Repetitorium PROG

Modul prog, WS23/24
Trier University of Applied Sciences
Informatik Fernstudium (M.C.Sc.)

22.03.2024
Thorsten Suckow-Homberg, `https://thorsten.suckow-homberg.de`

Hinweise

Syntax und Features Java-bezogener Themen basieren auf der Java Standard Edition 21[1].

[1] `https://docs.oracle.com/javase/specs/jls/se21/html/index.html` - abgerufen 22.03.2024

Inhaltsverzeichnis

1

Grundlagen

1.1 Kompilierungseinheit

In Abbildung 1.1 ist eine *Übersetzungseinheit*[1] in Java dargestellt.
Eine Übersetzungseinheit deklariert eine Klasse mit ihren Methoden und Attributen und kann dann dem Compiler zur Übersetzung übergeben werden.

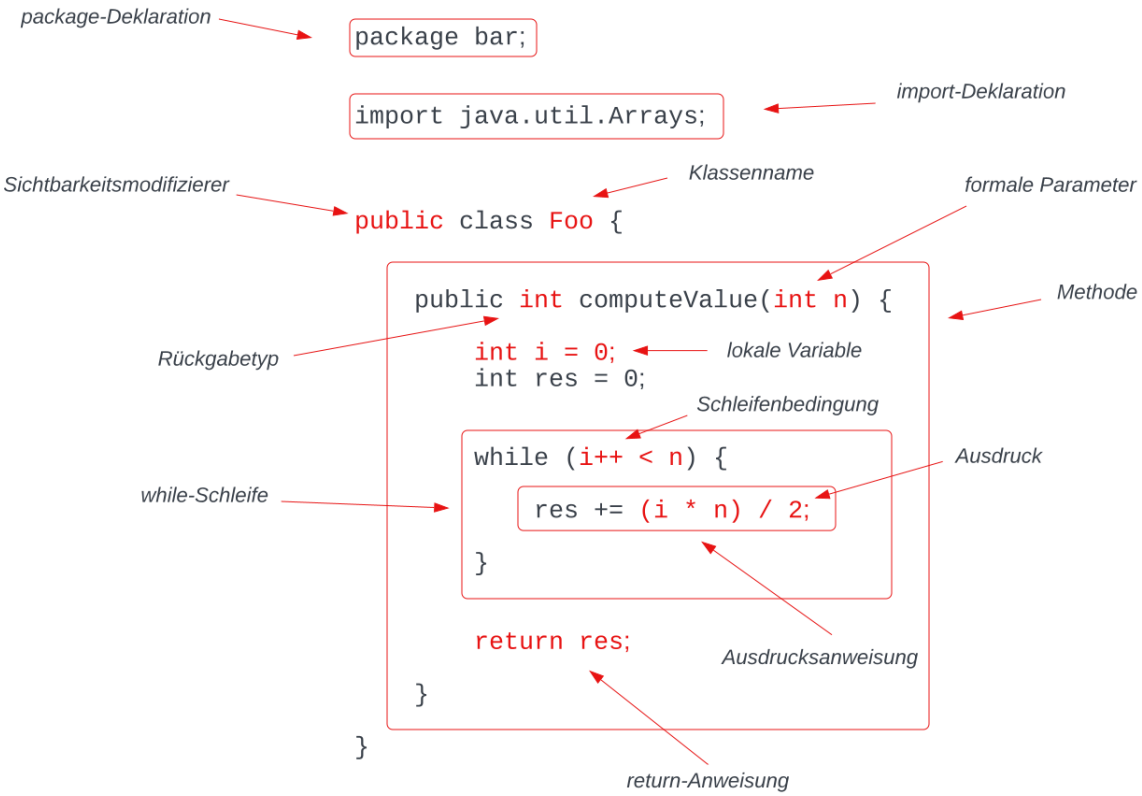

Abb. 1.1: Beispiel für eine Struktur einer Übersetzunsgeinheit in Java. (Quelle: eigene)

[1] *Compilationseinheit* bei [Ull23, 100]

1.2 Einfache Datentypen

Typ	Inhalt	Default	Bits	Wertebereich
boolean	`true`/ `false`	`false`	1	$\{true, false\}$
char	Unicode	\u0000[2]	16	\u0000 - \uffff[3]
byte	Ganze Zahl (+/-)	0	8	$[-2^{8-1}, 2^{8-1}-1]$[4]
short	Ganze Zahl (+/-)	0	16	$[-2^{16-1}, 2^{16-1}-1]$
int	Ganze Zahl (+/-)	0	32	$[-2^{32-1}, 2^{32-1}-1]$
long	Ganze Zahl (+/-)	0	64	$[-2^{64-1}, 2^{64-1}-1]$
float	IEEE754[5] Fließkommazahl (+/-)	0.0	32	$s * m * 2^{e-N+1}$
double	IEEE754 Fließkommazahl (+/-)	0.0	64	$s * m * 2^{e-N+1}$

Tabelle 1.1: Einfache Datentype in Java.

1.2.1 Anmerkungen zu Float und Double

Fließkommaliterale sind in Java standardmäßig vom Typ `double`.
`float` kann durch explizites casting erzwungen werden, oder indem man ein `f` (oder `F`) an das Literal anhängt:

```
float  x = 0.2f; // Postfix zwingend erforderlich, ansonsten kommt es
                 // zu einem Compilerfehler: Es kann nicht implizit
                 // von double nach float konvertiert werden (``possible
                 // lossy conversion from double to float``)
double y = 0.2;
float  z = (float)y;
```

Wertebereich

Der Wertebereich für `float` bzw. `double` kann durch den Term $s * m * 2^{e-N+1}$ ausgedrückt werden, wobei

- $s \in \{+1, -1\}$
- $m \in (0, 2^N)$[6]
- $e \in [E_{min}, E_{max}]$, mit $E_{min} = -2(^{K-1})$ und $E_{max} = 2^{K-1} - 1$
- N und K sind abhängig von dem verwendeten Typ, siehe Tabelle 1.2

[6] offenes Intervall

	N	K	E_{max}	E_{min}
`float`	24	8	+127	−126
`double`	53	11	+1023	−1022

Tabelle 1.2: Parameter zur Ermittlung der Wertebereiche von *float* und *double*[7].

1.3 Literale für Zahlen

Beispiele

- `byte`: `8`
- `short`: `128`
- `int`: `1024`
- `long` `4096` oder `4096L` (bzw. `l` statt `L`)
- `float` `123.0F` (bzw. `f` statt `F`)
- `double` `123.0D` (bzw. `d` statt `D`)

1.4 Zahlensysteme

In Java geben führende Zeichen bei Ganzzahl-Literalen das verwendete Zahlensystem an.

So läßt sich die Zahl 10 bspw. auch als `0b1010` schreiben - das Binärsystem ist hier gekennzeichnet durch das führende `0b`.

Folgende Präfixe können für die Zahlensysteme *Binär*, *Oktal*, *Dezimal* und *Hexadezimal* genutzt werden:

- **Binärsystem** (Basis 2): `0b` bzw. `0B`
- **Oktalsystem** (Basis 8): `0`
- **Dezimalsystem** (Basis 10): `1` ... `9`
- **Hexadezimalsystem** (Basis 16) `0x` bzw. `0X`

1.4.1 Umrechnung

Umrechnen von Werten aus den Zahlensystemen von bzw. in das Dezimalsystem folgt einem einfachen Schema, das zunächst anhand des Binärsystems erklärt wird.

