhrzeigersinn; Fälle: a1, b1, b2
  - linker Teilbaum zu tief: einfache Rotation des gesamten (Teil-) Baums im Uhrzeigersinn; entsprechende Symmetriefälle: a1*, b1*, b2*
  - Beachte: Bei der Betrachtung der Symmetriefälle („Spiegelung der Fälle an Senkrechte durch Wurzel des Baums“, d. h. „linker Teilbaum zu tief“) wird die Rotationsrichtung jeweils umgekehrt.
- Balancefaktoren der Wurzel des aus der Balance geratenen (Teil-) Baums und deren Kind y auf dem Pfad in den zu tiefen Teilbaum haben verschiedene Vorzeichen; Lösung: Doppelrotation
  - rechter Teilbaum zu tief: zunächst nur Teilbaum mit Wurzel y im Uhrzeigersinn rotieren; danach gesamten (Teil-) Baum gegen den Uhrzeigersinn rotieren; Fälle: a2.1, a2.2, a2.3, b3.1, b3.2, b3.3
  - linker Baum zu tief:
  - zunächst nur Teilbaum mit Wurzel y gegen den Uhrzeigersinn rotieren; danach gesamten (Teil-) Baum im Uhrzeigersinn rotieren; Symmetriefälle: a2.1*, a2.2*, a2.3*, b3.1*, b3.2*, b3.3*
  - Aufwand von Rotation und Doppelrotation: Es sind jeweils eine konstante Anzahl von Referenzen zu ändern, Aufwand daher jeweils: $O(1)$.

Weitere Eigenschaften von AVL-Bäumen

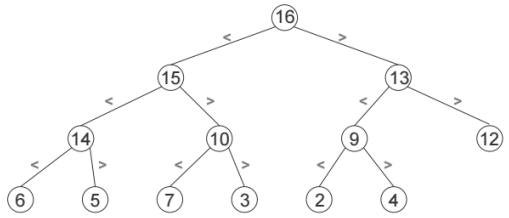
- Zusammenhänge für die Höhe h eines AVL-Baums mit nn Knoten:
  - $\log2(n+1) \leq h < 1{,}441 \log2(n+2)$ Im Vergleich zum vollständig ausgewogenen Binärbaum (mit minimaler Höhe) ist der AVL-Baum also höchstens ~44% höher. Alle Operationen bleiben daher vom Aufwand $O(\log2n)$.
- Nachteile von AVL-Bäumen:
  - Zusätzlicher Platzbedarf in den Knoten zur Speicherung der Höhe oder der Balancefaktoren (der andere Wert kann leicht berechnet werden, weil die Änderungsoperation vom Blatt in Richtung Wurzel läuft).
  - Komplizierte Implementierung.

Halde/Haufen (engl. Heap)

- Ein partiell geordneter Baum ist ein knotenmarkierter Binärbaum T, in dem für jeden Teilbaum T' mit Wurzel x gilt:
  - Variante 1: in der Wurzel steht immer das Minimum des Teilbaums $\forall y \in T' : \mu(x) \leq \mu(y)$
  - Variante 2: in der Wurzel steht immer das Maximum des Teilbaums $\forall y \in T' : \mu(x) \geq \mu(y)$
- Andere Bezeichnungen: Halde/Haufen (engl. Heap)
- Resultierende Halden-/Haufen-/Heap-Eigenschaft: Wert eines Knotens ist bezüglich einer gegebenen Ordnung kleiner oder gleich (Variante 1) bzw. größer oder gleich (Variante 2) dem Wert seiner zwei Nachfolger

Beispiel für max-Halde

Jede Wurzel eines Teilbaums hat einen größeren Wert, als alle übrigen Knoten dieses Teilbaums.



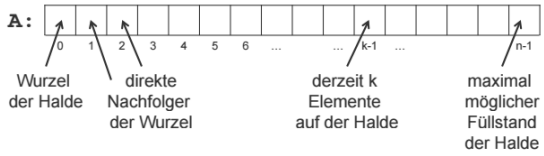
Wir betrachten im Folgenden links-vollständige partiell geordnete Bäume: alle Ebenen bis auf die letzte sind voll besetzt und auf letzter Ebene sitzen Knoten soweit links wie möglich.

Entnahme aus max-Halde

```
Algorithmus deleteMax(h) { // Laufzeit O(logk)
        // loesche das maximale Element aus der Halde h
        // und gib es aus
       entnimm der Wurzel ihren Eintrag und gib ihn als
              Maximum aus;
        nimm den Eintrag der letzten besetzten Position im
              Baum (loesche diesen Knoten) und setze ihn in
               die Wurzel;
        sei p die Wurzel und seien q, r ihre Kinder;
        solange q oder r existieren und
              (μ(p) < μ(q) oder μ(p) < μ(r)) fuehre aus: {
              vertausche den Eintrag in p mit dem groesseren
                      Eintrag der beiden Kinder;
              setze p auf den Knoten, mit dem vertauscht
                     wurde, und q, r auf dessen Kinder;
        }
}
```

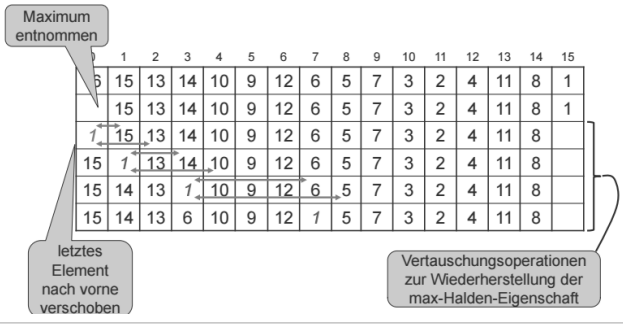
Implementierung einer Halde durch Einbettung in Array

Zur Speicherung einer Halde wird oft die BinärbaumImplementierung „Einbettung in ein Array“ verwendet.



Implementierung: Entnahme aus max-Halde (Array-Einbettung)

Schrittweise Veränderung der in das Array A eingebetteten max-Halde:



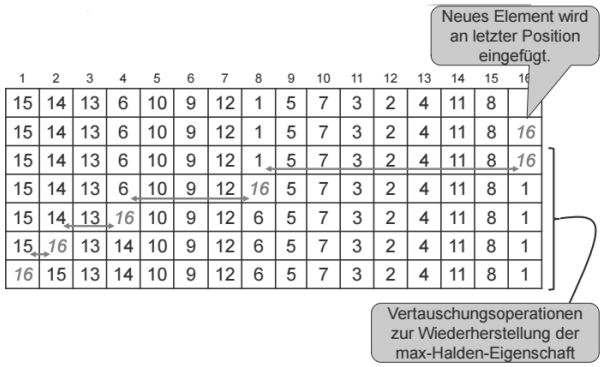
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	15	13	14	10	9	12	6	5	7	3	2	4	11	8	1
	15	13	14	10	9	12	6	5	7	3	2	4	11	8	1
1	15	13	14	10	9	12	6	5	7	3	2	4	11	8	
15	*1*	13	14	10	9	12	6	5	7	3	2	4	11	8	
15	14	13	*1*	10	9	12	6	5	7	3	2	4	11	8	
15	14	13	6	10	9	12	*1*	5	7	3	2	4	11	8	

Einfügen in max-Halde

```
Algorithmus insertMax(h,e) {
        // fuege Element e in den Heap h ein
        erzeuge neuen Knoten q mit Eintrag e;
        fuege q auf der ersten freien Position der
              untersten Ebene ein (falls unterste Ebene voll
              besetzt ist, beginne neue Ebene);
        sei p der Elternknoten von q;
        solange p existiert und μ(q) > μ(p), fuehre aus: {
              vertausche die Eintraege in p und q;
              setze q auf p und p auf den Elternknoten von p;
        }
}
```

Implementierung: Einfügen in max-Halde (Array-Einbettung)

Schrittweise Veränderung der in das Array A eingebetteten max-Halde



Anwendung für Halde

- effiziente Implementierung einer Prioritätswarteschlange
- Realisierung mit Halde (k Elemente)
  - Entnahme eines Elements: $O(\log k)$
  - Einfügen eines Elements: $O(\log k)$
  - Begründung:
    - beide Operationen folgen einem Pfad im Baum
    - Baum ist balanciert
    - Realisierung mittels Array-Einbettung deshalb in $O(\log k)$
- Vergleich dazu Realisierung mit Liste (k Elemente)
  - Entnahme eines Elements: $O(1)$ (zu entnehmendes Element steht vorne)
  - Einfügen eines Elements: $O(k)$

Tafelübung

Baumtraversierung

- Traversierung eines Baumes:
  - alle Knoten des Baumes ausgehend von der Wurzel „besuchen“
- man unterscheidet verschiedene Besuchsreihenfolgen
  - Tiefensuche (DFS, depth-first search):
    - Kinder vor „Geschwistern“ besuchen ; noch zu besuchende Knoten auf einen Stack ablegen
  - Breitensuche (BFS, breadth-first search):
    - „Geschwister“ vor Kindern besuchen noch zu besuchende Knoten in eine Queue ablegen
- Tiefen- und Breitensuche auch auf beliebigen Graphen möglich
  - dann aber notwendig: Markierung der bereits besuchten Knoten

Tiefensuche mit explizitem Stack

```
funktion dfs ( G : Graph ) {
       S := leerer Stack ;
       fuege Wurzel von G in S ein ;
       solange S nicht leer ist fuehre aus {
              a := entferne oberstes Element von S ;
              bearbeite Knoten a ;
              fuer alle Nachfolger n von a fuehre aus {
                     lege n oben auf S ;
              }
       }
}
```

Tiefensuche unter Verwendung des Aufruf-Stacks

```
funktion dfs ( G : Graph ) {
        dfs_helper ( Wurzel von G );
}
funktion dfs_helper ( k : Knoten ) {
        bearbeite k ;
        fuer alle Nachfolger n von k fuehre aus {
                dfs_helper ( n );
        }
}
```

Breitensuche mit Queue

```
funktion bfs ( G : Graph ) {
        Q := leere Queue ;
        fuege Wurzel von G in Q ein ;
        solange Q nicht leer ist fuehre aus {
                a := entferne vorderstes Element aus Q ;
                bearbeite Knoten a ;
                fuer alle Nachfolger n von a fuehre aus {
                        fuege n hinten in Q ein ;
                }
        }
}
```

Sortieralgorithmen

Sortieren

- Sortieren einer (Mehrfach-) Menge von Elementen über einem geordneten Wertebereich (z. B. int, double, String) ist zentrales und intensiv studiertes algorithmisches Problem.
- Mehrfachmenge: mehrfaches Vorkommen der Elemente erlaubt
- Ziel: Berechnung einer geordneten Sequenz aus einer ungeordneten Sequenz dieser Elemente.
- Gegeben:
  - Grundmenge U (U: Universum), totale Ordnung ≤ hierauf
  - Mehrfachmenge M mit Elementen $e_i \in U$
  - Repräsentation von M als Sequenz der Elemente: $l_0 = [e_0, e_1, \dots, e_{n-1}]$ (implementiert z. B. als verkettete Liste oder als (dynamisches) Array).
- Gesucht:
  - $l_s = [e_{j0}, e_{j1}, \dots, e_{jn-1}]$: Anordnung der Elemente aus l_0 gemäß der Ordnung ≤ mit Nachbedingung $e_{j0} \le e_{j1} \le \dots \le e_{jn-1} \wedge \mathrm{perm}(l_0, l_s)$; Prädikat perm ist erfüllt, wenn Sequenz l_s eine Permutation von l_0 ist.

Klassifizierung von Sortierverfahren

- intern / extern:
  - internes Sortierverfahren:
    - Alle Datensätze können gleichzeitig im Hauptspeicher gehalten werden.
    - Direkter Zugriff auf alle Elemente ist möglich und erforderlich.
  - externes Sortierverfahren:
    - Sortieren von Massendaten, die auf externen Speichermedien gehalten werden.
    - Zugriff ist auf einen Ausschnitt der Datenelemente beschränkt.
- methodisch:
  - Sortieren durch Auswählen
  - Sortieren durch Einfügen
  - Sortieren mittels Teile-und-Herrsche/Divide-and-Conquer-Verfahren
  - Sortieren durch Fachverteilen
- nach Effizienz:
  - Einfache Verfahren haben Laufzeit $O(n^2)$.
  - Effiziente Verfahren haben Laufzeit $O(n \log n)$.
  - Es ist beweisbar, das vergleichsbasierte Sortierverfahren nicht besser sein können → untere Schranke des Sortierproblems.
  - Manche Methoden: Unterschiede in Durchschnitts- und Worst-CaseVerhalten
- im Array oder nicht:
  - Array-basierte Verfahren benötigen keinen zusätzlichen Platz für Referenzen.
  - Besonders interessant: Verfahren, die nur ein einziges Array benötigen.
    - Sprechweise: solche Verfahren sortieren in situ (auch: in place).
    - Ergebnis wird erzielt durch Vertauschungen innerhalb dieses Arrays.

Kosten-Nutzen-Analyse für Sortierung

- Wesentliche Ausgangsfrage vor einem Sortiervorhaben: Lohnt sich der Aufwand für die Sortierung überhaupt?
- Strategie: Kosten-Nutzen-Analyse durchführen
  - Sei anz_{sv} die Zahl der Suchvorgänge über die Lebensdauer der Menge.
  - Dann ist es sinnvoll die Menge zu sortieren, falls gilt:
  - $T_s + anz_{sv} \cdot T' < anz_{sv} \cdot T$
    - mit T_s: Aufwand für das Sortieren der Menge
    - T': Aufwand für das Suchen in sortierter Menge
    - T: Aufwand für das Suchen in unsortierter Menge

Sortierverfahren für Listen bzw. für Arrays

- Sortierverfahren für Listen:
  - Verfahren, die Elemente der Liste entlang ihrer Anordnung aufgreifen möchten.
- Sortierverfahren für Arrays:
  - Verfahren, die ausnutzen, dass ein Zugriff auf ein Element des Arrays in $O(1)$ erfolgen kann.
  - Verfahren zur Sortierung von Listen sind prinzipiell auch anwendbar.
  - Im Allg. verbrauchen diese jedoch mehr Speicherplatz, weil das Ergebnis in eine neue Liste geschrieben wird und erst dann die alten Listen frei gegeben werden.
  - In dieser LE beschriebene Array-Implementierungen arbeiten ohne Kopie auf ein und demselben Array (Sortierung erfolgt in situ)

Überblick über die Sortierverfahren

vergleichsbasierte Sortierverfahren

Bezeichnung	Best Case	Average Case	Worst Case	stabil	in situ
Einfache Sortierverfahren					
Sortieren durch Auswählen (SelectionSort)	$O(n^2)$	$O(n^2)$	$O(n^2)$	(leicht mgl.)	(wenn mit Array)
Sortieren durch Einfügen (InsertionSort)	O(n)	$O(n^2)$	$O(n^2)$	X	(wenn mit Array)
Blasensortierung (BubbleSort)	O(n)	$O(n^2)$	$O(n^2)$	X	X
Verfeinertes Auswählen					
Haldensortieren (HeapSort)	O(n log n)	O(n log n)	**O(n log n)**		X
Teile-&-Herrsche/Divide-&-Conquer-Verf.					
Sortieren durch Verschmelzen (MergeSort)	O(n log n)	O(n log n)	**O(n log n)**	X	(in der Regel nicht)
Sortieren durch Zerlegen (QuickSort)	O(n log n)	O(n log n)	$O(n^2)$		X

nicht-vergleichsbasierte Sortierverfahren

Sortieren durch Fachverteilen	Zeit	stabil	in situ
BucketSort	O(n + m)	X	
RadixSort	O(n · k)	X	

Sortieren durch Auswählen (SelectionSort)

Grundidee (Varianten)

- Lösche nacheinander die Maxima aus einer Liste l und füge sie vorne an eine anfangs leere Ergebnisliste ls an.. → aufsteigende Sortierung (im Folgenden betrachtet)
- Lösche nacheinander die Minima aus einer Liste l und füge sie vorne an eine anfangs leere Ergebnisliste ls an. → absteigende Sortierung
- Lösche nacheinander die Maxima aus einer Liste l und füge sie hinten an eine anfangs leere Ergebnisliste ls an. → absteigende Sortierung
- Lösche nacheinander die Minima aus einer Liste l und füge sie hinten an eine anfangs leere Ergebnisliste ls an. → aufsteigende Sortierung

SelectionSort auf verketteten Listen: Entwurf rekursiver Lösung

- Algorithmenentwurf durch Induktion:
  - Induktionsanfang:
    - Leere Liste ls und einelementige Liste ls sind sortiert.
  - Induktionsschritt:
    - Sei $(n-1)$-elementige Liste ls sortiert.
    - Nächstes Element aus l ist < als alle Elemente in ls und wird als erstes Element in ls eingefügt,
    - Daher ist n-elementige Liste ls sortiert.
  - Induktionsende:
    - Wenn l leer ist, dann befinden sich alle Elemente in ls.
- Formulierung als rekursive Methode der Klasse LinkedList<E>
- Wir wollen nicht mit jedem Rekursionsschritt eine neue (Teil-) Liste erstellen, daher benötigen wir eine äußere Methode zur Einbettung der rekursiven Methode, in der insbesondere die Liste ls vereinbart wird.

SelectionSort auf verketteten Listen: rekursive Implementierung

Äußere Methode selsort

```
public LinkedList<E> selsort() {
 // Kopie der zu sortierenden Liste l0
 LinkedList<E> l = new LinkedList<E>();
 l.addAll(this);
 // Erzeuge leere Ergebnisliste ls
 LinkedList<E> ls = new LinkedList<E>();
 // {P: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert}
 // Aufruf der Rekursion
 return selsortRec(l, ls);
 // {Q: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert ∧ l leer}
}
```

SelectionSort auf verketteten Listen: rekursive Implementierung

Innere Methode selsortRec

```
LinkedList<E> selsortRec(LinkedList<E> l, LinkedList<E> ls) {
        // {I: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0 ∧ ls sortiert}
        if (!l.isEmpty()) {
                E max = getMaximum(l);
                l.remove(max);
                ls.addFirst(max);
                return selsortRec(l, ls);
        } else return ls;
}
```

SelectionSort auf verketteten Listen: iterative Implementierung

Nach Entrekursivierung und Einbettung in die Außenmethode:

```
public LinkedList<E> selsort() { //O(n²)
        // Kopie der zu sortierenden Liste l0
        LinkedList<E> l = new LinkedList<E>();
        l.addAll(this);
        // Erzeuge leere Ergebnisliste ls
        LinkedList<E> ls = new LinkedList<E>();
        // {P: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
        // ∧ ls sortiert}
        while (!l.isEmpty()) {
                E max = getMaximum(l); // Maximum auswaehlen
                l.remove(max); // Maximum entfernen
                ls.addFirst(max); // Maximum an Ergebnisliste
                // vorne anfuegen
                // {I: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
                ∧ ls sortiert}
        }
        // {Q: ∀ e ∈ l, e' ∈ ls: e ≤ e' ∧ l ∪ ls = l0
        // ∧ ls sortiert
        // ∧ l leer}
        return ls;
}
```

Korrektheit und Stabilität von selsort

- selsort ist korrekt
  - $Q = I \wedge \neg b = I \wedge l$ leer
  - I gilt für die leere Liste ls
  - Bei Verlassen der Schleife gilt I mit leerer Liste l: $ls = l0 \wedge ls$ sortiert (”=” steht für Gleichheit von Mehrfachmengen)
  - I ist Schleifeninvariante: Wiederherstellen von I durch remove(…) und add(…)
  - Terminierung, da l streng monoton verkürzt wird.
- Damit selsort stabil ist, muss getMaximum(l) das in der Reihenfolge von l letzte Maximum finden und remove(…) auch dieses Element löschen.