Jedes Bit repräsentiert eine Position - dabei hat das Bit, das ganz rechts steht, die Position 0[8] (*niedrigster Stellenwert*: *least significant bit, lsb*), das Bit, das ganz links steht, die Position $n-1$ (*höchsterStellenwert*: *most significant bit, msb*, bei n Bits pro Zahl).
Die Position i eines Bits repräsentiert einen Exponenten zur Basis $B_v = 2$ - jedes Bit, das an der Stelle i durch eine 1 repräsentiert wird, wird entsprechend 2^i umgerechnet und mit allen solchen Werten innerhalb der Bitfolge addiert (s. Tabelle 1.3):

Bitfolge	1	0	1	0
Position	3	2	1	0
Umrechnung	$2^3 = 8$	-	$2^1 = 2$	-

Tabelle 1.3: Umrechnen von *0b1010* in das Dezimalsystem

Um eine Zahl x zur Basis $B_v = 10$ in das Binärsystem umzurechnen ($B_z = 2$), teilt man die Zahl durch 2, notiert den Rest (0 oder 1) und führt die Rechenschritte mit dem abgerundeten Quotienten solange fort, bis der Wert 0 erreicht ist. Das Ergebnis wird dann von rechts (erste Operation) nach links (letzte Operation) notiert:

1. $\lfloor \frac{10}{2} \rfloor = 5,\ 10\ mod\ 2 = 0$
2. $\lfloor \frac{5}{2} \rfloor = 2,\ 5\ mod\ 2 = 1$
3. $\lfloor \frac{2}{2} \rfloor = 1,\ 2\ mod\ 2 = 0$
4. $\lfloor \frac{1}{2} \rfloor = 0,\ 1\ mod\ 2 = 1$

   $\rightarrow 1010$

Das Schema $\lfloor \frac{x_{B_v}}{B_z} \rfloor = x_{next},\ x_{B_v}\ mod\ B_z = r_0 \ldots$ läßt sich so ohne weiteres auf das Oktalsystem bzw. Hexadezimalsystem übertragen, was anhand der Zahl $101_{B_v=10}$ demonstriert wird.

Oktalsystem

Sei $B_z = 8$.

1. $\lfloor \frac{101}{8} \rfloor = 12,\ 101\ mod\ 8 = 5$
2. $\lfloor \frac{12}{8} \rfloor = 1,\ 12\ mod\ 8 = 4$
3. $\lfloor \frac{1}{8} \rfloor = 0,\ 1\ mod\ 8 = 1$

[8] bei LSb_0, was auch im folgenden verwendet wird (s. a. "LSb_0-Bitnummerierung":`https://de.wikipedia.org/wiki/Bitwertigkeit#LSb0-Bitnummerierung` - abgerufen 15.03.2024)

$\rightarrow 145$

Sei nun $B_v = 8$ und $B_z = 10$ mit $x = 145$:

$$1 * 8^2 + 4 * 8^1 + 5 * 8^0 = 64 + 32 + 5 = 101 \tag{1.1}$$

Hexadezimalsystem

Sei $B_z = 16$.

1. $\lfloor \frac{101}{16} \rfloor = 6$, $101 \ mod \ 16 = 5$
2. $\lfloor \frac{6}{16} \rfloor = 0$, $6 \ mod \ 16 = 6$
   $\rightarrow 65$

Sei nun $B_v = 16$ und $B_z = 16$ mit $x = 65$:

$$6 * 16^1 + 5 * 16^0 = 96 + 5 = 101 \tag{1.2}$$

1.5 Zweierkomplement zur Repräsentation von Ganzzahlen

Java verwendet das **Zweierkomplement** für die Codierung negativer und positiver *ganze* Zahlen (`byte`, `short`, `int`, `long`).
Für negative Zahlen ist das erste Bit[9] gesetzt, für positive Zahlen nicht.
Da somit das erste Bit für das Vorzeichen reserviert ist, lassen sich mit n Bits nur die Zahlen $[-2^{n-1}, 2^{n-1} - 1]$ darstellen.

Positiv zu Negativ

Der negative Wert einer Zahl wird ermittelt, indem

1. die Bitfolge invertiert wird [10]
2. +1 addiert wird

Beispiel anhand der Zahl 3 ($B_3 := 0_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 1_1\ 1_0$), für deren negativen Wert die Bitfolge ermittelt werden soll:

1. $\sim B_3 = 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 0_1\ 0_0$
2. $(\sim B_3) + 1 = 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 0_1\ 1_0$

[9] das *most significant bit*, also das Bit mit dem höchsten Stellenwert, steht ganz links in einer Bitfolge
[10] "invertiert" im Folgenden abgekürzt über den *Komplement-Operator* $\sim$

Negativ zu Positiv

Der positive Wert einer negativen Ganzzahl im Zweierkomplement (Bit mit dem höchsten Stellenwert = 1) wird ermittelt, indem

1. die Bitfolge invertiert wird
2. +1 addiert wird[11]

Beispiel anhand der Zahl -3 ($B_{-3} := 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 0_1\ 1_0$), für deren negativen Wert die Bitfolge ermittelt werden soll:

1. $\sim B_{-3} = 0_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 1_1\ 0_0$
2. $\sim (B_{-3}) + 1 = 0_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 1_1\ 1_0$

Alternativ:

1. $B_{-3} - 1 = 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 0_1\ 0_0)$
2. $\sim (B_{-3} - 1) = 0_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 1_1\ 1_0$

1.6 Bit-Shifting

Hinweis

Die in dem Abschnitt verwendeten Beispiele sollen das Prinzip von Bit-Verschiebungen verdeutlichen und nutzen der Einfachheit halber Werte mit einer Wortlänge von 8-bit.
Es wird nicht berücksichtigt, dass u.a. `byte`- und `short`-Werte zu `int` in Java *auto-promoted* werden[12].

Für das Verschieben von Bits existieren drei verschiedene Operationen in Java: $<<, >>, >>>$.
Alle diese Operatoren sind binäre Operatoren: Der erste Operand b ist das zu verschiebende Bit-Muster, der zweite Operand n ist eine ganze Zahl, die angibt, um wieviele Stellen verschoben werden soll.

Ist das Bit-Muster ein `int`-Wert, wird immer um max. 31 Stellen verschoben ($n \ mod\ 32$)[13]:

> If the promoted type of the left-hand operand is int, then only the five lowest-order bits of the right-hand operand are used as the shift distance. It is as if the right-hand operand were subjected to a bitwise logical AND operator [...] with the mask value 0x1f (0b11111). The shift distance actually used is therefore always in the range 0 to 31, inclusive. ("15.19. Shift Operators": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.19` - abgerufen 06.03.2024)

[11] alternativ: vor dem ersten Schritt 1 subtrahieren; die invertierte Bitfolge ist dann das Ergebnis
[12] vgl. "5.6. Numeric Contexts": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-5.html#jls-5.6` - abgerufen 06.03.2024
[13] im selben Kontext hierzu für *long*-Werte: 63 Stellen

1.6.1 Signed Right Shift Operator >>

Das Vorzeichen der durch das Bit-Muster repräsentierten Zahl ändert sich bei der Verwendung des **signed right shift Operators** nicht. Ist das *most significant bit* eine 1, wird von links eine 1 an diese Position nachgeschoben, sonst eine 0.

Beispiel:
Das Bit-Muster für `byte b = -128` soll um zwei Positionen nach rechts verschoben werden:

$b >> 2$
1. $1_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 0_1\ 0_0$
2. $>> \rightarrow 1_7\ 1_6\ 0_5\ 0_4\ 0_3\ 0_2\ 0_1\ 0_0$
3. $>> \rightarrow 1_7\ 1_6\ 1_5\ 0_4\ 0_3\ 0_2\ 0_1\ 0_0$
   (...invertieren, +1 addieren ergibt den Absolutwert der negativen Zahl ...)
   $\rightarrow -32$

1.6.2 Signed Left Shift Operator <<

Das Vorzeichen der durch das Bit-Muster repräsentierten Zahl ändert sich bei der Verwendung des **signed right shift Operators**, falls an der Stelle $msb - 1$ eine 1 steht und an der Stelle msb eine 0[14].

Beispiel:
Das Bit-Muster für `byte b = 127` soll um eine Position nach links verschoben werden:

$b << 1$
1. $0_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 1_1\ 1_0$
2. $<< \rightarrow 1_7\ 1_6\ 1_5\ 1_4\ 1_3\ 1_2\ 1_1\ 0_0$
   (...invertieren, +1 addieren ergibt den Absolutwert der negativen Zahl ...)
   $\rightarrow -2$

1.6.3 Unsigned Right Shift Operator >>>

Das Vorzeichen der durch das Bit-Muster repräsentierten Zahl ändert sich bei der Verwendung des **unsigned right shift Operators**, falls an der Stelle msb eine 1 steht - die Bitfolge wird von links mit n Nullen aufgefüllt.

Beispiel:
Das Bit-Muster für `byte b = -125` soll unter Verwendung des **unsigned right shift Operators** um eine Position nach rechts verschoben werden:

[14] bzw. umgekehrt. msb = *most significant bit*

$b >>> 1$

1. $1_7\ 0_6\ 0_5\ 0_4\ 0_3\ 0_2\ 1_1\ 1_0$
2. $>>> \rightarrow 0_7\ 1_6\ 0_5\ 0_4\ 0_3\ 0_2\ 0_1\ 1_0$
   $\rightarrow 65$

1.7 Typumwandlung

Numerische Datentypen werden in Java *implizit* oder *explizit* konvertiert.

Zu den numerischen Datentypen[15] gehören

- `byte`
- `short`
- `int`
- `long`
- `char`
- `float`
- `double`

Eine *implizite* Konvertierung erfolgt, wenn eine Variable eines Datentyps mit einer niedrigeren Präzision einer Variablen eines Datentyps einer höheren Präzision zugewiesen wird:

```
int i = 32;
long l = i;

double d = 32.0f;

// das folgende funktioniert, weil der Compiler
// 'd' auswertet und feststellt, dass es im Wertebereich
// von short liegt
// short s = '\u9000'; würde hingegen vom Compiler
// abgewiesen
short s = 'd';
```

Eine *explizite* Konvertierung muss erfolgen, wenn eine Zuweisung von einem Datentyp mit einer höheren Auflösung zu einem Datentyp mit niedriger Ausflösung erfolgen soll.
Hierbei muss explizit **gecasted** werden.

```
int i = (int)32L;

float f = (float)32.0;
```

[15] s. "Chapter 4. Types, Values, and Variables": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.2.5` - abgerufen 07.03.2024

```
short s = -65_536;
char c = (char)s;
```

Grundsätzlich können alle Datentypen, die Zahlen repräsentieren, untereinander zugewiesen werden.
Hierzu zählt auch der Datentyp `char`, der einen numerischen Wertebereich von $[0, 65.535]$ $(2^{16} - 1)$ abdeckt.
Eine Konvertierung zwischen `boolean` und nuemrischen Werten ist weder mit explizitem noch implizitem casten möglich[16]; allerdings kann das Ergebnis eines Ausdrucks als boolescher Wert gespeichert werden:

```
int i = 1;
boolean b = i!=0;
```

1.7.1 Auto-Promotion bei arithmetischen Operationen

Bei arithmetischen Operationen (`+`, `-`, ...) kann der Datentyp der Operanden auf den Datentyp mit der höheren Präzision erweitert werden.

Im folgenden Beispiel soll ein `short`- zu einem `int`-Wert addiert werden. Der Compiler lehnt die Anweisung mit dem Hinweis ab, dass der Ausdruck `(short)3 + i` vom Typ `int` ist. Für eine erfolgreiche Zuweisung müßte das Ergebnis erst zu `short` gecasted werden, da der Compiler an der Stelle nicht feststellen kann, ob das Ergebnis noch im Wertebereich von `short` liegt.

```
int i = 4;
short s = (short)3 + i; // Compilerfehler
```

Dasselbe gilt für arithmetische Operationen, bei denen der `char`-Typ involviert ist:

```
char a = 'A';
char z = 'Z';
// Compilerfehler:
// "incompatible types: possible lossy
// conversion from int to char"
// char az = a + z;

// funktioniert:
char az = (char)(a + z);
```

Dagegen funktioniert folgendes Beispiel, da der Compiler den Ausdruck während der Kompilierung berechnet[17], keinen Überlauf feststellt und das Ergebnis des Ausdrucks sicher dem `short`-Typ zuweisen kann:

[16] "A boolean value may be cast to type boolean, Boolean, or Object [...]. No other casts on type boolean are allowed." (s. `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.2.5` - abgerufen 07.02.2024)

[17] "Some expressions have a value that can be determined at compile time. These are constant expressions [...]." (s. `https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.2` - abgerufen 07.03.2024)

```
short s = 3 + 4;
```

Bei folgendem Code hingegen stellt der Compiler einen Überlauf fest und lehnt die Zuweisung ab[18]:

```
short s = 1 + 32767;
```

Typkonvertierung

- Ist ein Ausdruck vom Typ `double`, werden alle Ausdrücke zu `double` **erweitert**
- Ist ein Ausdruck vom Typ `float`, werden alle Ausdrücke zu `float` **erweitert**
- Ist ein Ausdruck vom Typ `long`, werden alle Ausdrücke zu `long` **erweitert**
- Ansonsten entscheidet der Kontext, wie konvertiert wird: In einem numerischen arithmetischen Kontext werden die Ausdrücke zu `int` erweitert. In anderen Kontexten kann der Compiler versuchen, den gültigen Wertebereich einer Operation zu ermitteln und ein *Narrowing* anstelle eines *Widening* durchführen[19].

Die Tabelle 1.4 fasst die Zuweisungskompatibilität einfacher Datentypen zusammen, bei denen kein explizites Casting erforderlich ist, eine kompaktere Übersicht findet sich in Tabelle 1.5.
Für weitere Informationen sei auf die Sprachspezifikationen, Abschnitt "5.1. Kinds of Conversion"[20], verwiesen.

1.8 Operatoren

1.8.1 Operatorrangfolge

Operatoren und deren Rangfolge sind in Tabelle 1.6 aufgeführt.
Tauchen mehrere Operatoren in einer Zeile auf, gilt: Die Auswertung der Operatoren findet von links nach rechts statt, es sei denn, es handelt sich um Zuweisungsoperatoren - diese werden von rechts nach links ausgewertet.

Anmerkung zum Postfix-Operator

Der durch den Postfix-Operator berechnete Wert ($+1$ / -1) steht erst nach Auswertung des Ausdrucks, in dem der Operator verwendet wurde, in der Variable zur Verfügung, wie folgendes Beispiel zeigt:

[18] Wertebereich *short*: $[-2^{15}, 2^{15} - 1]$
[19] s. a. "5.6. Numeric Contexts": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-5.html#jls-5.6` - abgerufen 15.03.2024
[20] `https://docs.oracle.com/javase/specs/jls/se21/html/jls-5.html#jls-5.1` - abgerufen 07.03.2024

	byte	short	char	int	long	float	double
byte	✓	-	-	-	-	-	-
short	✓	✓	-	-	-	-	-
char	-	-	✓	-	-	-	-
int	✓	✓	✓	✓	-	-	-
long	✓	✓	✓	✓	✓	-	-
float	✓	✓	✓	✓	✓	✓	-
double	✓	✓	✓	✓	✓	✓	✓

Tabelle 1.4: Zuweisungskompatibilität einfacher Datentypen. Ein ✓bedeutet, dass kein explizites casten von dem Datentyp (horizontale Spalte) zu dem Datentyp (vertikale Spalte) erforderlich ist.