Aufwandsabschätzung zu selsort

- Idee für Verbesserung (führt zur Haldensortierung (HeapSort)):
  - Die kritischen Operationen sind das Identifizieren und Entfernen des Maximums.
  - In einer Halde war es möglich, das Maximum in $O(1)$ zu entnehmen und in $O(\log n)$ die max-Halden-Eigenschaft wieder herzustellen

In situ SelectionSort auf Arrays: iterative Implementierung

```
public void selsort() {  //O(n2)
        E max;
        int maxIndex;
        for (int i = a.length - 1; i > 0; i--) {
              max = a[i];
              maxIndex = i;
              for (int j = 0; j < i; j++) {
                    if (a[j].compareTo(max) >= 0) {
                          max = a[j];
                          maxIndex = j;
                    }
              }
              swap(a, i, maxIndex);
        }
}
protected void swap(E[] a, int n, int m) {
        E tmp = a[n];
        a[n] = a[m];
        a[m] = tmp;
}
```

Sortieren durch Einfügen (InsertionSort)

- Grundidee
- Typisches Verfahren beim Sortieren von Spielkarten:
  - Starte mit der ersten Karte vom Stapel eine neue Kartensequenz auf der Hand.
  - Nimm jeweils die nächste Karte vom Kartenstapel und füge diese an der richtigen Stelle in die Kartensequenz auf der Hand ein.
- Angenommen wir können n – 1 Werte sortieren. Dann können wir den n-ten Wert einsortieren, indem wir seinen Platz in der sortierten Liste finden und die restlichen Elemente nach hinten verschieben.
- Nimm das nächste Element aus der Liste l und füge es an der richtigen Stelle in die (anfangs leere) Liste ls ein.

InsertionSort auf Listen: iterative Implementierung

```
public LinkedList<E> insort() {
        LinkedList<E> l = new LinkedList<E>();
        l.addAll(this);
        LinkedList<E> ls = new LinkedList<E>();
        // {P: l ∪ ls = l0 ∧ ls sortiert ∧ ls leer}
        while (!l.isEmpty()) {
              E e = l.get(0);
              l.remove(e);
              // suche in ls e', e" aufeinanderfolgend
              // mit e' ≤ e < e";
              // fuege in ls e zwischen e', e" ein
        }
        // {Q: l ∪ ls = l0 ∧ ls sortiert ∧ l leer}
        return ls;
}
```

Bei alternativer Implementierung auf Basis von Arrays entsteht analog zu SelectionSort wieder ein in situ-Verfahren

InsertionSort auf Listen: Aufwandsabschätzung

- Sequentielle Suche:
- mittlerer Aufwand $O(k/2)$
- schlimmster Fall $O(k)$ wenn k Länge von ls ist
- Günstigster Fall:
  - l ist bereits entgegengesetzt geordnet (jew. Entnahme des Maximums).
  - Innere Schleife verursacht dann konstanten Aufwand c', weil immer vorne angefügt werden kann.
  - Gesamtaufwand ist dann im besten Fall in $O(n)$.

InsertionSort auf Arrays: iterative Implementierung

```
public void insort() {
        E e;
        int j;
        for (int i = 1; i < a.length; i++) {
                e = a[i]; // entnommenes Element merken
                j = i - 1;
                while (j >= 0 && e.compareTo(a[j]) < 0) {
                        a[j + 1] = a[j];
                        j--;
                }
                a[j + 1] = e; // entnommenes Element
                              // einsetzen
        }
}
```

- i: vordere Grenze des Arraybereichs, aus dem erstes Element entnommen wird.
- j: Durchlaufen des vorderen Arraybereiches, um Einfügestelle zu finden; bis dahin: Elemente von oben kommend um eine Position nach rechts verschieben.

Blasensortierung (BubbleSort)

- Grundidee
  - Das Array wird immer wieder durchlaufen und dabei werden benachbarte Elemente in die richtige Reihenfolge gebracht.
  - Größere Elemente überholen so die kleineren und drängen an das Ende der Folge („wie aufsteigende (Luft-) Blasen“)

BubbleSort auf Arrays

```
static <E extends Comparable<E>> void bubblesort(E[] a) { // O(n^2)
        // Variable zum merken, ob beim aktuellen Durchlauf der Sequenz eine
        // Vertauschung stattfand
        boolean swapped;
        // oberster Index des Arrays a, bis zu dem noch korrekte Ordnung geprueft
        // werden muss
        int upper = a.length - 1;
        do {
                swapped = false;
                for (int i = 0; i < upper; i++) {
                        if (a[i].compareTo(a[i + 1]) > 0) {
                                // tausche im Array a die
                                // Eintraege an den Indizes
                                // i und i + 1
                                swap(a, i, i + 1);
                                // merke: es wurde getauscht
                                swapped = true;
                        }
                }
                upper--;
        } while (swapped);
}
```

BubbleSort auf Arrays: Aufwandsabschätzung

- Im ersten Durchlauf werden n Elementpaare verglichen.
- Im zweiten Durchlauf werden n – 1 Elementpaare verglichen (upper ist um 1 reduziert).
- Gesamtaufwand: $O(n^2)$
- Durch die Optimierung mit swapped sinkt der Aufwand im bestenFall auf $O(n)$.

Haldensortierung (HeapSort)

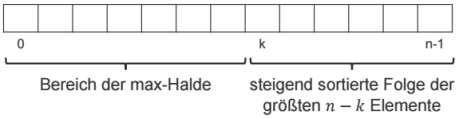
- Sortieren durch Auswählen bisher: SelectionSort
  - Je Schritt das Maximum (Minimum) der Werte auswählen und aus Ursprungsliste entnehmen: Aufwand $O(n)$.
  - Gesamtaufwand: $O(n^2)$
- Strategie:
  - Maximumsauswahl (Minimumsauswahl) durch geeignete Datenstruktur verbessern: Halde (Heap).
  - Auswahl des Maximums (Minimums) und Entfernung aus einer Halde hat (nur) den Aufwand $O(\log n)$.
- Grundidee der Haldensortierung (HeapSort):
  - Die n zu sortierenden Elemente werden in eine max-Halde (minHalde) eingefügt: Aufwand $O(n \log n)$. Optimiert sogar nur $O(n)$.
  - Es wird n-mal das Maximum (Minimum) aus der Halde entnommen: Aufwand $O(n \log n)$.
  - Gesamtaufwand damit $O(n \log n)$

Haldensortierung (HeapSort)

- Ziel: Aufbau der Halde optimieren und in situ sortieren.
- Teil-Array $a[i..kk]$, $0 \le i \le k < n$, heißt Teil-max-Halde (Teil-maxHeap), gdw.: $\forall j \in [i, \dots, k]\ a[j] \ge a[2j+1]$ falls $2j+1 \le k$
- und $a[j] \ge a[2j+2]$ falls $2j+2 \le k$
- Wenn $a[0..n-1]$ eine Teil-max-Halde ist, dann ist $a[0..n-1]$ auch eine max-Halde, d. h. die Array-Einbettung eines partiell geordneten Baums.
- Ablauf Haldensortierung (HeapSort):
  - Aufbau der max-Halde: Die hintere Hälfte des Arrays $a[n/2..n-1]$ ist bereits eine Teil-max-Halde (da diese Knoten keine Kinder mehr haben und somit die Bedingung trivialerweise erfüllt ist).

```
for i := n/2 - 1 downto 0 do {
        erweitere die Teil-max-Halde a[i+1..n] zu einer
        Teil-max-Halde a[i..n] durch „Einsinken lassen“ von a[i]
}
```

  - Baue die sortierte Folge rückwärts am hinteren Ende des Arrays auf



    - In jedem Schritt wird $a[0]$ mit $a[k-1]$ vertauscht und damit der Haldenbereich auf $a[0..k-2]$ reduziert.
    - $a[1..k-2]$ ist weiterhin eine Teil-max-Halde.
    - Durch Einsinken von $a[0]$ wird $a[0..k-2]$ wieder zu einer (Teil-max-) Halde

reheap: Element in max-Halde „einsinken“ lassen

Ziel: Wiederherstellen der max-Halden-Eigenschaft, indem a[i] in die Halde „einsinkt“.

```
protected void reheap(E[] a, int i, int k) {
        int leftKidIdx = 2 * i + 1;
        int rightKidIdx = leftKidIdx + 1;
        int kidIdx;
        if (leftKidIdx <= k && rightKidIdx > k) {
                // nur ein (= linkes) Kind
                if (a[leftKidIdx].compareTo(a[i]) > 0) {
                        swap(a, leftKidIdx, i);
                }
        } else {
                if (rightKidIdx <= k) {
                        // in kidIdx groesseren der beiden Kinder erfassen
                        kidIdx = a[leftKidIdx].compareTo(a[rightKidIdx]) > 0
                                        ? leftKidIdx : rightKidIdx;
                        if (a[kidIdx].compareTo(a[i]) > 0) { // gfs. tauschen
                                swap(a, i, kidIdx);
                                reheap(a, kidIdx, k);
                        }
                }
        }
}
```

Aufwand für Aufbau der Halde

- Grobe Abschätzung:
  - Jeder Aufruf von reheap hat Aufwand $O(\log n)$.
  - $O(n)$ Aufrufe von reheap haben den Gesamtaufwand $O(n \log n)$.
- Genauere Abschätzung:
  - Laufzeit für reheap ist abhängig von der Höhe h der Halde.
  - Diese ist aber meist viel kleiner als $\log n$.
  - Z.B. haben die ersten $(n/2)$ eingefügten Werte die Höhe 0.
  - Allgemein: es gibt $(n)/(2^{h+1})$ Knoten der Höhe h. Für jeden von diesen hat reheap den Aufwand $O(h)$, fast immer also weniger als $O(n \log n)$.
  - Es lässt sich damit für den Gesamtaufwand zeigen: $\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O(n)$

Implementierung von HeapSort

```
public void heapsort(E[] a) {
        int n = a.length;
        // Phase 1: Halde aufbauen
        for (int i = n / 2; i >= 0; i--) {
              reheap(a, i, n - 1);
        }

        // Phase 2: jeweils Maximum entnehmen und sortierte Liste am Ende aufbauen
        for (int i = n - 1; i > 0; i--) {
               // Maximum ans Ende des Haldenbereichs tauschen
               swap(a, 0, i);
               // nach vorne getauschtes Element einsinken lassen und
               // max-Halden-Eigenschaft wieder herstellen
               reheap(a, 0, i - 1);
        }
}
```

Sortieren unter Verwendung der Teile-und-Herrsche-Strategie

- Zur Erinnerung: Teile-und-Herrsche/Divide-and-Conquer
- Wenn die Objektmenge klein genug ist, dann löse das Problem direkt sonst
  - Divide: Zerlege die Menge in mehrere Teilmengen (möglichst ähnlicher/gleicher Größe).
  - Conquer: Löse das Problem rekursiv für jede Teilmenge.
  - Merge: Berechne aus den für die Teilmengen erhaltenen Lösungen eine Lösung des Gesamtproblems.
- im Folgenden betrachtete Strategien:
  - Sortieren durch Verschmelzen (MergeSort)
  - Sortieren durch Zerlegen (QuickSort)

Sortieren durch Verschmelzen (MergeSort)

- Grundidee
- gegeben sei eine Sequenz l = [e0, e1, … , en−1].
- Algorithmus MergeSort(l):
  - Wenn Länge(l) <= 1, dann gib l zurück, sonst
  - Divide:
    - l1 := [e0,…,e⌊n/2⌋-1];
    - l2 := [e⌊n/2⌋ ,…,en-1];
  - Conquer:
    - l1' := MergeSort(l1);
    - l2' := MergeSort(l2);
  - Merge:
  - gib merge(l1',l2') zurück
- Sortierung erfolgt beim Verschmelzen (merge)
- Rekursionstiefe ist in O(log2 n).
- Verfahren arbeitet (auch) auf Listen.

Aufwandsabschätzung zu MergeSort

- Gegeben: Liste der Länge n
- Rekursive Aufrufe lassen sich als Baum darstellen, dessen Knoten Aufrufe von mergesortRec repräsentieren.
- Wir benennen die Ebenen von oben kommend mit k = 0, 1, …
- Gesamtaufwand ist Summe der Aufrufe aller Knoten im Rekursionsbaum.
- Aufwand eines Knotens auf Ebene k ist jeweils in $O(n/2^k)$ für das Zerlegen und Verschmelzen der sortierten Hälften (Länge ca. $n/2^k$).
- Anzahl der Knoten/Listen auf Ebene k ist 2^k, also in $O(2^k)$.
- Summe der Aufwände auf Ebene k ist demnach $O(2^k)\cdot O(n/2^k) = O(n)$, unabhängig von k.
- Anzahl der Ebenen ist in $O(\log_2 n)$.
- Gesamtaufwand im schlechtesten Fall: $O(n\cdot\log_2 n)$

MergeSort auf Listen: rekursive Implementierung

äußere Methode mergesort

```
static <E extends Comparable<? super E>> LinkedList<E>
                        mergesort(LinkedList<E> l) {
        // Basisfall
        if (l.size() < 2) return l;
        // zerlege l in 2 ungefaehr gleich lange Teillisten
        LinkedList<E> left = new LinkedList<>(l.subList(0, l.size() / 2));
        LinkedList<E> right = new LinkedList<>(l.subList(l.size() / 2, l.size()));
        // sortiere beide Teillisten und verschmelze sie
        return merge(mergesort(left), mergesort(right));
}
```

Verschmelzen (sog. Reißverschlussverfahren)

```
static <E extends Comparable<? super E>> LinkedList<E>
              merge(LinkedList<E> l, LinkedList<E> r) {
        LinkedList<E> ret = new LinkedList<>();
        while (l.size() > 0 && r.size() > 0) { // Reissverschluss
               E left = l.getFirst(), right = r.getFirst();
               if (left.compareTo(right) <= 0) {
                      ret.addLast(left);
                      l.removeFirst();
               } else {
                      ret.addLast(right);
                      r.removeFirst();
               }
        }
        while (l.size() > 0) { // uebrige Werte von links
               ret.addLast(l.getFirst());
               l.removeFirst();
        }
        while (r.size() > 0) { // uebrige Werte von rechts
               ret.addLast(r.getFirst());
               r.removeFirst();
        }
        return ret;
}
```

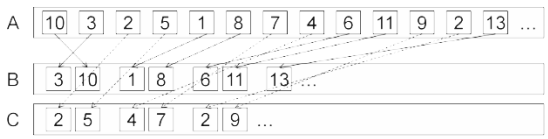
Wichtige Eigenschaften von MergeSort

- mergesort(…) ist stabil.
- Kein wahlfreier Zugriff notwendig.
- Schnellstes Verfahren auf verketteten Listen wegen der geringsten Zahl an Vergleichen.
- Sequenzieller Zugriff auf Teillisten, daher verbreitetes Sortierverfahren für große Listen, die nicht mehr in den Hauptspeicher passen (externes Sortierverfahren).
- Durch geschickte Listenimplementierung auch in-situ (auch ohne Kopieroperationen der einzelnen Elemente) möglich.

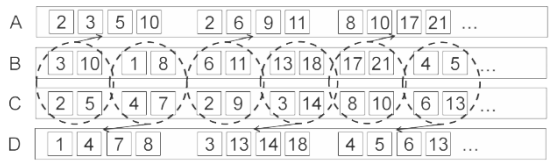
MergeSort als externes Sortierverfahren

Bottom-Up statt Top-Down.

- 4 „Bänder“ A-D (= sequentiell zugreifbare Dateien, Magnetbänder), Die Eingabe befindet sich auf Band A.
- 1. Schritt: Zweier-Tupel von Band A verschmelzen und abwechselnd auf Band B und Band C schreiben.
- Auf Band C und D liegen nun jeweils sortierte Zweier-Tup



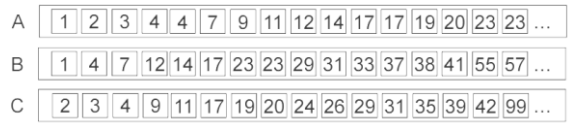
- 2. Schritt: Zweier-Tupel von Band B und C verschmelzen, die entstehenden Vierer-Tupel abwechselnd auf Band A und D speichern.



- 3. Schritt: Vierer-Tupel von Band A/D verschmelzen, Ergebnis abwechselnd auf Band B/C speichern.
- k. Schritt: Tupel der Länge 2^{k-1} verschmelzen, Ergebnis abwechselnd schreiben (dabei alterniert A/D und B/C als Eingabe).

MergeSort als externes Sortierverfahren

- Letzter Schritt: jeweils nur noch je ein Tupel mit ca. n/2 Elementen auf zwei Bändern (o.b.d.A. Band B/C, Ausgabe erfolgt auf A).

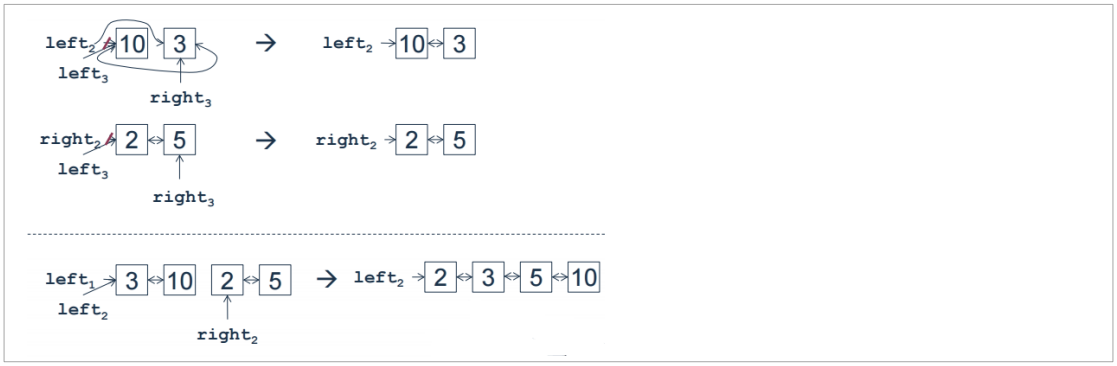


- Falls Band A vor dem letzten Schritt belegt ist, muss noch umkopiert werden.
- Insgesamt sind $\log_2 n$ Schritte notwendig. In jedem Schritt werden alle Elemente einmal bearbeitet Gesamtaufwand: $O(n \cdot \log_2 n)$.

MergeSort in-situ

- Mittels geschickter Listenoperationen ist es möglich, MergeSort in-situ zu implementieren.
- Zuerst rekursiv aufteilen (nur Referenzen "umbiegen" notwendig).

- Beim Verschmelzen wieder "Umbiegen" von Referenzen.



Sortieren durch Zerlegen (QuickSort)

Grundidee

- Gegeben sei eine Sequenz l = [$e0$, $e2$, … , en−1].
- Algorithmus QuickSort(l):
- Wenn Länge(l) <= 1, dann gib l zurück, sonst
  - Divide:
    - Wähle irgendeinen Wert p = ej aus l aus. Berechne eine Teilfolge l1 aus l mit den Elementen, deren Wert ≤ p ist, und eine Teilfolge l2 mit Elementen > p.
  - Conquer:
    - l1' := QuickSort(l1);
    - l2' := QuickSort(l2);
  - Merge: gib concat(l1',l2') zurück
- Sortierung erfolgt durch die Art der Zerlegung (divide).
- Verfahren arbeitet (vor allem) auf Arrays in situ.

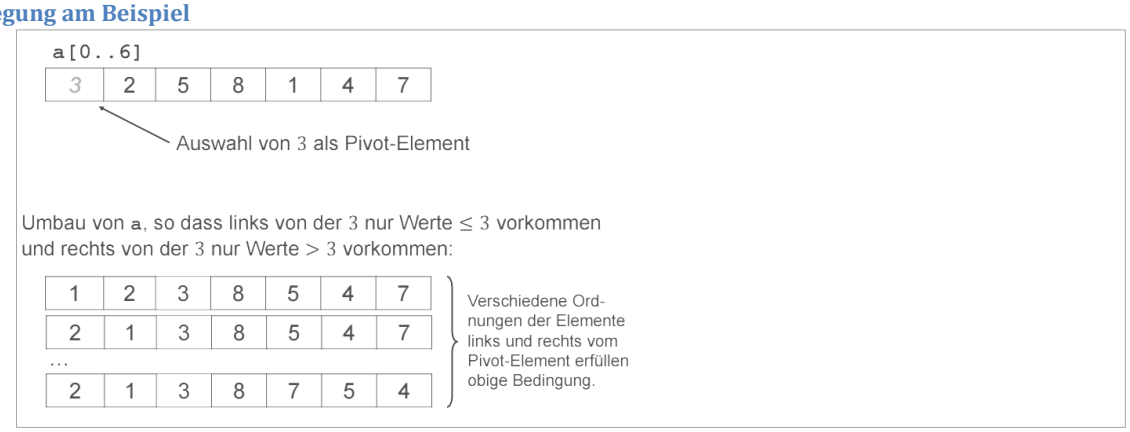
Zerlegung

- Gegeben:
  - Array a von Elementen mit Totalordnung ≤ und
  - Ausschnitt $a[m. . n]$ zwischen Indizes m, nn(einschließlich).
  - In diesem Ausschnitt vorkommendes beliebiges Element p, sogenanntes Pivot-Element (pivot (franz.): Dreh-/Angelpunkt).

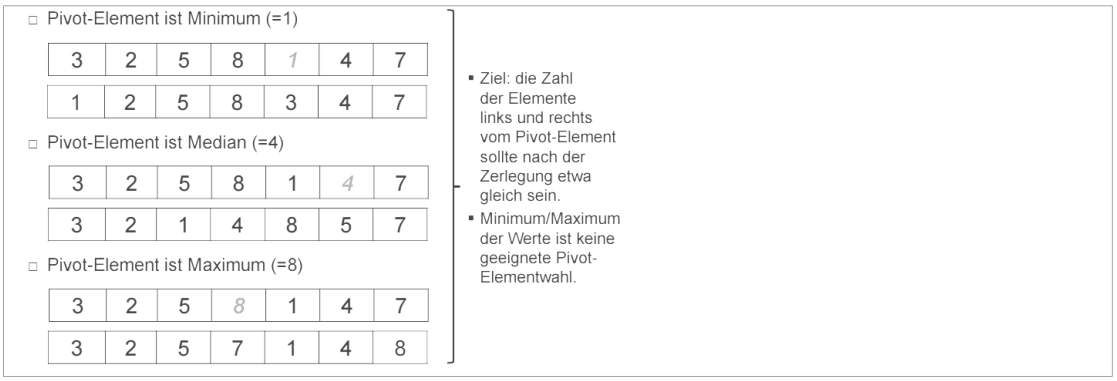
Zerlegung

- Aufgabe:
  - Umbau des Ausschnitts durch Platztauschen, so dass für einen Index r (r ist Index des Pivot-Elements; p: = $a[r]$ ist Pivot) mit $m \le r \le n$ gilt:
    - $a[k] \le p$ für $m \le k \le r$ und
    - $a[k] > p$ für $r < k \le n$.
  - Der Index r soll zurückgegeben werden, während der Umbau von $a[m..n]$ als Seiteneffekt erfolgt.

Zerlegung am Beispiel



Auswirkungen der Wahl des Pivot-Elements



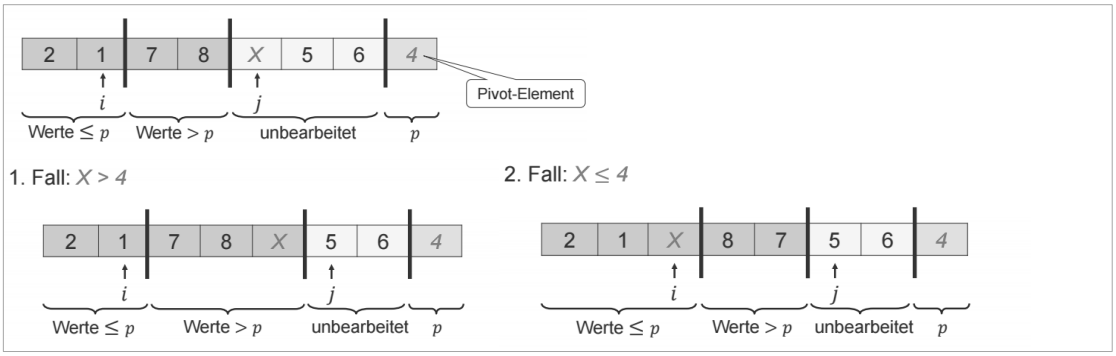
Spezifikation der Methode partition Vor-/Nachbedingung

- Methodensignatur:
  - private static <E extends Comparable<E>> int
  - partition(E[] a, int m, int n)
- Spezifikation der Methode:
  - Vorbedingung
    - P: $(0 \le m \le n < a.\,\mathrm{length})$
  - Nachbedingung
    - r: = Rückgabewert der Methode partition
    - a' : = „Belegung“ von a beim Betreten der Methode
    - Q: $(m \le r \le n) \wedge \mathrm{perm}(a[m..n], a'[m..n]) \wedge \forall k: ((m \le k \le r \Rightarrow a[k] \le a[r] \wedge (r < k \le n \Rightarrow a[r] < a[k]))$
  - Prädikat $\mathrm{perm}(x, y)$ ist wahr, falls die Sequenzen x und y Permutationen von einander sind.

Algorithmische Idee zum Finden der „Nahtstelle“

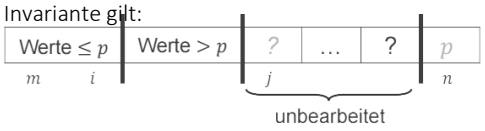
- Wähle das letzte Element des Arrays als Pivot-Element p.
- Zerlege den Array-Bereich logisch in 4 Bereiche: Sammelbereich für Werte ≤ p (anfangs leer), Sammelbereich für Werte > p (anfangs leer), Bereich noch zu bearbeitender Werte, Pivot-Element selbst.
- Grenzen zwischen den ersten drei Bereichen verschieben sich während des Verfahrens; Zeiger i und j zum Markieren der Grenzen.

Algorithmische Idee zum Finden der „Nahtstelle“



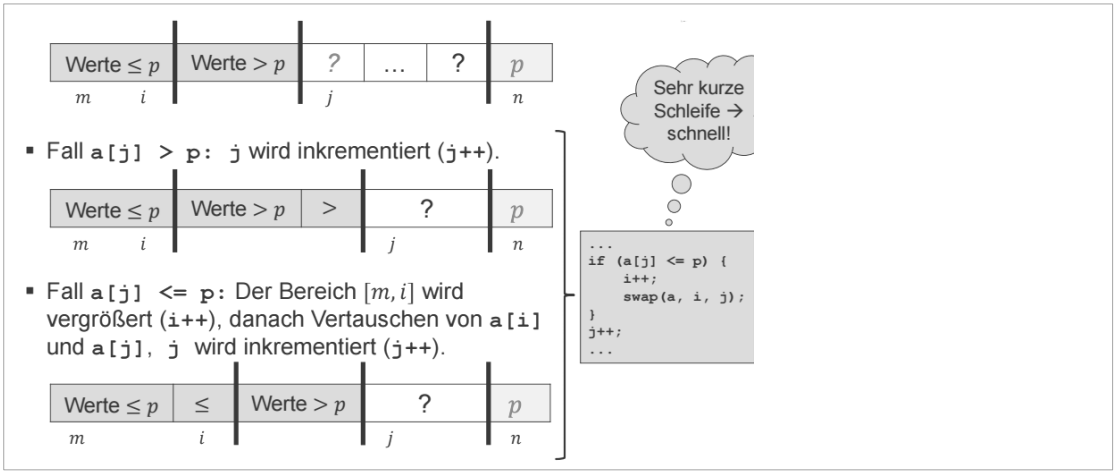
Spezifikation der Methode partition : Invariante

- Durch Wahl des Pivot-Elements am rechten Rand entstehen vier Bereiche $[m, i]$, $]i,j[$, $[j, n[$ und $[n, n]$, für deren Werte die folgende Invariante gilt:
  - grafisch:

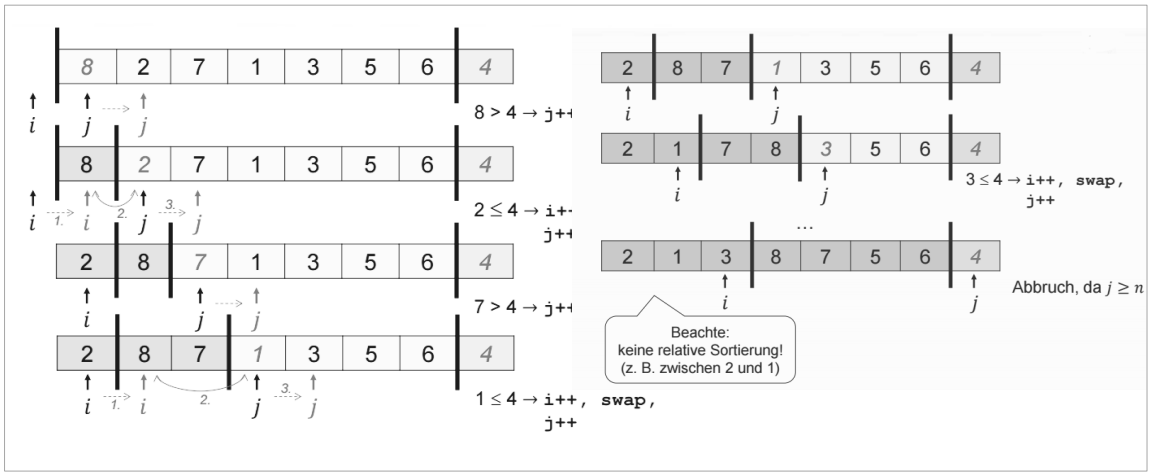


  - als Formel: I: $m - 1 \le i < j \le n \wedge \text{perm}(a[m..n]) \wedge \forall k$: $((m \le k \le i \Rightarrow a[k] \le p) \wedge (i < k < j \Rightarrow p < a[k]))$
  - Nachbedingung der Schleife: S: = $I \wedge \neg b$ S: $m - 1 \le i < j \le n \wedge \text{perm}(a[m..n], a'[m..n]) \wedge \forall k$: $((m \le k \le i \Rightarrow a[k] \le p) \wedge (i < k < j \Rightarrow p < a[k])) \wedge (j \ge n)$

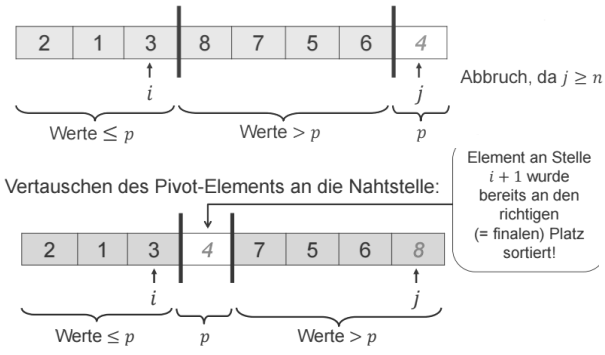
Verarbeitung eines Elements



Zerlegung am Beispiel



Zerlegung am Beispiel



Implementierung der Methode partition

```
protected int partition(E[] a, int m, int n) {
        // {P: (0 ≤ m ≤ n < a.length)}
        // a' := java.util.Arrays.copyOf(a, a.length); <- „virtual copy"
        E p = a[n];
        int i = m - 1;
        int j = m;
        while (j < n) {
                // {I: m - 1 ≤ i < j ≤ n ∧ perm(a[m..n],a'[m..n]) ∧
                // ∀ k: ((m ≤ k ≤ i ⇒ a[k] ≤ p) ∧ (i < k < j ⇒ p < a[k]))}
                if (a[j].compareTo(p) <= 0) {
                        i++;
                        swap(a, i, j);
                }
                j++;
        }
        // {S: I ∧ ¬b}
        int r = i + 1;
        swap(a, r, n);
        // {Q: (m ≤ r ≤ n) ∧ perm(a[m..n],a'[m..n]) ∧
        // ∀ k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧ (r < k ≤ n ⇒ a[r] < a[k]))}
        return r;
}
```

Korrektheit und Terminierung

- Korrektheit:
  - Die Bedingung {I: m - 1 ≤ i < j ≤ n ∧ perm(a[m..n],a'[m..n]) ∧ ∀ k: ((m ≤ k ≤ i ⇒ a[k] ≤ p) ∧ (i < k < j ⇒ p < a[k]))} ist Invariante. Beweis ist nicht schwierig.
- Terminierung:
  - Solange $j < n$ ist, wird der Abstand zwischen j und n pro Schleifendurchlauf um 1 verringert (Schleifenvariante z. B. V: = $n - j$).
- Nachbedingung der Methode partition:
  - Zu zeigen: $I \wedge \neg b \Rightarrow wp$("r = i + 1; swap(a, r, n);" ,Q) mit: Q: $(m \leq r \leq n) \wedge \mathrm{perm}(a[\mathrm{m}...\mathrm{n}]a'[m..n]) \wedge \forall k$: $((m \leq k \leq r \Rightarrow a[\mathrm{k}] \leq a[\mathrm{r}]) \wedge (r < k \leq n \Rightarrow a[\mathrm{r}] < a[\mathrm{k}]))$ Beweis ist nicht schwierig

QuickSort auf Arrays: rekursive Implementierung

äußere Methode quicksort

```
public void quicksort(E[] a) {
        quicksortRec(a, 0, a.length - 1);
}
```

innere Methode quicksortRec

```
protected void quicksortRec(E[] a, int m, int n) {
        if (n > m) {
                int r = partition(a, m, n);
                quicksortRec(a, m, r - 1);
                quicksortRec(a, r + 1, n);
        }
}
```

Nachbedingung von partition(…) Q

```
protected void quicksortRec(E[] a, int m, int n) {
        if (n > m) {
                int r = partition(a, m, n);
                // {Qpartition: (m ≤ r ≤ n) ∧ perm(a[m..n], a'[m..n]) ∧
                // ∀k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧
                // r < k ≤ n ⇒ a[r] < a[k]))}
                quicksortRec(a, m, r - 1);
                quicksortRec(a, r + 1, n);
        }
        // {Q: perm(a[m..n], a'[m..n]) ∧ a[m..n] sortiert}
}
```

- Für partition(…) lautete die Nachbedingung: {Qpartition: (m ≤ r ≤ n) ∧ perm(a[m..n], a'[m..n]) ∧ ∀k:((m ≤ k ≤ r ⇒ a[k] ≤ a[r]) ∧ r < k ≤ n ⇒ a[r] < a[k]))}
- Diese gilt nach dem Aufruf von partition(…).

Nachweis der Gültigkeit der Nachbedingung Q

- Induktionsanfang:
  - Rekursion so lange, bis partition(…) auf einelementiger Liste.
  - Diese ist sortiert.
- Induktionsschritt:
  - Sind beide Teile sortiert (Q), so ist wegen Qpartition auch die Verbindung der Teile sortiert

Aufwandsabschätzung zu QuickSort

- Bester Fall:
  - Wenn das Pivot-Element den zu sortierenden Ausschnitt genau halbiert (also die Hälfte der Werte kleiner ist als das Pivot-Element und die andere Hälfte der Werte größer ist als das Pivot-Element), dann hat QuickSort den Aufwand $O(n \log 2n)$.
- Schlechtester Fall:
  - Im schlechtesten Fall wird das Pivot-Element stets so gewählt, dass es das größte oder das kleinste Element der Liste ist.
  - Dies ist etwa der Fall, wenn als Pivot-Element stets das Element am Ende der Liste gewählt wird und die zu sortierende Liste bereits sortiert vorliegt.
  - Die zu untersuchende Liste wird dann in jedem Rekursionsschritt nur um eins kleiner und die Laufzeit wird beschrieben durch $n + (n-1) + (n-2) + \cdots + 1 \in O(n^2)$

Verbesserung der Wahl des Pivot-Elements

- Für jedes Verfahren, das nach einer feststehenden Regel das Pivot-Element aussucht (z. B. in der Mitte des Ausschnitts, am Ende des Ausschnitts, …), kann eine Eingabe gefunden werden, die zu quadratischem Aufwand führt.
- „Normalerweise“ tritt der $O(n^2)$-Fall aber nicht auf; dann hat QuickSort im Mittel den Aufwand $O(n \log 2n)$.
- Pivotwahl, um Wahrscheinlichkeit des $O(n^2)$-Falls zu reduzieren:
  - 3-Median-Strategie: Wähle drei Elemente vom linken und rechten Rand und aus der Mitte des zu sortierenden Ausschnitts. Wähle als Pivot-Element das mittlere dieser drei Elemente.
  - Zufallsstrategie: Wähle mit Zufallszahlengenerator (gleichverteilt) die Stelle des Pivot-Elements im zu sortierenden Ausschnitt aus.