von	**nach**
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Tabelle 1.5: Implizite Typumwandlung in Java in der kompakten Übersicht. (Quelle: in Anlehnung an [Ull23, 145, Tabelle 2.12])

```
int n = 5;
System.out.println(n++ - 1); // "4"
System.out.println(n); // "6"
```

Der Operator ändert nicht den Definitionsbereich der Variable, auf den er angewendet wird:

```
double = 3.42;
System.out.println(++d); // "4.42"

byte b = 127;
System.out.println(++b); // "-128"
```

Kategorie	Rang	Operatoren
Postfix	1	`expr++`, `expr--`
Unary	2	`++expr`, `--expr`, `+expr`, `-expr`, `~`, `!`
Multiplicative	3	`*`, `/`, `%`
Additive	4	`+`, `-`
Shift	5	`<<`, `>>`, `>>>`
Relational	6	`<`, `>`, `<=`, `>=`, `instanceof`
Equality	7	`==`, `!=`
Bitwise AND	8	`&`
Bitwise exclusive OR	9	`^`
Bitwise inclusive OR	10	`|`
Logical AND	11	`&&`
Logical OR	12	`||`
Ternary	13	`? :`
Assignment	14	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`, `>>>=`

Tabelle 1.6: Operatorrangfolge in Java. 1 entspricht dem höchsten Rang bei der Evaluierung, 14 dem niedrigsten. (Quelle: in Anlehnung an "The Java Tutorials - Operators": `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html` - abgerufen 11.03.2024)

1.8.2 Kurzschluss-Operatoren

Die mit den logischen Operatoren `&&` sowie `||` verwendeten Ausdrücke werden nur komplett ausgewertet, falls

- `&&`: Der linke Ausdruck `true` ergibt
- `||`: Der linke Ausdruck `false` ergibt

Das Prinzip lässt sich auf den **ternären Operator** `?:` übertragen[21].

1.8.3 Verbundoperatoren und implizite Typumwandlung

Werden Verbundoperationen bei Datentypen genutzt, bei denen ein größerer Datentyp einem kleineren zugewiesen wird, findet ein implizites Casting statt.

So muss bspw. für folgende Zuweisung *explizit* gecasted werden:

[21] s. "15.7.2. Evaluate Operands before Operation": `https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.7.2` - abgerufen 11.03.2024

```
int i = 32;
i = (int)50.0;
```

während in folgendem Beispiel implizit gecasted wird:

```
int i = 32;
i += 50.0; // 50
```

1.9 Schleifen

1.9.1 while / do-while

Bei einer `while`-Schleife wird *zuerst* die Schleifenbedingung überprüft - falls der enthaltene Ausdruck zu `true` evaluiert, wird das Statement ausgeführt.

Bei einer `do-while`-Schleife wird *zuerst* das Statement ausgeführt, und *danach* die Schleifenbedingung überprüft. Das Statement wird also immer mindestens einmal ausgeführt.

1.9.2 for-Schleife

Eine `for`-Schleife ist folgendermaßen aufgebaut[22]:

`for` ([`ForInit`] ; [`Expression`] ; [`ForUpdate`]) `Statement`

- `ForInit`, `Expression`, `ForUpdate` sind optional.
- ist `Expression` angegeben, **muss** der Ausdruck ein *boolescher* Ausdruck sein
- `ForInit` darf eine kommaseparierte Liste von lokalen Variablendeklarationen sein, oder eine kommaseparierte Liste von Ausdrucksanweisungen
- `ForUpdate` darf eine kommaseparierte Liste von Ausdrucksanweisungen sein
- `ForUpdate` wird *nach* einem Durchlauf der Schleife ausgeführt, sofern der boolesche Ausdruck (falls vorhanden) zu `true` ausgewertet wurde. Die Reihenfolge entspricht grob:
  - Initialisierung des Statements über `ForInit`
    1. Auswertung der `Expression` - falls `true`, zu **2**.
    2. Ausführen des `Statement`s
    3. Ausführen von `ForUpdate`
    4. falls keine andere Abbruchbedingung vorliegt; zurück zu **1**

Beispiele für gültige `for`-Schleifen:

[22] s. "14.14.1. The basic for Statement": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-14.html#jls-14.14.1` - abgerufen 12.03.2024

```
// endlosschleife, boolescher Ausdruck nicht angegeben
// i und a sind nur in dem Block der for-Schleife sichtbar
for (int i = 0, a = 1; ; i++) {
    System.out.println(a++);
}

int z = 0;
int a = 0;
for (System.out.println("Hello World!"),
     System.out.println("starte Schleife");
    z < 10;
    z++,
    System.out.println("z und a jetzt bei: " + z + "(z), " + a + "(a)")) {
    System.out.println("im Schleifenrumpf: " + z + "(z), " + a + "(a)");
    a++;
}
```

1.10 Kontrollstrukturen

1.10.1 Das switch Statement

Ist in dem `switch`-Block nach einer Anweisung kein `break`[23] angegeben, werden alle nachfolgenden Anweisungen des switch-Blocks ausgeführt, bis der Block verlassen wird:

> Each break statement terminates the enclosing switch statement. Control flow continues with the first statement following the switch block. The break statements are necessary because without them, statements in switch blocks fall through: All statements after the matching case label are executed in sequence, regardless of the expression of subsequent case labels, until a break statement is encountered. ("The switch Statement": `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html` - abgerufen 12.03.2024)

Im folgenden Beispiel wird das switch-Statement mit `n=2` aufgerufen. Es finden *drei* Ausgaben statt - auch die durch das `default`-Label eingeführte Anweisung wird ausgeführt.

```
switch (n) {
    case 2:
        System.out.println("case 2");
```

[23] bzw. *return*

```
        case 4:
            System.out.println("case 4");

        default:
            System.out.println("default");
    }
```

Es ist egal, an welcher Stelle das `default`-Label in dem switch-Block steht. Wenn keine `case`-Label mit entsprechendem `caseconstant` dem ausgewerteten Ausdruck entspricht, wird das durch die `default`-Label eingeführte Anweisung aufgerufen.

Im folgenden Beispiel wird `n=5` übergeben, es finden *drei* Ausgaben statt:

```
    switch (n) {
        default:
            System.out.println("default");

        case 2:
            System.out.println("case 2");

        case 4:
            System.out.println("case 4");
    }
```

1.11 Namenskonventionen

- Für eine als `final` deklarierte lokale Variable gelten die üblichen Namenskonventionen, wie für andere lokale Variablen auch.

Für weitere Informationen bzgl. der Namenskonventionen, siehe "The Java™ Tutorials - Variables": `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html`[24].

[24] abgerufen 12.03.2024

2

OOP

2.1 Grundlagen

2.1.1 Methodensignatur

In Java gehört zu der Methodensignatur der Methodenname sowie die formale Parameterliste[1].

2.1.2 Überschreiben

Eine Methode wird in einer Unterklasse *überschrieben*, wenn Signatur und Rückgabetyp der Methode der Unterklasse mit der Methode der Oberklasse übereinstimmen[2].

2.1.3 Überladen

Eine Methode wird *überladen*, wenn eine Methode mit gleichem Namen bereitgestellt wird, aber eine unterschiedliche Signatur aufweist.
Der Rückgabetyp darf abweichen:

> There is no required relationship between the return types or between the throws clauses of two methods with the same name, unless their signatures are override-equivalent. (Java Language Specification - 8.4.9. Overloading[3])

Unter einem *mehrdeutigen* Methodenaufruf versteht man einen Aufruf zu einer überladenen Methode, die der Compiler nicht eindeutig zuordnen kann[4].
Im Folgenden Beispiel wird die Methode `foo(x: int, y: double):void` durch `foo(x: double, y: int):void` überladen:

[1] Java Language Specification - 8.4.2. Method Signature: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.2` - abgerufen 07.03.2024
[2] stimmt nur die Signatur überein, kommt es zu einem Compilerfehler
[3] `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.9` - abgerufen 07.03.2024
[4] s. "15.12.2.5. Choosing the Most Specific Method": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.12.2.5` - abgerufen 07.03.2024

```
public void foo(int x, double y) {
    ...
}

public void foo(double x, int y) {
    ...
}
```

Den Aufruf

```
foo(1, 2);
```

quittiert der Compiler mit einer Fehlermeldung "Ambiguous method call.".
An dieser Stelle ist explizites casten nötig, damit für den Compiler klar ist, welche Methode aufgerufen werden soll:

```
foo(1, (double)2);
```

2.2 Vererbung

- Wird in einer Superklasse eine *Objektmethode* definiert, die mit gleicher Signatur in der Unterklasse implementiert wird, dann **überschreibt** die Methode der Unterklasse die der Oberklasse
- Wird in einer Superklasse eine *Klassenmethode* definiert, die mit gleicher Signatur in der Unterklasse implementiert wird (ebenfalls als `static` deklariert), dann **überdeckt** die Methode der Unterklasse die der Oberklasse[5].
- Statische Methoden / Eigenschaften werden vererbt, können aber in Unterklassen *nicht* überschrieben, sondern nur überdeckt werden

```
class A {

    static int snafu = 42;

    static int baz = 0;

    public void foo() {}

    public static void bar() {
        System.out.println("in A");
    }
}

class B extends A {
```

[5] s. "Overriding and Hiding Methods": `https://docs.oracle.com/javase/tutorial/java/IandI/override.html` - abgerufen 12.03.2024

```
        /**
         * überdeckt A.baz
         */
        static int baz = 2;

        /**
         * überschreibt foo()
         */
        public void foo() {

            /**
             * Ruft implizit A.snafu auf
             */
            System.out.println(snafu);

        }

        /**
         * überdeckt bar()
         */
        public static void bar() {
            System.out.println("in A");
        }
    }
```

2.3 Konstruktoren

Ein **allgemeiner Konstruktor** ist ein Konstruktor mit einer formalen Parameterliste (vgl. [Ull23, 424]).

Der **Standard-Konstruktor** ist ein *parameterloser* Konstruktor (vgl. [Ull23, 423]).

Konstruktoren in Java sind nicht mit Objektmethoden gleichzusetzen, und zählen *nicht* zu den vererbbaren Eigenschaften einer Klasse.

Wenn für eine Klasse kein Konstruktor definiert wurde, stellt der Compiler einen **Standard-Konstruktor** zur Verfügung[6].

```
class A {

    // implizit deklarierter default constructor
```

[6] The Java Language Specification - 8.8.9. Default Constructor: https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.8.9

```
    // public A() {
    // }

    public static void main(String[] args) {
        A a = new A(); // ruft den implizit deklarierten Standard-Konstruktor
                       // auf
    }

}
```

Wenn ein *explizit* deklarierter Konstruktor vorhanden ist, stellt der Compiler keinen Standard-Konstruktor zur Verfügung.

Im folgenden Beispiel führt `new A()` zu einem Compiler-Fehler, da kein parameterloser Konstruktor explizit oder implizit für `A` deklariert ist.

```
class A {

    int a;

    // explizit deklarierter Konstruktor
    // unterbindet die Bereitstellung eines implizit
    // deklarierten Standard-Konstruktor
    public A(int value) {
        a = value;
    }

    public static void main(String[] args) {
        A a1 = new A(i); // ruft den explizit deklarierten Konstruktor auf
        A a2 = new A();  // fuehrt zu einem Compiler-Fehler
    }

}
```

Wenn ein Konstruktor nicht *explizit* einen Konstruktor seiner Elternklasse aufruft, ruft er *implizit* `super()` auf.
Wenn die Elternklasse in solchen Fällen keinen parameterlosen Konstruktor deklariert hat, wird ein Compiler-Fehler erzeugt.
Im folgenden Beispiel erweitert `B` die Klasse `A`. `B` hat einen Konstruktor deklariert, dessen formale Parameterliste `int` ist. Dieser Konstruktor ruft implizit `super()` auf:

```
class A {

}
```

```
class B {

    int val;

    public B (int a) {
        // implizit aufgerufen:
        // super();
        val = a;
    }

    public static void main(String[] args) {
        B b = new B(1);
    }

}
```

Ergänzend sei daran erinnert, dass jede Klasse in Java direkt oder indirekt von `java.lang.Object` erbt[7]: Der Konstruktor der Klasse `java.lang.Object` ist also in einer Konstruktor-Aufrufkette enthalten.

Im nächsten Beispiel erbt `C` von `B`. Ein Standard-Konstruktor wird implizit für `C` deklariert. Allerdings führt der Versuch, ein Objekt vom Typ `C` zu erzeugen, zu einem Compiler-Fehler, da `B` keinen parameterlosen Konstruktor - den Standardkonstruktor - deklariert:

```
class C extends B {
    public static void main(String[] args) {
        C c = new C(); // Compiler-Fehler
    }
}
```

Damit ein Objekt vom Typ `C` erzeugt werden kann, benötigt `C` einen Standardkonstruktor mit einem expliziten Aufruf des Konstruktors von `B` - da `B` selber keinen Standardkonstruktor deklariert hat, muss der Konstruktor mit der Signatur `B(int)` aufgerufen werden:

```
class C extends B {

    public C() {
       super(2); // ruft den Konstruktor von B mit dem Argument 2 auf
    }

    public static void main(String[] args) {
        C c = new C();
```

[7] The Java Tutorials - Object as a Superclass: https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

```
    }

}
```

Das zeigt, dass in Java bei einem explizit deklarierten Konstruktor als *erstes* der Konstruktor der Elternklasse aufgerufen wird (implizit durch den Compiler) bzw. aufgerufen werden muss (falls explizit implementiert)[8].

Es folgen ergänzende Beispiele.
Details sind den Quelltextkommentaren zu entnehmen.

```
class A {
    // impliziter Standard-Konstruktor: vorhanden

    // impliziter Standard-Konstruktor: ruft Konstruktor von java.lang.Object
    // auf
}


class B extends A {

    // impliziter Standard-Konstruktor: vorhanden

    // impliziter Standard-Konstruktor: ruft Konstruktor von A auf
}


class C {

    // impliziter Standard-Konstruktor: nicht vorhanden

    public C() {
        // impliziter Aufruf von java.lang.Object's Konstruktor
        System.out.println("C created.");
    }
}


class D extends C {

    // impliziter Standard-Konstruktor: vorhanden

    // impliziter Standard-Konstruktor: ruft Konstruktor von C auf
```

[8] es kann auch zunächst durch *this()* ein anderer Konstruktor aufgerufen werden; s.a. "The Java Language Specification - 8.8.7. Constructor Body": https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.8.7 - abgerufen 12.03.2024

```
}


class E extends D {

    // impliziter Standard-Konstruktor: nicht vorhanden

    public E() {
        // impliziter Aufruf von D's Konstruktor
        System.out.println("E created.");
    }
}


class F extends E {

    // impliziter Standard-Konstruktor: nicht vorhanden

    public F(int x) {
        // impliziter Aufruf von E's Standardkonstruktor
        System.out.println("F created");
    }
}


class G extends F {

    // impliziter Standard-Konstruktor: nicht vorhanden

    public G(int x) {
        super(x); // expliziter Aufruf von F's Konstruktor

        // Das Auskommentieren der o.a. Anweisung fuehrt zu einem impliziten
        // Aufruf von 'super()', was einen Compiler-Fehler produziert:
        // Da kein Standardkonstruktor in F deklariert ist, muss dem
        // Konstruktor explizit mitgeteilt werden, welcher Konstruktor der
        // Elternklasse aufgerufen werden soll.
        System.out.println("G created");
    }
}
```