Vorteile von QuickSort trotz Worst-Case-Verhaltens

- MergeSort und HeapSort scheinen überlegen, weil sie immer O(n log2n) Aufwand haben.
- Aber (1): QuickSort ist in der Praxis bei nicht pathologischer Pivotwahl schneller als MergeSort und HeapSort, weil die wesentliche Schleife weniger Instruktionen hat.
- Aber (2):
  - Bei wenigen Elementen (10 - 20) ist Sortieren durch Einfügen schneller als QuickSort.
  - Ursache: Im O-Kalkül sind die konstanten Faktoren von QuickSort größer, so dass bei kleinem n der quadratische Algorithmus trotzdem schneller ist als der $n \log 2n$-Algorithmus.
  - Daher die Rekursion nicht bis zum leeren oder einelementigen Ausschnitt laufen lassen, sondern rechtzeitig umsteigen.Umstieg ist einfacher zu machen als bei Merge- und HeapSort.

Sortieren durch Fachverteilen (BucketSort)

- Wenn die zu sortierenden Werte bestimmten Einschränkungen unterliegen und auch andere Operationen als Vergleiche angewendet werden können, so ist es möglich schneller zu sortieren, z. B. in $O(n)$.

Sortieren durch Fachverteilen (BucketSort)

- Beispiel:
  - Sei l= [e0, e1, … , en−1] wobei ei ganze Zahlen zwischen 0 und n – 1 sind und l keine Duplikate enthält.
  - Benutze zwei n-elementige Arrays: eines für l und eines für die sortierte Folge ls
  - Erzeugung der sortierten Folge in ls in $O(n)$
- Seien die zu sortierende Werte nun ganze Zahlen zwischen 0 und m − 1 und seien Duplikate erlaubt.
- Nutzung einer Menge von Behältern B0, … , Bm−1:
  - Jeder Behälter lässt sich als Liste von Elementen implementieren.
  - Die Behälter werden über Array verwaltet.
  - Datenstruktur entspricht der des offenen Hashing

```
Algorithmus BucketSort(l):
        für i := 0 bis m – 1 führe aus:
                Bi := ∅;
        für i := 0 bis n – 1 führe aus:
                füge ei in Bei ein;
        für i := 0 bis m – 1 führe aus:
                schreibe die Elemente von Bi in die
                Ergebnisfolge;
```

- Beispiel: Poststellen
  - Pro Person wird ein Postfach angelegt, insgesamt m Fächer.
  - Jedes Element der eingehenden Post (Menge mit n Elementen) wird in das Fach des Adressaten gelegt.
  - Jeder der n Briefe ist anzufassen, seine Fachnummer ist zu bestimmen, dann ist er in dieses Fach zu legen.
  - Am Schluss müssen die m Fächer geleert werden.
- Eigenschaften
  - Einfügen eines Elements in einen Behälter ist in $O(1)$ möglich.
  - Die Laufzeit von BucketSort ist in $O(n + m)$:
  - Wenn $m = O(n)$, dann sogar in $O(n)$
  - Bei $m \gg n$ aber nicht praktikabel (es gibt dann viel mehr mögliche Werte, als in der zu sortierenden Menge tatsächlich vorkommen).
  - Verfahren ist stabil, wenn zur Implementierung der Behälter Schlangen verwendet werden.

Verallgemeinertes Sortieren durch Fachverteilen (RadixSort)

- Statt für jeden in Frage kommenden Wert ein Fach anzulegen, wird der Wert in (kürzere) Segmente aufgeteilt.
- Für den kleinen Wertebereich des Segments gibt es Fächer, z. B.
  - einzelne Ziffern der Postleitzahl (d = 10 Fächer)
  - Buchstaben eines Worts fester Länge (d = 26 Fächer)
- Die Segmente befinden sich in einer definierten Reihenfolge, z. B. Stellen der Postleitzahl: 9-1-0-5-8
- Ablauf
  - Die Elemente der Menge werden gemäß des betrachteten Segments per BucketSort sortiert.
  - Für die Sortierung nach den restlichen Segmenten wird BucketSort rekursiv angewendet.
  - Dabei: Segmentbetrachtung von rechts nach links
- Wichtig: Realisierung der Behälter als Schlangen.

Korrektheit von RadixSort

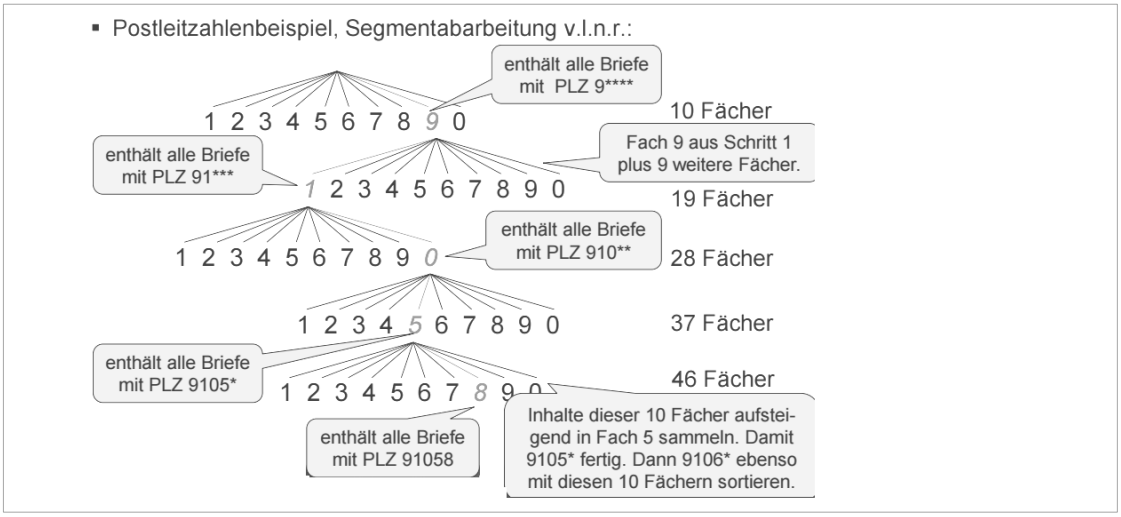
- Induktionsanfang: ein Segment/vorderstes Segment: Fächersortierung mit min. d Fächern erzeugt korrekte Sortierung.
- Induktionshypothese: Werte mit weniger als k Segmenten, die jeweils maximal d mögliche Werte einnehmen können, können sortiert werden.
- Induktionsschluss:
  - Fall: Zwei Elemente unterscheiden sich im k-ten Segment. Dann wird der k-te Sortierschritt nach Induktionsvoraussetzung zur korrekten Sortierung der Elemente führen.
  - Fall: Zwei Elemente sind im k-ten Segment gleich.
    - Dann sind sie nach Induktionsvoraussetzung in den k − 1 Segmenten rechts davon korrekt sortiert.
    - Ein stabiles Sortierverfahren behält die relative Ordnung der beiden Elemente bei.
    - Damit sind sie auch noch korrekt sortiert, wenn nach dem k-ten Segment sortiert worden ist

Aufwandsabschätzung zu RadixSort

- Pro Segment der Radix-Sortierung werden alle n Werte untersucht.
- Bei k Segmenten ergibt sich ein Gesamtaufwand von $O(n \cdot k)$.

Variante: Radix-Exchange-Sortierung

- Radix-Exchange-Sortierung am Beispiel:
  - Sortiere Post nach der ersten Stelle der Postleitzahl in 10 Fächer.
  - Sortiere jedes Fach (rekursiv) nach der zweiten Stelle der Postleitzahl in wiederum 10 Fächer.
  - Sortiere jedes dieser Fächer nach der dritten Stelle der Postleitzahl in wiederum 10 Fächer.
  - Man benötigt keine 100 000 sondern nur rund 50 Fächer zum Sortieren!



Graphen und Graphalgorithmen

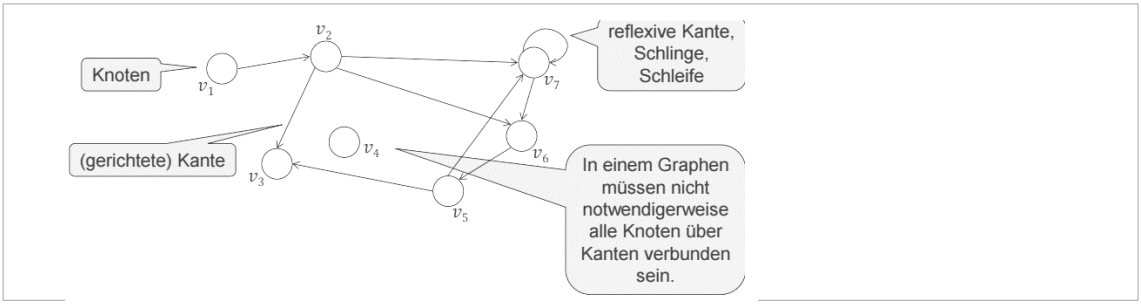
Objekte und Beziehungen

- Gegeben: Menge von Objekten zusammen mit einer Beziehung (Relation) auf diesen Objekten
- Beispiel: Objekt (Personen), Beziehung (Person A kennt Person B)

Gerichtete Graphen

- Ein (gerichteter) Graph (auch: Digraph, engl. directed graph) ist ein Paar $G = (V, E)$, wobei gilt:
  - V ist eine endliche, nichtleere Menge (die Elemente werden Knoten genannt),
  - E ist eine zweistellige Relation auf V: $E \subseteq V \times V$ (die Elemente heißen (gerichtete) Kanten (engl. edge)).
- Eine (gerichtete) Kante ist also ein Paar von Knoten v, w; (gerichtete) Kante (v, w) wird graphisch wie folgt dargestellt:

- Sprechweisen:
  - Kante (v, w) ist inzident mit v und w.
  - □ Knoten v und w sind benachbart, adjazent oder Nachbarn
- Grafische Darstellung eines gerichteten Graphen $G1$:

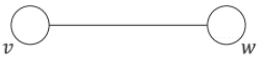


- Mengendarstellung von $G1$:

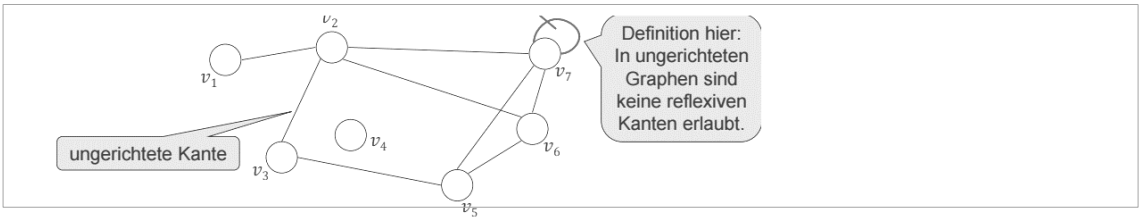
$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$
$$E = \{ (v_1, v_2), (v_2, v_3), (v_2, v_6), (v_2, v_7), (v_5, v_3), (v_5, v_7), (v_6, v_5), (v_7, v_6), (v_7, v_7)\}$$

Ungerichtete Graphen

- Ein ungerichteter Graph ist ein Graph $G = (V, E)$, für den gilt: $\forall\, v_i, v_j \in V$: $(v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E$.
- Bei ungerichteten Graphen ist E also symmetrisch.
- Eine ungerichtete Kante ist ein Paar von Knoten v, w.
- In der grafischen Darstellung gehört zu jedem Pfeil ein Pfeil in die Gegenrichtung. Statt Pfeilpaar zeichnet man eine spitzenlose Linie.
- Kante (v, w) wird graphisch wie folgt dargestellt:



- In der Mengenschreibweise schreibt man $[v_i , v_j]$ für beide Paare
- Grafische Darstellung eines ungerichteten Graphen $G2$

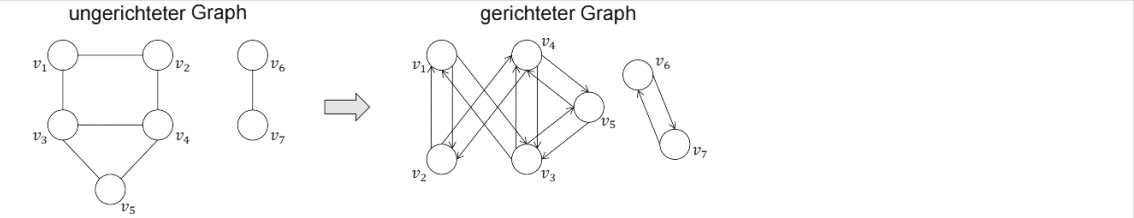


- Mengendarstellung von $G2$:

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$$
$$E = \{ [v_1, v_2], [v_2, v_3], [v_2, v_6], [v_2, v_7], [v_5, v_3], [v_5, v_7], [v_6, v_5], [v_7, v_6]\}$$

Ungerichtete Graphen

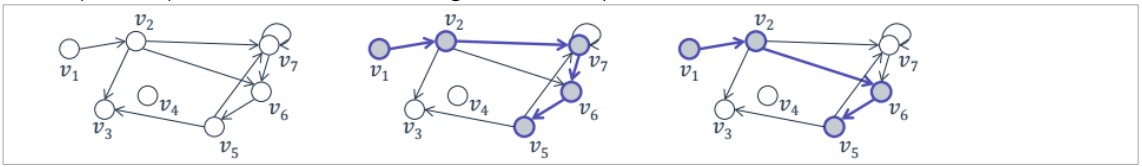
Da eine symmetrische Beziehung ein Spezialfall einer Beziehung ist, lässt sich ein ungerichteter Graph immer durch einen gerichteten Graphen darstellen:



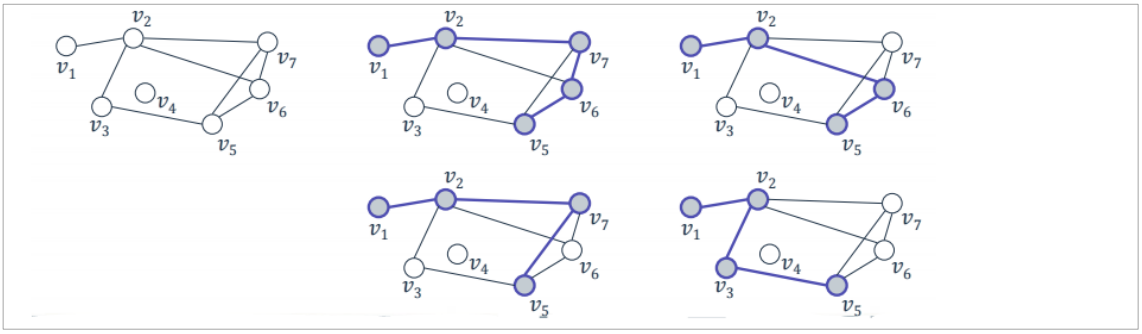
- Beispiel zeigt auch, dass Platzierung von Knoten und Kanten in der graphischen Darstellung keine Rolle spielt.
- Es gibt also verschiedene Darstellungen desselben Graphen.

Pfade in Graphen

- Ein Pfad (Weg, Kantenzug) von v nach w ist eine endliche Folge von Knoten $v = v_1, v_2, \dots, v_n = w$, so dass für $1 \leq i \leq n-1$ gilt $(v_i, v_{i+1}) \in E$.
- Sprechweisen:
  - Der Pfad verbindet v und w.
  - w ist von v aus erreichbar.
- Hinweis: In unserer Graph-Definition kann es keine „parallelen Kanten (mit gleicher Richtung)“ zwischen zwei Knoten geben (sonst sog. Pseudograph).
- Die Länge l eines Pfades ist die Anzahl der Kanten im Pfad, also $l = n - 1$. Ein einzelner Knoten ist demnach ein Pfad der Länge 0.
- Ein Pfad heißt einfach, wenn alle Knoten im Pfad paarweise verschieden sind.
- Zwei (einfache) Pfade von $vv1$ nach $vv5$ im gerichteten Graphen:

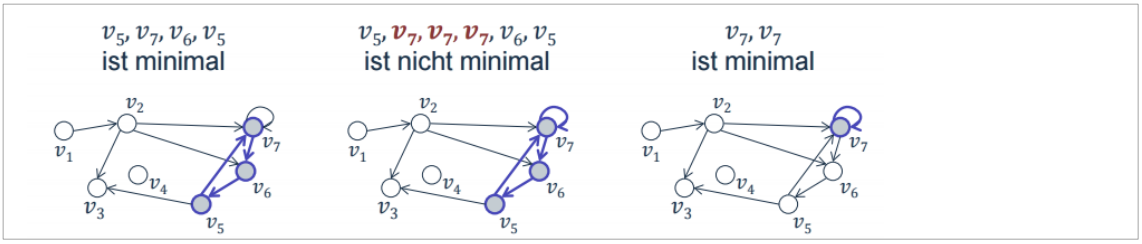


- Vier (einfache) Pfade von v_1 nach v_5 im ungerichteten Graphen:



Zyklen in Graphen

- In einem gerichteten Graphen heißt ein Pfad der Länge $l \geq 1$ von vv nach vv ein Zyklus (von v nach v).
- Ein Zyklus von v nach v heißt minimaler Zyklus (von v nach v), wenn außer v kein anderer Knoten mehr als einmal vorkommt.



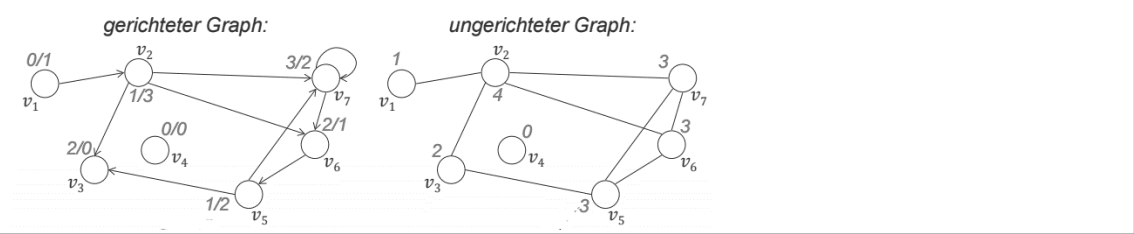
- Bei ungerichteten Graphen muss ein Zyklus mindestens dreiKnoten enthalten.

Vorgänger und Nachfolger

- Ist $(v, w) \in E$, so heißt
  - v direkter Vorgänger (bei gerichteten Graphen auch: Elternknoten) von w,
  - w direkter Nachfolger (bei gerichteten Graphen auch: Kind) von v.
- Falls ein Pfad von v_i nach v_j führt, so heißt
  - v_i Vorgänger (bei gerichteten Graphen auch: Vorfahr) von v_j,
  - v_j Nachfolger (bei gerichteten Graphen auch: Nachkomme) von v_i.
- v_2 und v_7 sind direkte Vorgänger (Elternknoten) von v_6
- v_3, v_6 und v_7 sind direkte Nachfolger (Kinder) von v_2
- v_1 ist Vorgänger (Vorfahr) aller anderer Knoten (bis auf v_4)
- v_6 ist Nachfolger (Nachkomme) von v_1

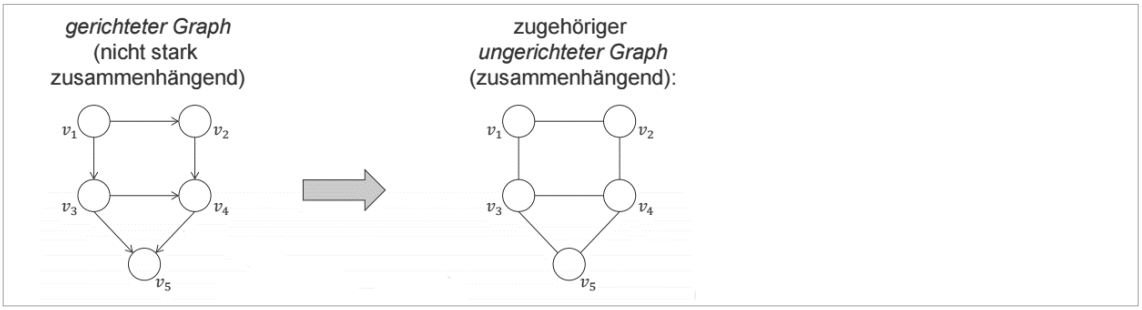
Grad von Knoten und Graph

- Die Anzahl der direkten Vorgänger eines Knotens heißt Eingangsgrad des Knotens. Die Anzahl der direkten Nachfolger eines Knotens heißt Ausgangsgrad des Knotens.
- Bei ungerichteten Graphen spricht man vom Grad des Knotens.
- Ein Knoten mit Grad 0/* heißt Quelle
- Ein Knoten mit Grad */0 heißt Senke.



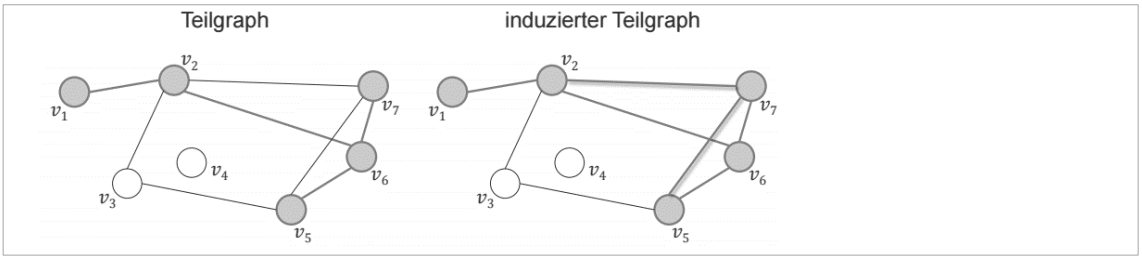
Zusammenhang in Graphen

- Ein gerichteter Graph $G = (V, E)$ heißt stark zusammenhängend (oder: stark verbunden), wenn es für alle $v, w \in V$ einen Pfad
- von v nach w gibt. Bei ungerichteten Graphen lässt man „stark“ weg.
- anschaulich: Jeder Knoten ist von jedem anderen Knoten aus erreichbar.
- Ein gerichteter Graph heißt schwach zusammenhängend, wenn der zugehörige ungerichtete Graph (der durch Hinzunahme aller Rückwärtskanten entsteht) zusammenhängend ist.



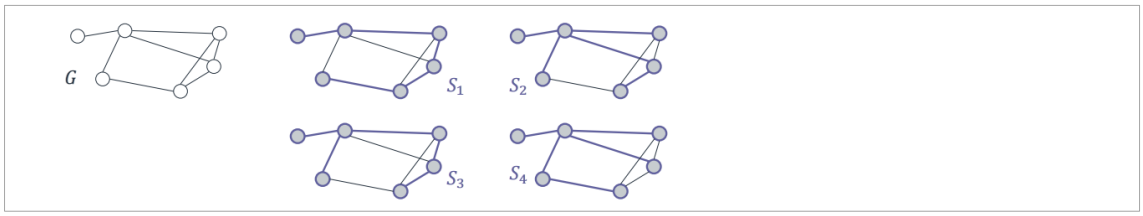
Teilgraphen

- Ein Graph $G' = (V', E')$ ist ein Teilgraph eines Graphen $G = (V, E)$ genau dann, wenn $V' \subseteq V$ und $E' \subseteq E$.
- Ein Graph $G' = (V', E')$ ist ein induzierter Teilgraph eines Graphen $G = (V, E)$ genau dann, wenn $V' \subseteq V$ und $E' = \{(v, w) \in E \mid v, w \in V'\}$ anschaulich: Wähle einige Knoten aus G aus; in E' werden alle Kanten übernommen, die diese ausgewählten Knoten verbinden.



Teilgraphen

- Sind G ein Graph und S ein zyklenfreier Teilgraph von G, der alle Knoten von G enthält, und sind G und S beide zusammenhängend, dann heißt S Spannbaum (auch: aufspannender Baum) von G.



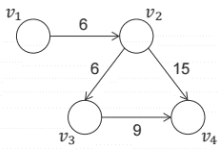
- Ist G ein Baum (siehe frühere LE, Baum ist spezieller ungerichteter Graph), dann gibt es zwischen je zwei verschiedenen Knoten genau einen einfachen Pfad.

Gerichteter azyklischer Graph, Wurzel

Ein Knoten v eines gerichteten azyklischen Graphen (DAG) heißt Wurzel, falls es keine auf ihn gerichteten Kanten gibt.

Bewertete/gewichtete Graphen

- Ein Graph $G = (V, E)$ wird zu einem bewerteten/gewichteten Graphen, indem man eine Gewichtsfunktion $c: E \to \mathbb{N}$ bzw. $c: E \to \mathbb{R}+$ ergänzt, die jeder Kante $e \in E$ ein positives Gewicht $c(e)$ zuordnet.
- Die Kosten (auch: bewertete Länge) $c(p)$ eines Pfades p definiert man in gewichteten Graphen als die Summe der Gewichte seiner Kanten: für $p = (v = v_1, v_2, \dots, v_n = w)$ ist $c(p) = \sum_{i=1}^{n-1} c((v_i, v_{i+1}))$
- Beispiel: Städte mit Straßennetz und Entfernungen, Reisezeiten oder Transportkosten.
- Beispielgraph:
  - Kosten auf jedem Pfad von Knoten v_1 zu Knoten v_4 sind immer 21,
  - auch über den „Umweg“ mit Knoten v_3.

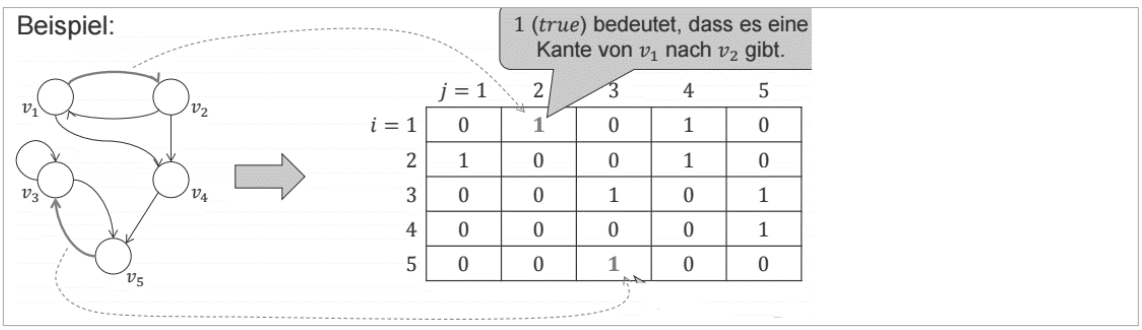


Darstellungen von Graphen

- Bisher betrachtete Darstellungsformen für Graphen:
  - graphische Darstellung: gute Visualisierung für Menschen
  - Mengendarstellung: gut für mathematische Operationen
- Nun gesucht: geeignete Datenstruktur, mit der ein Graph im Rechner repräsentiert werden kann.
- Anforderungen: Datenstruktur soll …
  - … möglichst effizient unterstützen:
    - Feststellung, ob Kante zwischen zwei Knoten existiert
    - Bestimmung aller Nachfolger eines Knoten
    - Graphen-Durchlauf
    - Änderungen am Graphen
  - … möglich speichereffizient sein.

Adjazenzmatrix

- Sei $V = \{v_1, \dots, v_n\}$.
- Die Adjazenzmatrix für $G = (V, E)$ ist eine boolesche $n \times n$-Matrix A mit $A_{ij} = \begin{cases} true, & falls (v_i, v_j) \in E \\ false, & sonst \end{cases}$
- Meist werden true und false durch 1 und 0 dargestellt.



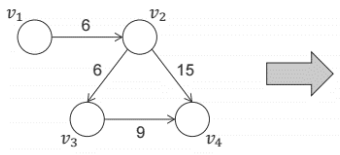
Adjazenzmatrix

- Bei ungerichteten Graphen sind Adjazenzmatrizen symmetrisch, es reicht die Speicherung einer Dreiecksmatrix.
- Graphen ohne Schlingen haben in der Diagonalen stets false.

```
boolean[][] edges = {
        {false, true, false, true, false},
        {true, false, false, true, false},
        {false, false, true, false, true},
        {false, false, false, false, true},
       {false, false, true, false, false}
}
```

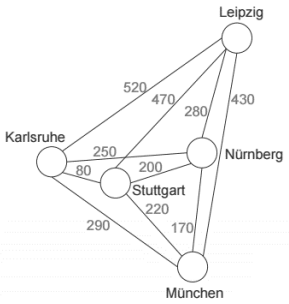
Adjazenzmatrix bei bewerteten/gewichteten Graphen

- Bei bewerteten/gewichteten Graphen speichert man statt true/false die Kantengewichte in der Matrix.
- Fehlende Kanten stellt man mit einem Spezialwert für Unendlich dar: ∞ oder im Programm-Code z.B. Integer.MAX_VALUE/3 (damit man sicher ggf. noch zwei solche Werte addieren kann).



	$j=1$	2	3	4
$i=1$	∞	6	∞	∞
2	∞	∞	6	15
3	∞	∞	∞	9
4	∞	∞	∞	∞

- Entfernungstabelle ist die bekannteste Form der Adjazenzmatrix



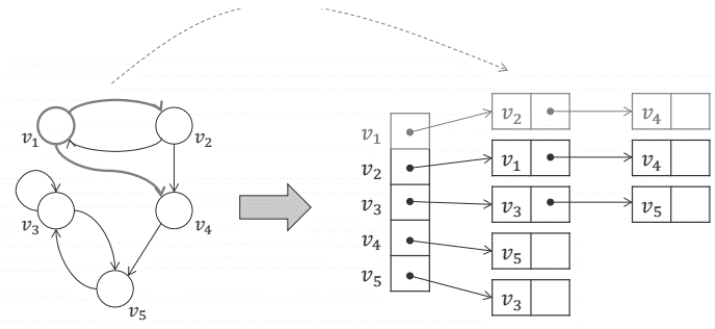
	Karlsruhe	Leipzig	München	Nürnberg	Stuttgart
Karlsruhe	-	520	290	250	80
Leipzig	520	-	430	280	470
München	290	430	-	170	220
Nürnberg	250	280	170	-	200
Stuttgart	80	470	220	200	-

Eigenschaften von Adjazenzmatrizen

- Vorteile:
  - Es ist mit Laufzeit $O(1)$ feststellbar, ob Kante von v nach w existiert.
  - Manche Graphenoperationen lassen sich als Matrizenoperationen darstellen.
- Nachteile (fallen vor allem bei spärlich besetzten Graphen, also bei Graphen mit wenigen Kanten in's Gewicht):
  - Hoher Platzbedarf von $O(n^2)$: Speicherverschwendung, wenn Anzahl der Kanten relativ klein gegenüber dem Quadrat der Knotenanzahl.
  - Auffinden aller k Nachfolger eines Knotens benötigt $O(n)$ Zeit (ebenfalls relativ ungünstig).
  - Initialisierung der Matrix benötigt Laufzeit von $\Theta\ (n^2)$.
- Also: diese Repräsentation wählen, wenn Tests auf das Vorhandensein von Kanten häufig vorkommen.

Adjazenzlisten

- Man verwaltet für jeden der nn Knoten eine Liste seiner (Nachfolger-) Nachbarknoten.
- Über ein Array der Länge $n = |V|$ ist jede Liste direkt zugänglich.



Eigenschaften von Adjazenzlisten

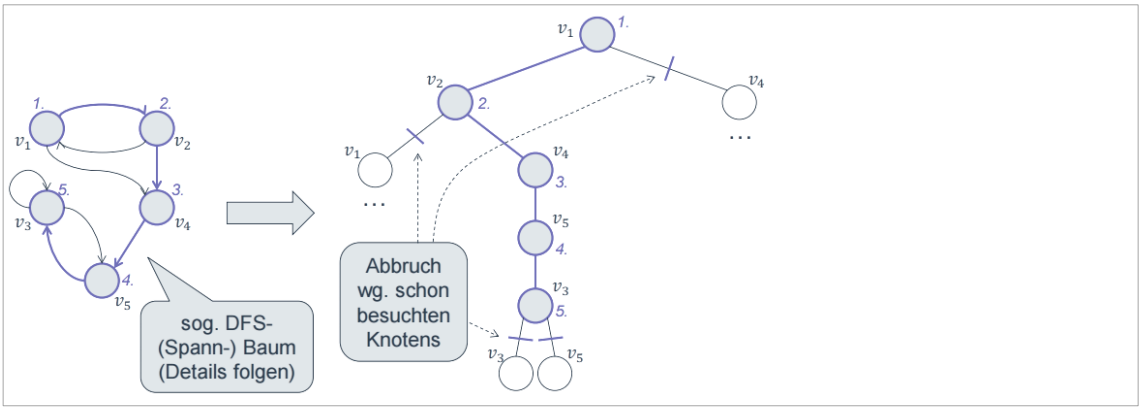
- Vorteile:
  - Geringerer Platzbedarf von $O(\ |V| + |E|\ )$.
  - Alle k Nachfolger eines Knotens sind in Zeit $O(k)$ erreichbar.
- Nachteile:
  - Test, ob v und w benachbart sind, kann nicht mehr in konstanter Zeit durchgeführt werden, da Adjazenzliste von v durchlaufen werden muss, um auf das Vorhandensein von w zu prüfen (und ggf. auch umgekehrt).
- Erweiterung: inverse Adjazenzlisten
  - Falls zu Knoten Vorgängerknoten aufgesucht werden müssen, kann man noch inverse Adjazenzlisten verwalten, die zu jedem Knoten eine Liste seiner Vorgänger enthalten.

Motivation Graphdurchlauf

- Bisher war Durchlaufen von Eingabedaten einfach:
  - Sequentielle Vorgehensweise
  - Betrachtung eines Elements der Eingabedaten nach dem anderen
- Jetzt: Durchlauf von Graphen
  - Gegeben: Graph G
  - Gesucht: Durchlauf/Traversierung des Graphen, der/die jeden Knoten einmal besucht.
  - Aufgabe, die in vielen Graph-Algorithmen bewältigt werden muss.
- Strategien:
  - Tiefensuche (auch: Tiefendurchlauf) (engl. depth first search (DFS) / depth first traversal)
  - Breitensuche (auch: Breitendurchlauf) (engl. breadth first search (BFS) / breath first traversal)

Tiefensuche (engl. depth-first-search – DFS)

Ein Durchlauf eines Graphen G per Tiefensuche (engl. depth-first-search – DFS) entspricht einem Preorder-Durchlauf von G, der jeweils in einem bereits besuchten Knoten von G abgebrochen wird.



- Analogie: Besuch eines Museums mit vielen Gängen, wobei man jeden Gang ablaufen möchte.
- Verfahren:
  - Man läuft in das Museum hinein.
  - Man betritt einen neuen Gang, sobald er sich öffnet. Wenn sich mehrere Gänge gleichzeitig öffnen, wählt man einen (meist den linken)
  - und hinterlässt an der Kreuzung einen Kieselstein.
  - Wenn man an eine Kreuzung stößt, in der bereits ein Kiesel liegt, kehrt man um und geht d. gleichen Weg zurück bis zur vorhergehenden Kreuzung (Gänge u. Kreuzungen können mehrfach betreten werden).
  - Wenn von dieser noch ein unausprobierter Gang abgeht, wird dieser erforscht.
  - Gibt es keinen unausprobierten Gang mehr, muss weiter zurück gegangen werden.
- In der griechischen Mythologie benutzte Ariadne ein Garnknäuel, um Theseus die Rückkehr aus dem Labyrinth zu ermöglichen, in dem er den Minotaurus tötete (sog. Ariadne-Faden).
- Siehe Rücksetzverfahren („backtracking“) .
- Algorithmus (rekursiv):

```
Algorithmus dfs(v):
markiere v als besucht;
für jeden Nachfolger vi von v führe aus: {
 Wenn vi noch nicht besucht wurde,
 dann dfs(vi);
}
```

Tiefensuche

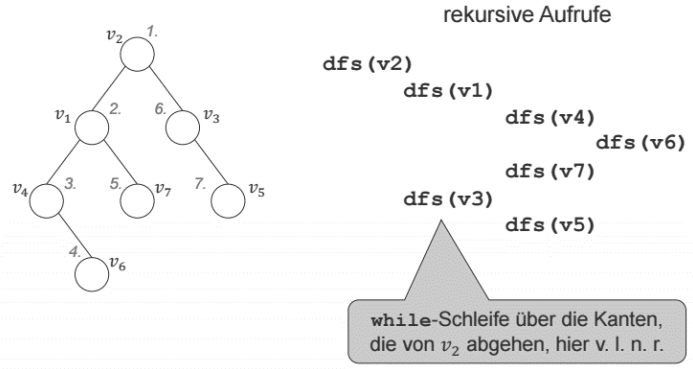
- Z. B. in Array merken, ob $vvii$ besucht wurde: boolean[] visited
- Erinnerung: Der Graph muss nicht zusammenhängend sein. Daher wie folgt um alle Knoten zu besuchen:

```
markiere alle Knoten als unbesucht;
für jeden Knoten v des Graphen führe aus: {
wenn v noch nicht besucht wurde,
dann dfs(v);
}
```

Tiefensuche: rekursive Implementierung

```
void dfs(Node v) {
        // markiere v als besucht
        v.mark();
        // hole alle Kanten zu Nachfolgern;
        // verwaltet als Adjazenzliste
        Iterator<Edge> iter = v.getEdges();
        while (iter.hasNext()) {
                Edge e = (Edge) iter.next();
                if (e.target.isUnmarked()) {
                        dfs(e.target);
                }
        }
}
```

- Jeder Knoten des zusammenhängenden Teilgraphen wird erreicht: sonst gäbe es einen Knoten ohne Markierung, der über eine Kante mit einem markierten Knoten verbunden ist.
- Das kann nicht vorkommen.
- Aufwand $O(|V| + |E|)$:
  - Jede der $|E|$ Kanten wird (höchstens) von beiden Seiten betrachtet.
  - Zusätzlich gibt es ggf. isolierte Knoten in V.
- Rekursion am Beispiel: Besuch der Nachfolger von links nach rechts (v. l. n. r.)