2.4 Objektinitialisierung

Die Sprachspezifikationen beschreiben die Objekterzeugung und die Reihenfolge der damit verbundenen Konstruktor-Aufrufe und Attribut-Initialiserung vereinfacht wie folgt[9]:

Führe *rekursiv* aus:

1. Der Konstruktor wird mit den Argumenten aufgerufen
2. Ruft der Konstruktor über `this` einen anderen Konstruktor auf, gehe zu Schritt 1
3. Ruft der Konstruktor implizit oder explizit einen Konstruktor der Elternklasse auf, gehe zu Schritt 1
4. Die Objektattribute werden initialisiert
5. Die übrigen Anweisungen innerhalb des Konstruktors werden ausgeführt

Im Folgenden ein Beispiel für die Initialisierungsreihenfolge durch den Aufruf `new C()`.

```
class Dummy {
    public Dummy(String from) {
        System.out.println("Dummy for " + from);
    }
}

class A {
    Dummy d = new Dummy("A");
    public A() {
        System.out.println("A");
    }
}

class B extends A {
    public B() {
        System.out.println("B");
    }
}

class C extends B {
    Dummy d = new Dummy("C");
    public C() {
        System.out.println("C");
```

[9] s. "12.5. Creation of New Class Instances": https://docs.oracle.com/javase/specs/jls/se21/html/jls-12.html#jls-12.5 - abgerufen 13.03.2024

```
        }
    }
```

Die Ausgabe lautet:

```
Dummy for A
A
B
Dummy for C
C
```

2.5 Zugriffsmodifizierer

Tabelle 2.1 listet die verfügbaren Zugriffsmodifizierer in Java auf[10]

Modifizierer	Klasse	Package	Unterklasse	Welt	UML
`public`	✓	✓	✓	✓	`+`
`protected`	✓	✓	✓	–	`#`
kein Modifizierer	✓	✓	–	–	`~`
`private`	✓	–	–	–	`-`

Tabelle 2.1: Zugriffsmodifizierer und ihre Sichtbarkeiten. Die UML-Notation bzgl. der Sichtbarkeiten wurde im Skript (Teil 1), Abschnitt 5.4.3 und 5.5 *nicht* verwendet und ist der Vollständigkeit halber aufgelistet (s. a. [Ull23, 404, Tabelle 6.4] sowie [Oes05, 251 ff.]).

Wenn kein expliziter Zugriffsmodifizierer angegeben wurde, ist die Sichtbarkeit implizit *package-private* (Zeile 3 in Tabelle 2.1).

Die Sichtbarkeit einer überschrieben Methode darf nicht reduziert werden. Erweitern ist dagegen möglich:

```
class A {
    protected function foo() {
    }

    public function bar() {
    }
}

class B extends A {
    public function foo() { // Sichtbarkeit von foo()
```

[10] "The Java™ Tutorials - Controlling Access to Members of a Class": `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html` - abgerufen 08.03.2024

```
    }                          // erweitert

    protected function bar() { // Compilerfehler:
    }                          // "Attempting to assign weaker
                               // access privileges"

}
```

2.6 Packages

Das Package-Konzept in Java repräsentiert kein hierarchisches Modell:

> [...], packages appear to be hierarchical, but they are not.[11]

Aus diesem Grund kennt Java so etwas wie "innere Pakete" nicht.

Verdeutlicht wird dies anhand des Beispiels von `import`-Deklarationen, die Wildcards ("*") in Verbindung mit einem qualifizierten Namen nutzen[12].
So können alle öffentlichen Klassen aus dem Paket `java.awt` in Typdeklarationen einer Übersetzungseinheit[13] über ihren Namen referenziert werden, wenn wie folgt importiert wird:

```
import java.awt.*;
```

Dies schließt allerdings nicht die Klassen ein, die in Paketen liegen, die ihrem qualifizierten Namen nach scheinbar "in" `java.awt` liegen, wie bspw. `java.awt.color`. Diese müssen separat importiert werden:

```
import java.awt.*;
import java.awt.color.*;
```

2.7 Typkompatibilität

Für die folgenden Beispiele sei in Java folgende Vererbungsbeziehung gegeben:

```
class A {
    public int x = 0;
    public void inA() {
        ...
```

[11] "The Java™ Tutorials - Apparent Hierarchies of Packages": `https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html` - abgerifen 08.03.2024
[12] Eine sog. "Type-Import-on-Demand Declaration". Siehe "Java Language Specification - 7.5.2. Type-Import-on-Demand Declarations": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html#jls-7.5.2` - abgerufen 08.03.2024
[13] "Compilation Unit". Siehe hierzu "Java Language Specification - 7.3. Compilation Units": `https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.3` - abgerufen 08.03.2024

```
    }
    public void whoAmI() {
        System.out.println("A");
    }
}

class B extends A {
    public int x = 1;
    public void inB() {
    }
    public void whoAmI() {
        System.out.println("B");
    }
}
```

- Klasse `A` ist ein `java.lang.Object`
- Klasse `B` ist ein `A`
- Klasse `B` ist ein `java.lang.Object`

2.7.1 Ersetzbarkeitsprinzip

Immer dann, wenn ein bestimmter Typ gefordert wird, ist auch sein Untertyp erlaubt (vgl. [Ull23, 466])[14].

Der folgenden Methode kann ohne weiteres ein Objekt vom Typ `B` übergeben werden: Da `B` ein `A` ist, hat es automatisch seine Methode `inA()` geerbt:

```
class C {
    public void foo(A a) {
        a.inA();
    }
}
C c = new C();
B b = new B();
c.foo(b);
```

2.7.2 Typumwandlung

Wie *Ullenboom* in [Ull23, 467 f.] zeigt, ist für den Compiler der *Referenztyp* wesentlich, für die Laufzeitumgebung hingegen der *Objekttyp* (s. Abbildung 2.1).

So kompiliert folgende Ausdrucksanweisung problemlos:

```
A a = new B();
```

[14] Der Kern des *Liskovschen Substitutionsprinzip*, das in Abschnitt 2.7.3 näher beleuchtet wird

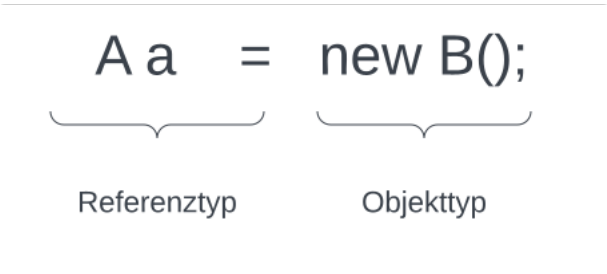

Abb. 2.1: Referenztyp und Objekttyp in Java. Der Compiler berücksichtigt vor allem den Referenztyp. (Quelle: in Anlehnung an [Ull23, 467 f., "Begrifflichkeit"])

Allerdings kann im folgenden Code nur auf Eigenschaften der Klasse `A` über die lokale Variable `a` zugegriffen werden - dass eigentlich ein Objekt vom Typ `B` in `a` steckt, interessiert den Compiler nicht:

```
A a = new B();
a.inB(); // Fehler "Cannot resolve method 'inB' in 'A'"
```

Obwohl der Compiler die Methode `inB()` nicht auflösen kann, löst die Laufzeitumgebung problemlos den Typen von `a` zu `B` auf:

```
A a = new B();
System.out.println(a instanceof B); // true
```

Auch, wenn der *Compiler* nur Aufrufe zu Methoden der Klasse `A` auflösen kann, werden Methoden, die in `B` überschrieben werden, zur *Laufzeit* auf `b` aufgerufen:

```
A a = new B();
System.out.println(a instanceof B); // true
a.whoAmI();// "B"
```

Felder werden überdeckt

Felder werden in Java *nicht* überschrieben.
Enthält eine Unterklasse ein Feld gleichen Namens, überdeckt dieses Feld das Feld der Superklasse.
Ein Zugriff auf das Feld der Superklasse ist von der Unterklasse nur über `super.[feldname]` möglich, oder über explizites Casting: `((Superklasse)this).[feldname]`.