Anwendungsbeispiele für Tiefensuche

- Nachfolgend: Anwendungsbeispiele für eine Tiefensuche
- Dazu erforderlich: Aktionen auf besuchten Knoten.
- Betrachtete Anwendungsbeispiele:
  - Knoten eines Unterbaums eines Knotens zählen.
  - Inorder-/Preorder-/Postorder-Durchlauf eines Ausdruckbaums und dessen Ausgabe in Infix-/Präfix-/Postfix-Form.

Tiefensuche mit Nutzarbeit: rekursive Implementierung

- Graph wird besucht, um eine bestimmte Aufgabe zu erledigen.
- Abhängig von der Aufgabe sind zu konkretisieren:
  - (1) Aktion(en) beim ersten Betreten des Knotens **v**
  - (2) Aktion(en) zwischen den Besuchen der Nachfolger des Knotens **v**
  - (3) Aktion(en) beim Abstieg: im Falle erfüllter Bedingung und immer
  - (4) Aktion(en) nach Besuch aller Nachfolger des Knotens **v**

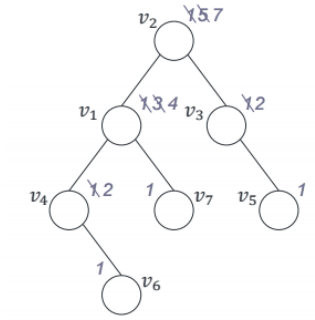
Tiefensuche mit Nutzarbeit: rekursive Implementierung

```
void dfs(Node v) {
        v.mark();
        // preNodeWork(v); (1)
        Iterator iter = v.getEdges();
        while (iter.hasNext()) {
                Edge e = (Edge) iter.next();
                if (e.target.isUnmarked()) {
                        // inNodeWork(v); (2)
                        dfs(e.target);
                        // edgeWorkIf(e); (3)
                }
                // edgeWorkAlways(e);(3)
        }
        // afterNodeWork(v); (4)
}
```

Knotenanzahl des Unterbaums per Tiefensuche bestimmen

- Gegeben: Baum G
- Gesucht: Methode, die für jeden Knoten vv die Anzahl der Knoten des Unterbaums bestimmt, der v als Wurzel hat.
- G ist ein Baum: dort gibt es keine bereits markierten Nachfolger.

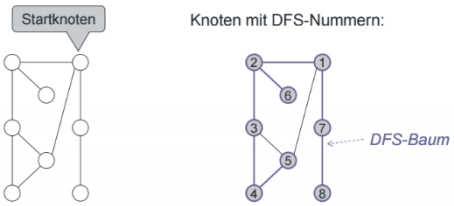
```
void dfsCountSubtreeNodes(Node v) {
        ~~v.mark();~~
        // preNodeWork(v):
        v.counter = 1;
        Iterator iter = v.getEdges();
        while (iter.hasNext()) {
                Edge e = (Edge) iter.next();
                ~~if (e.target.isUnmarked()) {~~
                        dfsCountSubtreeNodes(e.target);
                ~~}~~
                // edgeWorkAlways(e):
                v.counter += e.target.counter;
        }
}
```




```
v2.counter = 1;
    v1.counter = 1;
        v4.counter = 1;
            v6.counter = 1;
        v4.counter += v6.counter; // = 2
    v1.counter += v4.counter; // = 3
        v7.counter = 1;
    v1.counter += v7.counter; // = 4
v2.counter += v1.counter; // = 5
    v3.counter = 1;
        v5.counter = 1;
    v3.counter += v5.counter; // = 2
v2.counter += v3.counter; // = 7
```

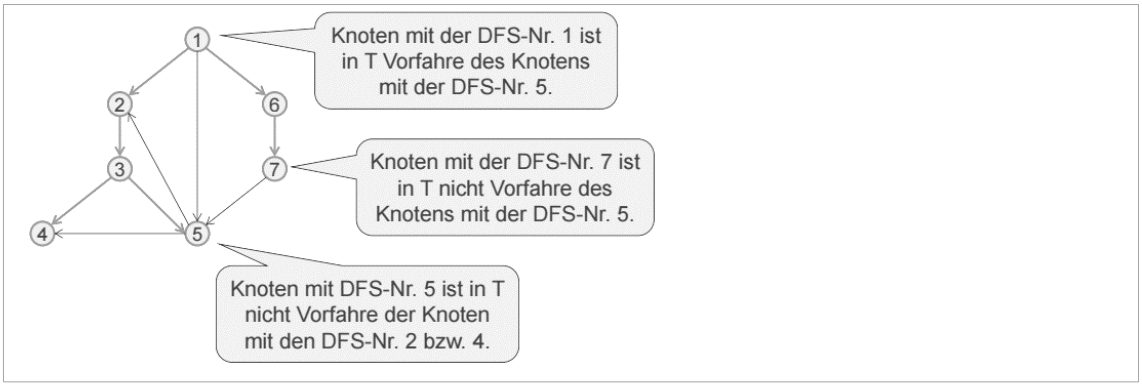
DFS-Nummerierung

- Die Tiefensuche durchläuft die Knoten eines Graphen in einer bestimmten Reihenfolge.
- Wenn jedem Knoten die Position in dieser Reihenfolge zugeordnet wird, ist das eine DFS-Nummerierung.
- DFS-Nummern sind für viele Graph-Algorithmen nützlich.
- Umsetzung mit DFS-Algorithmus:
  - statische Variable dfsPos = 1 vor dem Start initialisieren
  - preNodeWork(v): v.dfsNr = dfsPos++ DFS-Nummer des Knotens wird auf den Wert der statischen Variablen gesetzt; diese wird inkrementiert.
- Prinzip:
  - dicke Kanten verbinden Knoten mit denjenigen Nachfolgern,die die DFS-Nummerierung als noch unmarkiert vorgefunden hat.
  - edgeWorkIf baut aus diesen Kanten den sog. DFS-Baum (oder Tiefensuchbaum, engl.: DFS-tree) auf, einen Spannbaum.



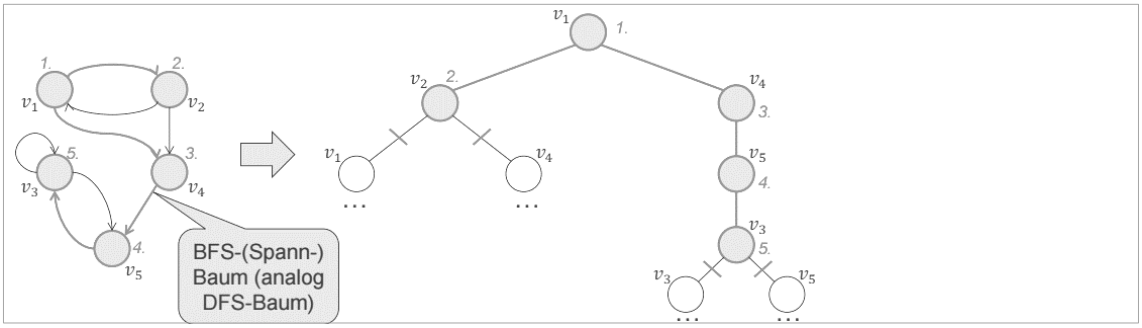
DFS-Baum gerichteter Graphen: Vorfahre

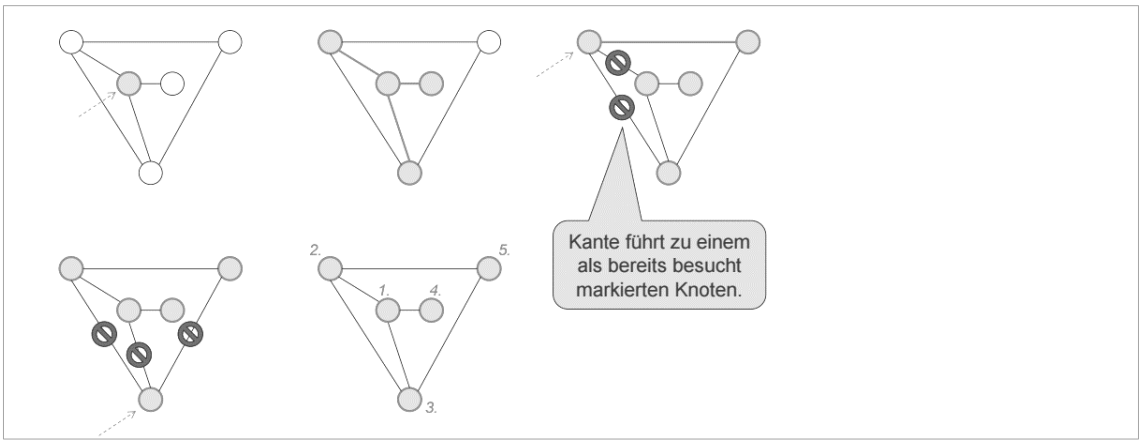
Seien $G = (V, E)$ ein gerichteter Graph und $T = (V, E')$ ein DFS-Baum von G, dann gilt für alle Kanten $e = (v, w) \in E$: wenn $v.\mathrm{dfsNr} < w.\mathrm{dfsNr}$, dann ist v Vorfahre von w in T.



Breitensuche (engl. breadth-first-search – BFS)

Ein Durchlauf eines Graphen G per Breitensuche (engl. breadthfirst-search – BFS) besucht die Knoten von G„ebenenweise“, also zuerst die Knoten, die über einen Pfad der Länge 1 von der Wurzel aus erreichbar sind, dann über einen Pfad der Länge 2 usw. In einem bereits besuchten Knoten von G wird abgebrochen.



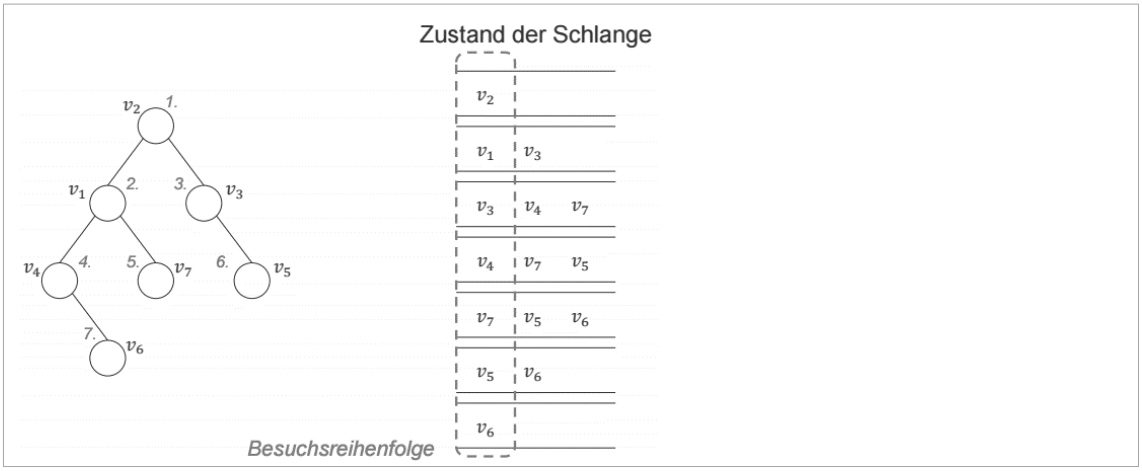


Breitensuche

Idee: benutze Schlange, in der jeweils Nachfolger eines Knotens gemerkt werden, während dessen Geschwister verarbeitet werden. Algorithmus (iterativ):

```
Algorithmus bfs(v):
füge in die leere Schlange q den Knoten v ein;
solange q enthält Elemente führe aus: {
        entnimm q das erste Element =: w;
        markiere w als besucht;
        verarbeite w;
        für jeden Nachfolger wi von w führe aus: {
                wenn wi noch nicht besucht,
                dann hänge wi an q an;
        }
}
```

Breitensuche



- bfs muss für jeden unbesuchten Knoten aufgerufen werden, um das Erreichen jedes Knotens sicherzustellen:

```
markiere alle Knoten als unbesucht;
für jeden Knoten v des Graphen führe aus: {
 wenn v noch nicht besucht wurde,
 dann bfs(v);
}
```

- Aufwand (bei Adjazenzlistendarstellung): $O(\,|V+E|\,)$

Breitensuche: iterative Implementierung mit Schlange

```
void bfs(Node v) {
        Queue<Node> q = new LinkedList<>();
        q.add(v); // merke Wurzel fuer Besuch vor
        while (!q.isEmpty()) {
                // besuche vorderstes Schlangenelement
                v = q.poll();
                v.mark(); // markiere v als besucht
                // hier: ggfs. Aktion auf v;
                Iterator<Edge> iter = v.getEdges();
                while (iter.hasNext()) {
                        // lege alle Nachfolger in Schlange
                        Edge e = iter.next();
                        if (e.target.isUnmarked()) {
                                q.add(e.target);
                        }
                }
        }
}
```

Problemstellung „Kürzeste Wege 1:n“

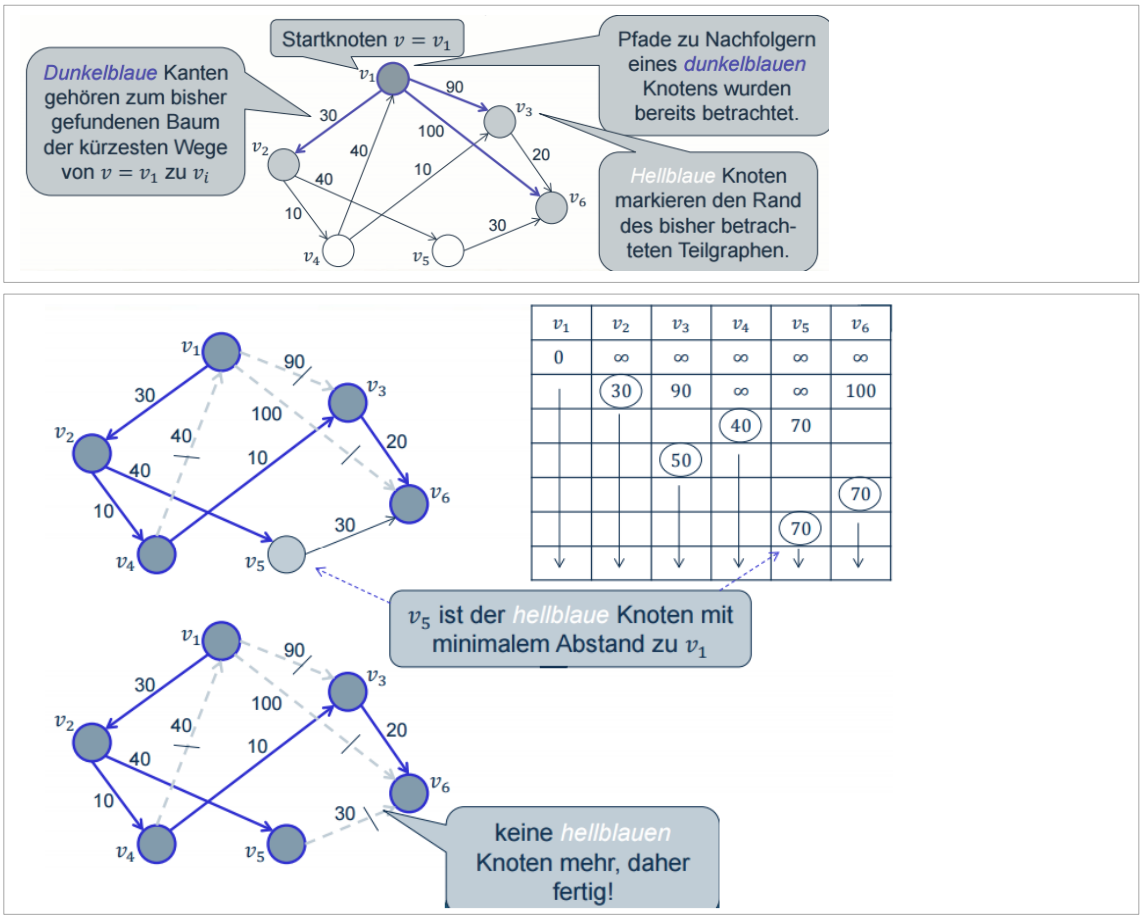
- Gegeben:gerichteter Graph G, dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind.
- Aufgabe:
  - Berechne für einen beliebigen Startknoten vv die kürzesten Wege zu allen anderen Knoten des Graphen.
  - Länge eines Pfades = Summe der Kantenkosten.
  - sog. single source shortest path-Problem

Grundidee des Algorithmus von Dijkstra

- Vom Startknoten vv aus wächst „erkundeter“ Teilgraph von G.
- Knoten
  - ungefärbt: noch nicht vom Algorithmus erkundet.
  - dunkelblau: alle Kanten zu Nachfolgern wurden betrachtet; Nachfolger liegen bereits im erkundeten Teilgraphen; dunkelblaue Knoten im Inneren des Teilgraphen; kürzeste Wege zu ihnen sind gefunden.
  - hellblau: ausgehende Kanten noch nicht betrachtet; hellblaue Knoten bilden Rand des Teilgraphen.
- Kanten
  - ungefärbt: noch nicht vom Algorithmus erkundet.
  - dunkelblau: dunkelblaue Kanten bilden innerhalb des Teilgraphen einen Baum der (bisher gefundenen) kürzesten Wege; in jedem Knoten wird der Abstand von vv verwaltet (über den bisher bekannten kürzesten Weg).
  - hellblau: Kante, die sicher nicht zum Baum der kürzesten Wege gehört (z. B. wegen eines kürzeren Weges daraus entfernt wurde)

Grundidee des Algorithmus von Dijkstra

- Teilgraph wächst schrittweise, indem der hellblaue Knoten w mit minimalem Abstand von v dunkelblau markiert wird.
- Drei Fälle für die Farbe der Nachfolger von w:
  - Ungefärbt (nicht schon im Teilgraphen)→ Knoten hellblau markieren, die Kanten dorthin dunkelblau.
  - Hellblau → ggf. die für sie bisher bekannten kürzesten Pfade (dunkelblaue Kanten) korrigieren (und hellbau markieren).
  - Dunkelblau → alter Pfad dorthin kürzer, neue Kante dorthin hellblau



v_1	v_2	v_3	v_4	v_5	v_6
0	∞	∞	∞	∞	∞
	(30)	90	∞	∞	100
			(40)	70	
		(50)			
					(70)
				(70)	
↓	↓	↓	↓	↓	↓

Algorithmus von Dijkstra

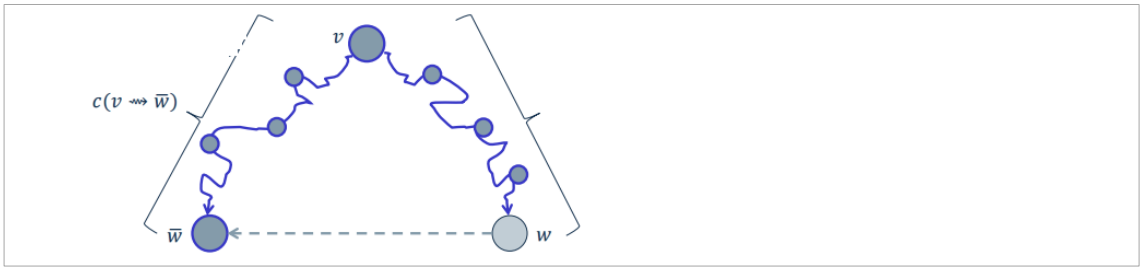
```
Algorithmus dijkstra(v)
dunkelblaueKnoten := ∅; hellblaueKnoten := {v}; dist(v) := 0;
solange hellblaueKnoten ≠ ∅ führe aus: {
        wähle w ∈ hellblaueKnoten, so dass ∀ w' ∈ hellblaueKnoten: dist(w) ≤ dist(w');
        färbe w dunkelblau; //zum Teilgraph hinzunehmen
        für alle Nachfolger wi von w führe aus: {
        falls wi ∉ (dunkelblaueKnoten ∪ hellblaueKnoten) // noch nicht besuchter
        dann führe aus: { // Knoten, ungefaerbt
                färbe wi hellblau;
                färbe die Kante (w, wi) dunkelblau;
                dist(wi) := dist(w) + cost(w,wi);
        }
        sonst falls wi ∈ hellblaueKnoten // wi erneut besucht, hellblau
               dann falls dist(wi) > dist(w) + cost(w,wi)
                       dann führe aus: { // es gibt kuerzeren als bisher
                               färbe die Kante (w, wi) dunkelblau;
                               färbe die bisherige Kante zu wi hellblau;
                               dist(wi) := dist(w) + cost(w,wi);
                       }
                        sonst färbe (w, wi) hellblau;
               sonst färbe (w, wi) hellblau;
        // sonst keine Aktion da wi dunkelblau
        }
}
```

Korrektheit des Algorithmus von Dijkstra

- Lemma:
  - Zu jeder Zeit ist für jeden hellblauen Knoten ww der dunkelblaue Pfad (vom Startknoten) zu ww minimal unter allen blauen Pfaden zu w, das heißt solchen, die nur hellblaue oder dunkelblaue Kanten haben.
- Beweis (per Induktion über die Folge dunkelblau gefärbter Knoten):
  - Ind.-Anfang: Behauptung gilt für vv, da alle Kanten ungefärbt sind.
  - Ind.-Annahme: Aussage gilt für alle bisher hellblau markierten Knoten.
  - Ind.-Schluss:
  - Es wird w dunkelblau gefärbt. Seien $w1$, …, wn die Nachfolger von w. Für die wi sind folgende Fälle zu unterscheiden:
    - wi wurde zum ersten Mal erreicht, war also bisher ungefärbt und wird jetzt hellblau. Der einzige blaue Pfad zu wi hat die Form $v \dashrightarrow w \rightarrow wi$ ($\dashrightarrow$: Pfad; $\rightarrow$: einzelne Kante). Der Pfad $v \dashrightarrow w$ ist nach Induktionsannahme minimal. Also ist $v \dashrightarrow wi$ minimal.
    - wi ist hellblau und wurde erneut erreicht. Der bislang dunkelblaue Pfad zu wi war $v \dashrightarrow x \rightarrow wi$; der neue dunkelblaue Pfad zu wi ist der kürzere der Pfade
      - $v \dashrightarrow x \rightarrow wi$
      - $v \dashrightarrow w \rightarrow wi$
      - Pfad 1 ist minimal unter allen blauen Pfaden, die nicht über w führen, Pfad 2 ist minimal unter allen, die über w führen, also ist der neue Pfad minimal unter allen blauen Pfaden.
    - wi ist dunkelblau : keine Auswirkungen für irgendeinen hellblauen Knoten und Pfade dort hin. (Die ihn erreichende Kante wird hellblau).

Informelle Begründung für Fall c

- $c(v \dashrightarrow \underline{w})$ ist gem. Lemma (bislang) minimal für alle bekannten Wege von v nach $\underline{w}$.
- $c(v \dashrightarrow w)$ ist gem. Lemma (bislang) minimal für alle bekannten Wege von v nach w. Da $\underline{w}$ vor w dunkelblau gefärbt wurde gilt: $c(v \dashrightarrow \underline{w}) \leq c(v \dashrightarrow \mathrm{w})$
- Daher muss diese Kante hellblau sein. $v \dashrightarrow w \rightarrow \underline{w}$ kann nicht kürzer als $v \dashrightarrow \underline{w}$ sein. Denn sonst hätte man zuerst w dunkelblau gefärbt.



Hinweise zum Algorithmus von Dijkstra

- In der Literatur finden sich:
  - verschiedene Färbungsvarianten von Knoten und Kanten mit tlw. auch leicht unterschiedlicher Semantik,
  - die vollständigen Beweise der Korrektheit.
- Dijkstras Algorithmus ist ein weiteres Beispiel eines gierigen Algorithmus.
- Wenn alle Kanten dieselben Kosten haben, dann löst „normale“ Breitensuche das Problem der kürzesten Wege auch (und sogar effizienter).

Implementierung des Algorithmus von Dijkstra: 1.) mit Adjazenzmatrix

- Benötigte Datenstrukturen:
  - Knotenklasse Node, Knotennummerierung von 1 … n
  - Graph G dargestellt durch Kostenadjazenzmatrix cost[i][j] mit Einträgen ∞ an den Matrixelementen, für die keine Kante existiert.
  - Arrays der Größe n für n Knoten:
    - double[] dist; Abstand des jeweiligen Knotens vom Startknoten
    - Node[] parent; Baum der dunkelblauen Kanten, indem zu jedem Knoten sein Vorgängerknoten festgehalten wird.
    - boolean[] darkblueNodes; mit true markiert Elemente der Menge der dunkelblauen Knoten.
- Ablauf:
  - Das gesamte Array dist wird durchlaufen, um den hellblauen Knoten w mit minimalem Abstand zu finden (Aufwand: $O(n)$).
  - Die Zeile cost[w][*] der Matrix wird durchlaufen, um für alle Nachfolger von w ggf. den Abstand und den Vorgängerknoten zu korrigieren (Aufwand: $O(n)$).
  - Zeitaufwand insgesamt $O(n^2)$, da obige Teilschritte n-mal ausgeführt werden.
  - Implementierung ineffizient, wenn nicht n sehr klein oder $|E| \approx n^2$

Implementierung des Algorithmus von Dijkstra: 2.) mit Adjazenzlisten und Halde

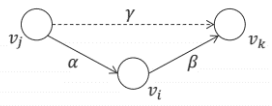
- Benötigte Datenstrukturen:
  - Knotenklasse Node, Knotennummerierung von 1 … n
  - Graph G dargestellt durch Adjazenzlisten mit Kosteneinträgen
  - Arrays dist und parent wie bei 1.)
  - min-Halde der Abstände hellblauer Knoten vom Ausgangsknoten
  - Haldeneinträge und Knoten sind doppelt verkettet, d. h. Haldeneintrag enthält Verweis auf Knoten und Array int[] heapaddress enthält zu jedem Knoten dessen Position in der Halde.
- Ablauf:
  - Entnimm den hellblauen Knoten $wwii$ mit minimalem Abstand zu v aus der Halde (Aufwand: $O(\log |V|)$); für V Knoten: Gesamtaufwand $O(|V| \cdot \log |V|)$ (1)
  - Finde in der entsprechenden Adjazenzliste die mi Nachfolger von wi (Aufwand: $O(mi)$)
    - Für jeden neuen hellblauen Nachfolger erzeuge Eintrag in der Halde (Aufwand: $O(\log |V|)$).
    - Für jeden alten hellblauen Nachfolger korrigiere ggfs. seinen Eintrag in der Halde (via heapaddress).
    - Da sein Abstandswert bei der Korrektur sinkt, kann der Eintrag in der Halde weiter nach oben wandern (Aufwand: O (log $|V|$)).
    - Die Halden-Verweise der vertauschten Einträge im Array heapadress können in $O(1)$ Zeit geändert werden.
    - Gesamtaufwand: $O(mi \log|V|)$; mit $\sum mi = |E|$ deshalb $O(|E| \cdot \log |V|)$ (2)
  - Gesamtaufwand (1)+(2) mit $n = |V|$: $O((|V| + |E|) \cdot \log |V|)$
  - Es gibt Optimierungen, die sogar $O(|E| + |V| \cdot \log |V|)$ erreichen

Problemstellung „Kürzeste Wege n:m“

- Gegeben:gerichteter Graph G, dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind
- Aufgabe:
  - Berechne die kürzesten Wege zwischen allen Paaren von Knoten des Graphen.
  - Länge eines Pfades = Summe der Kantenkosten.
  - sog. all pairs shortest path-Problem

Algorithmus von Floyd

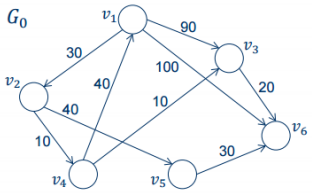
- Gegeben:gerichteter Graph G mit n Knoten, die von 1 bis n durchnummeriert sind.
- Algorithmus berechnet Folge von Graphen $G_0, \dots, G_n$, wobei G_i durch Modifikation von G_{i-1} entsteht.
- Definition von G_i:
  - Jeder Graph G_i hat die gleiche Knotenmenge $\{v_1, \dots, v_n\}$ wie G.
  - Es existiert in G_i eine (direkte) Kante $v \xrightarrow{\alpha} w$ genau dann, wenn ein Pfad von v nach w in G existiert, der nur Knoten aus der Menge $\{v_1, \dots, v_i\}$ als Zwischenknoten verwendet. Der kürzeste derartige Pfad hat Kosten α.
- Der Algorithmus startet mit $G_0 = G$.
- Berechnung von G_i aus G_{i-1}:
  - Sei der Knoten v_j ein Vorgänger des Knotens v_i im Graphen G_{i-1} und sei v_k einer seiner Nachfolger.
  - Betrachte alle solche Paare (v_j, v_k) 1. Falls noch keine (direkte) Kante von v_j nach v_k existiert, so erzeuge eine Kante $v_j \xrightarrow{\alpha+\beta} v_k$.

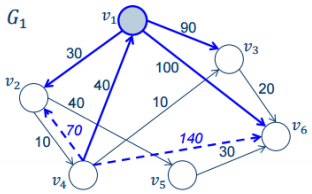


  - Falls schon eine (direkte) Kante $v_j \xrightarrow{\gamma} v_k$ existiert, so ersetze die Markierung γ dieser Kante durch $\alpha + \beta$, falls $\alpha + \beta < \gamma$ (also falls man günstiger über v_i von v_j nach v_k gelangt).

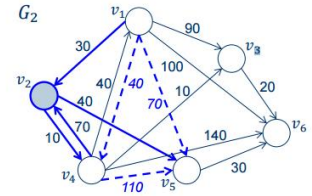
- G_i erfüllt wiederum die oben geforderten Eigenschaften:
  - In G_{i-1} waren alle kürzesten Pfade bekannt und durch Kanten repräsentiert, die als Zwischenknoten nur Knoten aus $\{v_1, \dots, v_{i-1}\}$ benutzten.
  - Jetzt sind in G_i alle Pfade bekannt, die Knoten $\{v1, \dots, vi\}$ benutzen.
  - Deshalb sind nach n Schritten die Kosten aller kürzesten Wege von jedem Knoten zu jedem anderen Knoten aus G im letzten Graphen G_n repräsentiert.

Algorithmus von Floyd am Beispiel

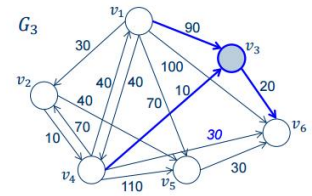




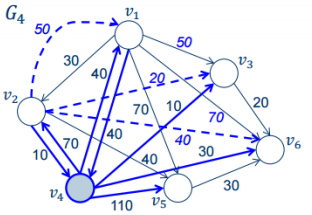
- Betrachte alle Paare von Vorgängern $\{v_4\}$ und Nachfolgern $\{v_2, v_3, v_6\}$ des Knotens v_1.
- Ergänze die Kanten (v_4, v_2) und (v_4, v_6) mit entsprechenden Kosten.
- (v_4, v_3) bleibt unverändert.



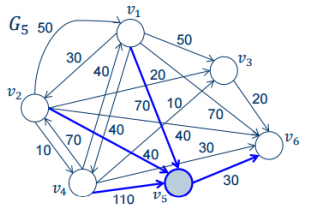
- Betrachte alle Paare von Vorgängern $\{v_1, v_4\}$ und Nachfolgern $\{v_4, v_5\}$ des Knotens v_2.
- Ergänze die Kanten (v_1, v_4), (v_1, v_5) und (v_4, v_5) mit entsprechenden Kosten.



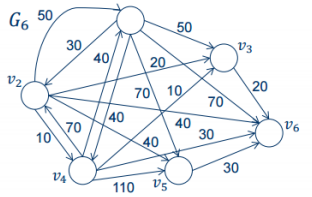
- Betrachte alle Paare von Vorgängern $\{v_1, v_4\}$ und Nachfolgern $\{v_6\}$ des Knotens v_3.
- (v_1, v_6) bleibt unverändert.
- (v_4, v_6) erhält neue Markierung.



- Betrachte alle Paare von Vorgängern $\{v_1, v_2\}$ und Nachfolgern $\{v_1, v_2, v_3, v_5, v_6\}$ des Knotens v_4.
- (v_1, v_2), (v_1, v_5), (v_2, v_5) bleiben unverändert.
- (v_1, v_3), (v_1, v_6) erhält neue Markierung.
- Ergänze die Kanten (v_2, v_1), (v_2, v_3) und (v_2, v_6) mit entsprechenden Kosten.



- Betrachte alle Paare von Vorgängern $\{v_1, v_2, v_4\}$ und Nachfolgern $\{v_6\}$ des Knotens v_5.
- (v_1, v_6), (v_2, v_6) und (v_4, v_6) bleiben unverändert.



- Da v_6 keine Nachfolger hat, ist $G_6 = G_5$.

Implementierung des Algorithmus von Floyd mit Adjazenzmatrix

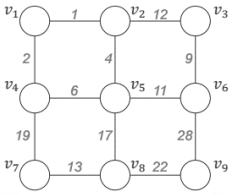
- gegeben: Graph GG mit Adjazenzmatrix cost[j][k]
  - cost[i][i] = 0 für alle v_i
  - cost[j][k] = ∞, falls in GG keine (direkte) Kante von v_j nach v_k

Implementierung des Algorithmus von Floyd mit Adjazenzmatrix

```
public double[][] floyd() {
        int n = cost.length;
        double[][] g = new double[n][n];
        for (int j = 0; j < n; j++) { // G0 = G initialisieren
              for (int k = 0; k < n; k++) {
                      g[j][k] = cost[j][k];
               }
        }
        for (int i = 0; i < n; i++) { //vi
               for (int j = 0; j < n; j++) { //vj
                     for (int k = 0; k < n; k++) { (( vk Laufzeit O(|V|^3)
                            if (g[j][i] + g[i][k] < g[j][k]) { // α + β < γ
                                   g[j][k] = g[j][i] + g[i][k];
                             }
                     }
              }
        }
        return g;
}
```

Minimaler Spannbaum Problemstellung

- Ziel: finde den jeweils bezüglich eines Kostenmaßes „billigsten“ Spannbaum eines Graphen.
- Anwendungsbeispiel:
  - Verlegung neuer Kabel für Kabel-TV in einem Wohngebiet.
  - Kabel dürfen nur an bestimmten Pfaden (z. B. parallel/quer zu Straßen) entlang laufen.
  - Pfade haben unterschiedliche Kosten.
  - Wie sind alle Haushalte des Wohngebiets möglichst günstig zu versorgen?
- Gegeben:ungerichteter Graph G, dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind
- Aufgabe:
  - Berechne minimalen Spannbaum dieses Graphen.
  - Dabei ist der Spannbaum minimal, wenn die Summe seiner Kantengewichte minimal ist.

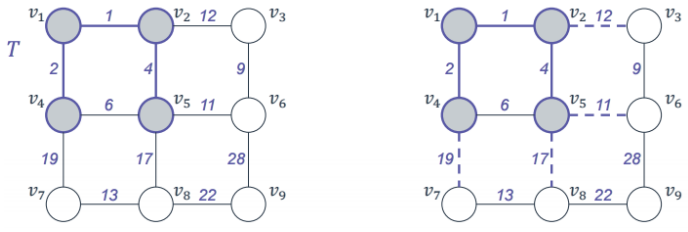


Erste Idee: Algorithmus von Dijkstra verwenden

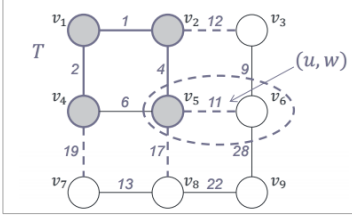
- Der Algorithmus von Dijkstra für die Suche nach dem kürzesten Pfad lieferte für einen gegebenen Knoten einen Spannbaum kürzester Pfade.
- In jedem Schritt wurde eine Kante so hinzugefügt, dass der entstehende Pfad minimale Länge hatte.
- Aufwand (mit Optimierung): $O(|E| + |V| \cdot \log|V|)$
- Strategie für minimalen Spannbaum:
  - Bestimme für jeden der $|V|$ Knoten den Spannbaum kürzester Pfade.
  - Wähle den kleinsten dieser Spannbäume aus.
- $|V|$-fache Anwendung des Algorithmus von Dijkstra, Gesamtaufwand: $O\ |V| \cdot (|E| + |V| \cdot \log|V|)$.

Grundidee des Algorithmus von Prim

- Angenommen man hat einen Teilgraph T von G gefunden, der auch Teilgraph des minimalen Spannbaums ist
- Es gibt Knoten, die über eine Kante mit T verbunden sind, die aber noch nicht zu T gehören
- Frage: Wie kann man T um eine weitere Kante vergrößern?



- Die Kante mit dem kleinsten Gewicht, die einen zu T benachbarten Knoten verbindet, wird zum minimalen Spannbaum hinzugenommen.