„ Bei Methodenaufrufen bindet das Laufzeitsystem immer dynamisch; bei Zugriffen auf Objektvariablen ist das nicht so: Hier bestimmt der Compiler, von welcher Klasse die Objektvariable genommen werden soll. “ ([Ull23, 505])

Aus diesem Grund liefert das folgende Programm als Ausgabe zunächst 0. Erst ein explizites casten ermöglicht den Zugriff auf die in der Unterklasse überdeckte Eigenschaft, worauf die Ausgabe 1 erfolgt:

```
A a = new B();
System.out.println(a.x); // "0", weil in B überdeckt.
System.out.println(((B)a).x); // "1"
```

Um dennoch auf `inB()` zugreifen zu können, kann `a` explizit nach `B` gecasted werden.

I.d.R. wird der Compiler aufgrund erkannter Typinkompatibilität das Kompilieren verhindern; werden durch den Compiler diese Inkompatibilitäten aber nicht erkannt, wird die Laufzeitumgebung veranlasst, die Operation durchzuführen - gelingt das nicht, wird eine `ClassCastException`[15] geworfen, die folgerichtig eine Exception vom Typ `RuntimeException` ist[16]:

```
Object a = new B();
String s = (String)a; // ClassCastException
```

Die eben erwähnte Typumwandlung funktioniert letztendlich durch explizites casting wie folgt:

```
A a = new B();
((B)a).inB();
```

Eine implizite Typumwandlung kann durch eine explizite ersetzt werden[17]:

```
B b = new B();
A a = (A)b;
Object o = (Object)a;
```

[15] “Class ClassCastException“: `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/ClassCastException.html` - abgerufen 07.03.2024
[16] Abschnitt 3 widmet sich etwas ausführlicher den Eigenschaften von Laufzeit-Ausnahmen.
[17] je nach Entwicklungsumgebung wird an dieser Stelle der Hinweis gegeben, dass dies bereits durch den Compiler geschieht.

Explizites casting funktioniert nicht zum spezielleren Typ:

```
A a = new A();
B b = (B)a; // ClassCastException
```

2.7.3 Kovarianz

Sobald es sich bei dem Rückgabetyp einer Methode um einen *Referenztypen* handelt, muss der Rückgabetyp der Methode in der Unterklasse folgende Bedingung erfüllen[18]:

- Der Rückgabetyp der Methode in der Unterklasse muss **kovariant** zu dem Rückgabetyp der überschriebenen Methode sein[19]

Kovariant bedeutet, dass der Rückgabetyp in der Methode der Unterklasse nicht allgemeiner sein darf als der Rückgabetyp der Methode der Oberklasse.
Sowohl Richtung der Vererbungshierarchie der als auch die Richtung der Typhierarchie stimmen überein:

- Unterklasse ← Oberklasse[20]
- Rückgabetyp Methode Unterklasse ← Rückgabetyp Methode Oberklasse[21]

Zur Verdeutlichung können die in dem Kurs bereits verwendeten geometrischen Figuren betrachtet werden:

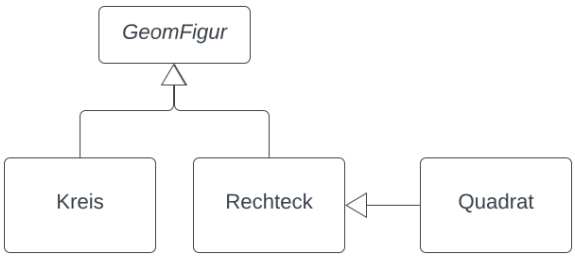

Abb. 2.2: Vererbungshierarchie der Klasse *GeomFigur* und ihrer Subklassen.

Eine Klasse `FigurenFabrik` sei folgendermaßen gegeben:

```
class FigurenFabrik {

    /**
     * Erzeugt ein neues Quadrat mit der spezifizierten Seitenlänge.
     *
```

[18] vgl. "Java Language Specification - 8.4.8.3. Requirements in Overriding and Hiding": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.8.3`
[19] vgl. "Java Language Specification - 8.4.5. Method Result": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.5`
[20] "←": Richtung d. Vererbungshierarchie
[21] "←": Richtung d. Typhierarchie

```
     * @param seitenLaenge die spezifizierte Seitenlänge.
     */
    public Quadrat erzeugeQuadrat(double seitenLaenge) {
        return new Quadrat(seitenLaenge);
    }
}
```

Eine Klasse, die von `FigurenFabrik` erbt und `erzeugeQuadrat` überschreiben möchte, muss sicherstellen, dass das von der Methode zurückgelieferte Objekt auch tatsächlich ein `Quadrat` ist.

Hierzu darf der Rückgabetyp nicht allgemeiner sein. 'Allgemeiner' bedeutet in diesem Fall: Typ `Rechteck` oder `Figur`.

Ansonsten würden Zugriffe auf (vererbbare) spezifische Eigenschaften der Klasse `Quadrat` mit den zurückgelieferten Objekten zu Fehlern führen. Der Compiler verhindert dies bereits:

```
class KaputteFigurenFabrik extends FigurenFabrik {

    public Rechteck erzeugeQuadrat(double seitenLaenge) {
        return new Rechteck(seitenLaenge, seitenLaenge*2);
    }
}

public class FigurenMacher {

    public static void main(String[] args) {
        FigurenFabrik f = new FigurenFabrik();

        // bevor es zum Programmabsturz waehrend der Laufzeit
        // aufgrund eines Aufrufs einer nur in Quadrat bekannten
        // Methode kommt, bricht der Compiler den Uebersetzungsvorgang
        // bereits mit einem Fehler ab
        f.erzeugeQuadrat().spezifischeMethodeAusQuadrat();
    }

}
```

Der vom Compiler ausgegebene Fehler weist auf die Inkompatibilität der Rückgabetypen hin:

```
return type Rechteck is not compatible with Quadrat.
```

Für **Kovarianz** (und in diesem Zusammenhang **Kontravarianz**) und formalere Herleitungen sei auf *Das Liskovsche Substitutionsprinzip*[22] ([**Lis87**]) sowie den ausführlichen Beitrag im englischsprachigen Wikipedia [23] verwiesen.

2.8 UML

2.8.1 Assoziation

Eine **Assoziation** ist eine Beziehung zwischen mehreren Objekten (s. Abbildung 2.3).

Abb. 2.3: Beispiel für eine *Assoziation* (Quelle: eigene)

2.8.2 Aggregation

Eine **Aggregation** stellt eine Teil-Ganzes-Beziehung dar.
Eine offene Raute weist hier hierbei auf die Aggregat-Wurzel hin, die Linie endet bei den verschiedenen Teilen, aus denen das Aggregat besteht (s. Abbildung 2.4).

Abb. 2.4: Beispiel für eine *Aggregation* (Quelle: eigene)

2.8.3 Komposition

Eine **Komposition** stellt wie eine Aggregation eine Teil-Ganzes-Beziehung dar, wobei eine Komposition aus *existenzabhängigen* Teilen besteht .
Eine gefüllte Raute weist hier hierbei auf das Ganze hin, die Linie endet bei den verschiedenen Teilen, aus denen das Kompositum besteht (s. Abbildung 2.5), und die auch gelöscht werden, falls das Ganze gelöscht wird.