Grundidee des Algorithmus von Prim

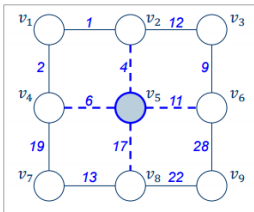
- Korrektheitsüberlegungen
  - (u, w) sei die billigste Kante, die einen Knoten von T mit einem Knoten außerhalb von T verbindet.
  - (u, w) gehört zum minimalen Spannbaum.
  - Andernfalls gäbe es einen Pfad von irgendeinem Knoten aus T zu w mit niedrigeren Kosten.
  - Da jeder Pfad T verlassen muss, hat dieser Pfad mindestens die Kosten der ersten Kante, die T verlässt.
  - Diese sind aber höher als die Kosten der Kante $(u\ w)$.
  - Einen kleineren Spannbaum bekäme man also, indem man diese erste Kante durch (u, w) ersetzt.

Algorithmus von Prim

```
Algorithmus prim():
Wähle einen beliebigen Knoten als Startgraph T.
Solange T noch nicht alle Knoten enthält, führe aus: {
        Wähle eine Kante e mit minimalen Kosten aus, die
                einen noch nicht in T enthaltenen Knoten v mit
                T verbindet.
 Füge e und v dem Graphen T hinzu.
}
```

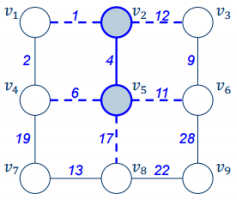
- Hinweis:
  - Der Algorithmus wählt in jedem Schritt eine Kante mit minimalen Kosten aus der Menge zu betrachtender Kanten aus.
  - Der Algorithmus von Prim ist damit ein weiteres Beispiel für einen gierigen Algorithmus.

Algorithmus von Prim am Beispiel



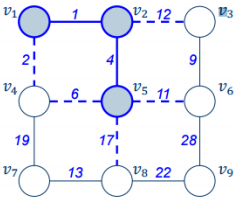
- Wähle als Startgraph den Knoten v_5.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_2, v_5), *Kosten: 4*
  - (v_4, v_5), Kosten: 6
  - (v_5, v_6), Kosten: 11
  - (v_5, v_8), Kosten: 17

Wähle Kante (v_2, v_5) mit minimalen Kosten für T.



- Ergänze Knoten v_2 und Kante (v_2, v_5) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_1, v_2), *Kosten: 1*
  - (v_4, v_5), Kosten: 6
  - (v_5, v_6), Kosten: 11
  - (v_2, v_3), *Kosten: 12*
  - (v_5, v_8), Kosten: 17

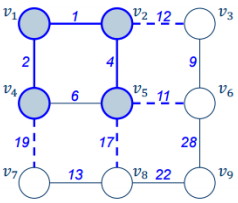
Wähle Kante (v_1, v_2) mit minimalen Kosten für T.



- Ergänze Knoten v_1 und Kante (v_1, v_2) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_1, v_4), *Kosten: 2*
  - ~~(v_4, v_5), Kosten: 6~~
  - (v_5, v_6), Kosten: 11
  - (v_2, v_3), Kosten: 12
  - (v_5, v_8), Kosten: 17

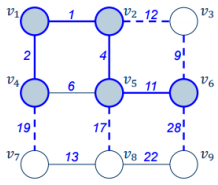
Wähle Kante (v_1, v_4) mit minimalen Kosten für T.

Durch Hinzunahme von v_4 in T wird diese Kante gestrichen.



- Ergänze Knoten v_4 und Kante (v_1, v_4) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_5, v_6), *Kosten: 11*
  - (v_2, v_3), Kosten: 12
  - (v_5, v_8), Kosten: 17
  - (v_4, v_7), *Kosten: 19*

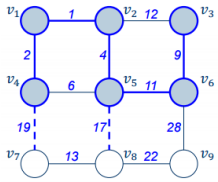
Wähle Kante (v_5, v_6) mit minimalen Kosten für T.



- Ergänze Knoten v_6 und Kante (v_5, v_6) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_3, v_6), *Kosten: 9*
  - ~~(v_2, v_3), Kosten: 12~~
  - (v_5, v_8), Kosten: 17
  - (v_4, v_7), Kosten: 19
  - (v_6, v_9), *Kosten: 28*

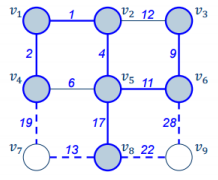
Wähle Kante (v_3, v_6) mit minimalen Kosten für T.

Durch Hinzunahme von v_3 in T wird diese Kante gestrichen.



- Ergänze Knoten v_3 und Kante (v_3, v_6) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_5, v_8), *Kosten: 17*
  - (v_4, v_7), Kosten: 19
  - (v_6, v_9), Kosten: 28

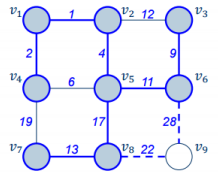
Wähle Kante (v_5, v_6) mit minimalen Kosten für T.



- Ergänze Knoten v_8 und Kante (v_5, v_8) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_7, v_8), *Kosten: 13*
  - ~~(v_4, v_7), Kosten: 19~~
  - (v_8, v_9), *Kosten: 22*
  - (v_6, v_9), Kosten: 28

Wähle Kante (v_7, v_8) mit minimalen Kosten für T.

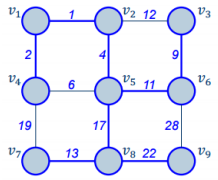
Durch Hinzunahme von v_7 in T wird diese Kante gestrichen.



- Ergänze Knoten v_7 und Kante (v_7, v_8) in T.
- zu untersuchende Kanten (sortiert nach Kosten):
  - (v_8, v_9), *Kosten: 22*
  - ~~(v_6, v_9), Kosten: 28~~

Wähle Kante (v_5, v_6) mit minimalen Kosten für T.

Durch Hinzunahme von v_9 in T wird diese Kante gestrichen.



- Ergänze Knoten v_9 und Kante (v_8, v_9) in T.
- Minimaler Spannbaum gefunden, fertig.

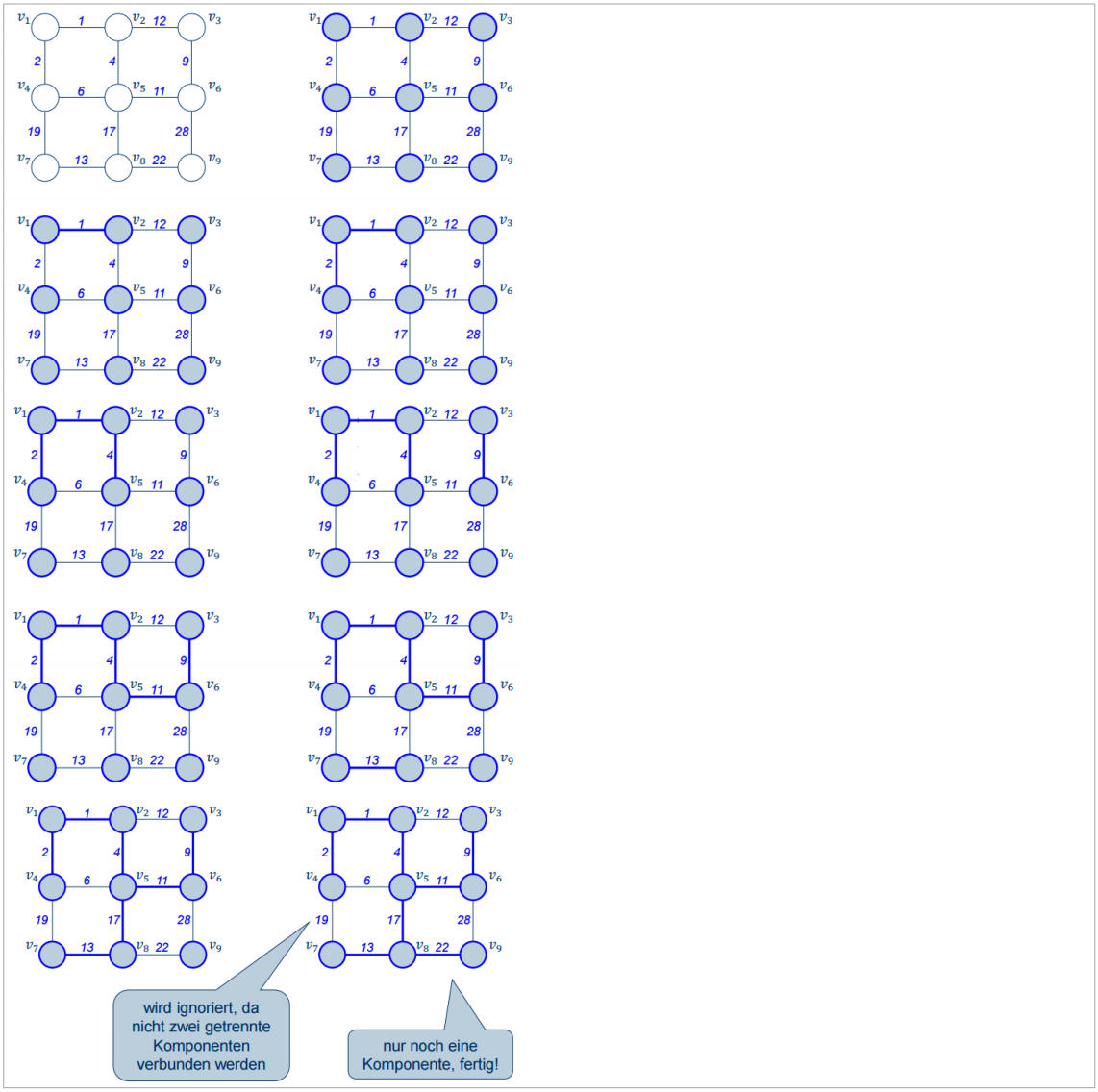
Aufwand des Algorithmus von Prim

- Verwende min-Halde als Prioritätswarteschlange zur Verwaltung der noch zu betrachtenden Kanten.
- Damit kann (mit Optimierung) als Gesamtaufwand $O(|E| + |V| \cdot \log|V|)$ erreicht werden.
- Gut, wenn der Graph viele Kanten hat.

Grundidee des Algorithmus von Kruskal

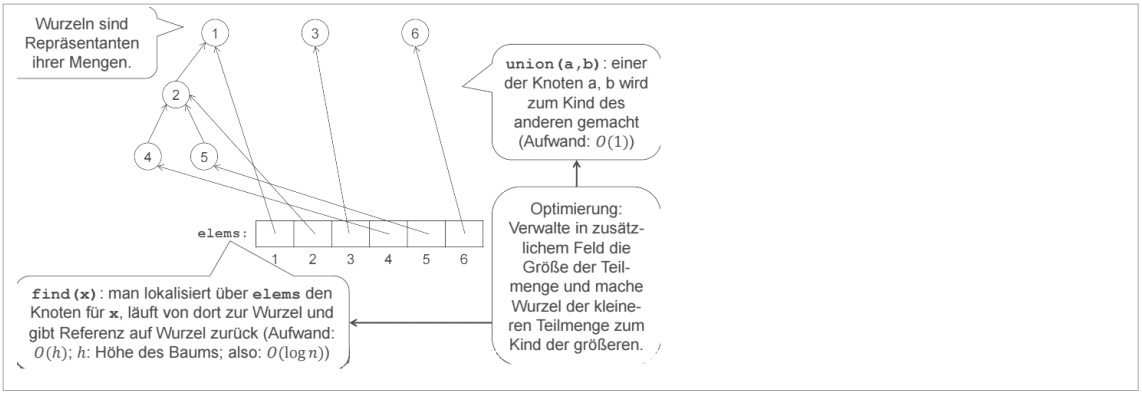
- Minimale Spannbäume haben eine interessante Eigenschaft:
- Lemma:
  - Sei für $G = (V, E)$ $\{U, W\}$ eine restlose und überschneidungsfreie Zerlegung der Knotenmenge V.
  - Sei u, w eine Kante in G mit minimalen Kosten unter allen Kanten $\{(u', w')\}$ $u' \in U, w' \in W$.
  - Dann gibt es einen minimalen Spannbaum T für G, der $(u, ww$ enthält.
- Beweis:
  - Sei $T = (V, E')$ ein beliebiger minimaler Spannbaum für G, der (u, w) nicht enthält.
  - Dann gibt es in T mindestens eine, vielleicht aber auch mehrere Kanten, die U mit W verbinden
  - Wenn man nun T die Kante $(u, ww$ hinzufügt, entsteht ein Graph T' mit einem Zyklus, der über (u, w) verläuft.
  - u muss also noch auf einem anderen Weg mit w verbunden sein, der mindestens eine weitere Kante (u', w') enthält, die U mit W verbindet.
  - Wenn man nun (u', w') aus T' löscht, entsteht ein Spannbaum T'',dessen Kosten höchstens so hoch sind, wie die von T, da (u, w) minimale Kosten hat.
  - Also ist T'' ein minimaler Spannbaum, der (u, w) enthält.
- Das Lemma ist die Grundlage des Algorithmus von Kruskal zur Konstruktion minimaler Spannbäume, der wie folgt vorgeht:
  - Zu Anfang sei T ein Graph, der genau die Knoten von G enthält, aber keine Kanten.
  - Die Kanten von G werden nun in der Reihenfolge aufsteigender Kosten betrachtet.
  - Wenn eine Kante zwei getrennte Komponenten von T verbindet, so wird sie in den Graphen T eingefügt und die Komponenten werden verschmolzen.
  - Anderenfalls wird die Kante ignoriert.
  - Sobald T nur noch aus einer einzigen Komponente besteht, ist T ein minimaler Spannbaum für G.

Algorithmus von Kruskal am Beispiel

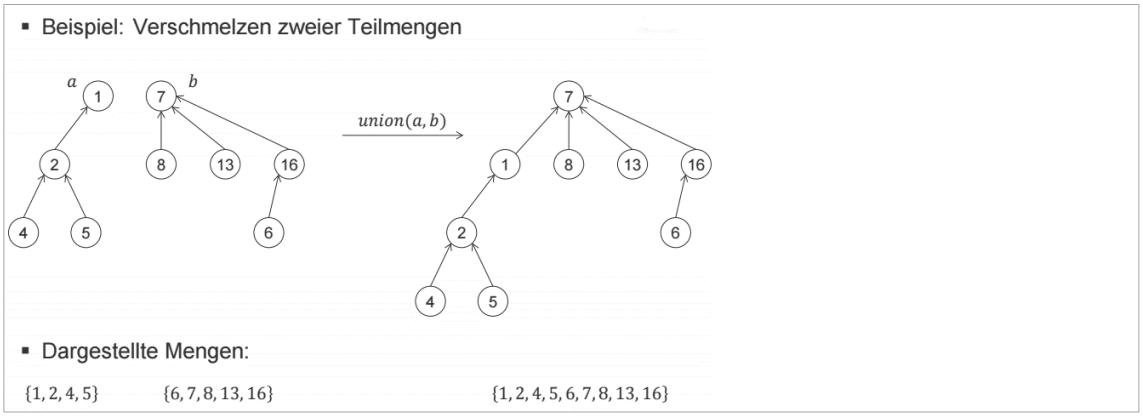


Einschub: Partitionen von Mengen mit union und find

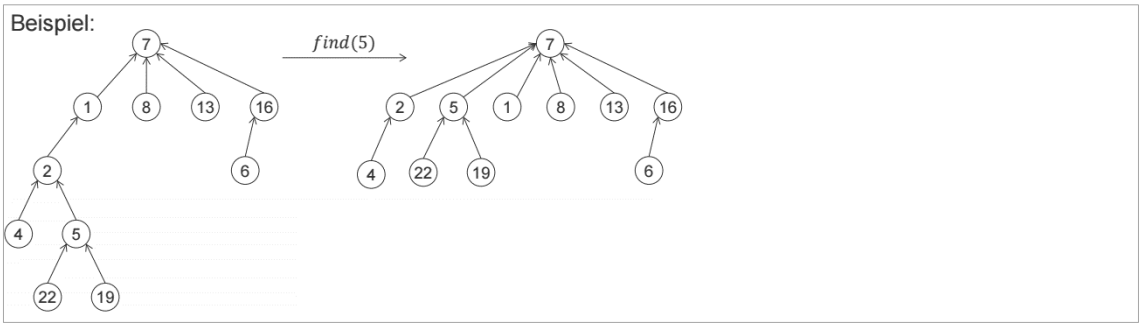
- Aufgabe:
  - Verwaltung einer Zerlegung einer Menge in Teilmengen (sog. Partition einer Menge).
  - Benötigte Operationen:
  - union verschmilzt zwei Teilmengen, d. h. bildet die Vereinigung ihrer Elemente,
  - find stellt für ein gegebenes Element fest, zu welcher Teilmenge es gehört.
- Implementierung mit Bäumen
  - Darstellung jeder Teilmenge durch einen Baum, dessen Knoten Elemente der Teilmenge repräsentieren und in dem Verweise jeweils vom Kind zum Elternknoten führen.
  - Wurzeln sind Repräsentanten der Mengen ihrer Kinder.
- Zusätzlich: Array, um die Knoten im Baum direkt zu erreichen.
- Beispiel: Zerlegung der Menge {1, 2, 3, 4, 5, 6} in die Teilmengen {1, 2, 4, 5 }, {3} und {6} .



Einschub: Partitionen von Mengen mit union und find



- Letzte Verbesserung: sog. Pfadkompression
- Folge von nn find-Operationen könnte $OO(n \log n)$ Zeit brauchen.
- Idee: bei jedem find-Aufruf alle Knoten auf dem Pfad zur Wurzel direkt zu Kindern der Wurzel machen.



Einschub: Partitionen von Mengen mit union und find

- Aufwand abhängig von der Höhe der Bäume
  - Höhe der Bäume im Worst-Case: $O(n)$
  - Höhe der Bäume unter Beachtung der Größe bei union-Operation: $O(\log n)$
- Aufwand für m union-/find-Operationen auf n Elementen: $O((m + n) \log n)$
- Aufwand mit Pfadkompression (ohne Beweis): $O(m + n)$ für alle praktisch relevanten n und m

Aufwand des Algorithmus von Kruskal

- Jede Kante wird aus der Halde entfernt: $O(|E| \cdot \log|E|)$.
- Aufwand für Baumerweiterung und Test auf Zyklenfreiheit?
  - Union-Find-Datenstruktur“
  - Initial bildet jeder Knoten seine eigene Menge: $\{v_0\}\{v_1\} \ldots \{v_{n-2}\}\{v_{n-1}\}$
  - Baumerweiterung um Kante (u, v) mittels $\mathbf{union}(u, v)$
  - Test auf Zyklenfreiheit inkl. Kante (u, v) mittels $\mathbf{find}(u)$ =? = $\mathbf{find}(v)$
  - $|E|$ union-/find-Operationen: $O(|E| + |V|)$
- Gesamtaufwand: $O(|E| \cdot \log|E|)$
- Gut, wenn der Graph wenige Kanten (und evtl. viele Knoten hat).