[22] "Wikipedia - Liskov substitution principle": `https://en.wikipedia.org/wiki/Liskov_substitution_principle`
[23] "Wikipedia - Covariance and Contravariance": `https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)`

Abb. 2.5: Beispiel für eine *Komposition* (Quelle: eigene)

3

Exceptions

In Java findet eine Unterscheidung zwischen **checked** (*geprüften*) und **unchecked** (*ungeprüften*) Exceptions statt.

3.1 Checked Exceptions

Als eine **checked Exception** wird jede Exception bezeichnet, die im Programmcode behandelt werden muss.

> Jede Exception, die nicht von `Runtime Exception` ableitet, gilt als eine **checked Exception**.

Eine *checked* Exception wird wird in der `throws`-Klausel einer Methode deklariert, wenn sie nicht innerhalb der Methode abgefangen wird.

Als Beispiel sei der Aufruf der Methode `createNewFile():boolean` der Klasse `File`[1] gegeben, die eine `java.io.IOException`[2] werfen kann:

```
private void createFile() {
    File f = new File("./datei.txt");
    f.createFile();
}
```

Da es sich bei einer IOException um eine *checked* Exception handelt, muss diese Ausnahme im Code behandelt werden, oder sie muss in der `throws`-Klausel der *aufrufenden* Methode deklariert werden, ansonsten wird der Programmcode vom Compiler mit einem Fehler abgewiesen:

[1] "Class File": `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/File.html` - abgerufen 17.2.2024

[2] "Class IOException": `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/io/IOException.html` - abgerufen 17.2.2024

```
java: unreported exception java.io.IOException;
must be caught or declared to be thrown
```

Damit der Compiler nicht mit einer Fehlermeldung abbricht, kann man den Code folgendermaßen ändern:

```
private void createFile() {
    File f = new File("./datei.txt");

    try {
        f.createFile();
    } catch (IOException e) {
        // Ausnahmebehandlung
    }
}
```

In dem Fall wird die Ausnahme in der Methode abgefangen, und sie steigt in der Anwendung nicht mehr nach oben.
Im Gegensatz hierzu wird im folgenden Beispiel in der `throws`-Klausel die Exception deklariert.
Dem Compiler wird damit mitgeteilt, dass bei der Verwendung der Methode eine `IOException` geworfen werden *kann*.
Die Ausnahme steigt also im Programmcode nach oben und muss dann von einer anderen Methode, die `createFile()` aufruft, abgefangen werden - oder wiederum in deren `throws`-Klausel als mögliche Ausnahme deklariert werden:

```
private void createFile() throws IOException {
    File f = new File("./datei.txt");
    f.createFile();
}
```

3.2 Unchecked Exceptions

Als **unchecked Exception** gilt jede Exception, die von `RuntimeException` abgeleitet ist.

> RuntimeException and its subclasses are *unchecked exceptions*. Unchecked exceptions do *not* need to be declared in a method or constructor's throws clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary. ("Class RuntimeException": `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/RuntimeException.html` - abgerufen 17.2.2024)

Ungeprüfte Exceptions können ebenfalls in der `throws`-Klausel der entsprechenden Methode deklariert werden, was aber - bis auf dokumentarische - keinerlei weitere Auswirkungen auf das Programm hat: Wird eine `RuntimeException` in

der throws-Klausel deklariert, muss sie nicht in aufrufenden Methoden behandelt werden.

```
private void runtimeThrow() throws RuntimeException {
    throw new RuntimeException();
}

private void runtimeThrowCaller() {
    runtimeThrow();
}
```

3.3 Vererbung und Schnittstellen

3.3.1 Schnittstellenbeziehungen

Wenn eine Schnittstellenmethode eine throws-Klausel deklariert (mit einer oder mehreren *checked* Exceptions), muss diese Ausnahme bei der Implementierung der Methode nicht berücksichtigt werden, wenn die Implementierung diese Exception nicht werfen kann.

```
interface Exceptionable {
    void createFile() throws IOException;
}
class ExceptionImpl implements Exceptionable {

    @Override
    public void createFile() {
        // Implementierung benutzt keine
        // Methode, die eine IOException werfen kann
    }
}
```

Wird bei der Implementierung die throws-Klausel mit angegeben, oder wird eine Implementierung verwendet, die eine *checked* Exception wirft, dann muss die Ausnahmebehandlung kompatibel zu der throws-Klausel der abstrakten Methode der Schnittstelle sein.
Die throws-Klausel muss dann *dieselbe* oder eine von der über die abstrakte Methode deklarierten Ausnahme abgeleiteten Klasse werfen[3]:

```
interface Exceptionable {
    void createClient() throws RemoteException;
}
```

[3] "For every checked exception type listed in the throws clause of m2, that same exception class or one of its supertypes must occur in the erasure [..] of the throws clause of m1; otherwise, a compile-time error occurs." https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3 - abgerufen 17.2.2024

```
class ExceptionImpl implements Exceptionable {

    @Override
    public void createClient() throws UnknownHostException {
        //...
    }
}
```

Hier wird `createClient()` so implementiert, dass eine Ausnahme vom Typ `java.rmi.UnknownHostException` geworfen wird.
Diese Ausnahme ist von `java.rmi.RemoteException` abgeleitet.

Der Schnittstellenvertrag kann im Anwendungscode dann wie folgt erfüllt werden:

```
public void foo(Exceptionable rmiCreator) {
    try {
        rmiCreator.createClient();
    } catch (RemoteException e) {
        // Ausnahmebehandlung
    }
}
```

Umgekehrt kann eine implementierende Klasse die zu werfende Ausnahme nicht erweitern.
Im obigen Code kann bspw. nicht einfach die geworfene Ausnahme zu `IOException` verallgemeinert werden, auch wenn `IOException` die Elternklasse von `RemoteException` ist:

```
interface Exceptionable {
    void createClient() throws RemoteException;
}
class ExceptionImpl implements Exceptionable {

    @Override
    public void createClient() throws IOException {
        throw new FileNotFoundException();
    }
}
```

Dieser Code wird vom Compiler zurückgewiesen:
Implementierende APIs, die die Methode `createClient()` von Objekten des Typs `Exceptionable` aufrufen, müssen sich darauf verlassen können, dass eine `RemoteException` geworfen wird.
Wird als spezieller Typ für `Exceptionable` aber `ExceptionImpl` verwendet, und eine `FileNotFoundException` geworfen[4], würde die Ausnahmebehandlung fehlschlagen:

[4] die ebenfalls von IOException abgeleitet ist

```
public void foo(Exceptionable rmiCreator) {
    try {
        rmiCreator.createClient();
    } catch (RemoteException e) {
        // Ausnahme wird nicht berücksichtigt, wenn rmiCreator
        // ein Objekt des Typs ExceptionImpl ist
        // und eine FileNotFoundException geworfen wird
    }
}
```

Die Sprachspezifikationen gehen in "8.4.8.3. Requirements in Overriding and Hiding"[5] auch auf die zu berücksichtigenden Kompatibilitäten der `throws`-Klausel bei Implementierung bzw. Vererbung ein.

3.3.2 Vererbungsbeziehungen

Für die Implementierung von abstrakten Methoden einer als abstrakt deklarierten Klasse gelten dieselben Bedingungen wie bei Schnittstellen.

Wird in einer abgeleiteten Klasse eine Methode überschrieben, die eine *checked* Exception in der `throws`-Klausel verwendet, muss diese in der Methode der abgeleiteten Klasse nicht angegeben werden, wenn deren Methode nicht die Elternimplementierung aufruft. Ansonsten gelten an dieser Stelle die gleichen Bedingungen wie bei den geprüften Ausnahmen i.A., nämlich dass der Aufruf zu `super.methodenname()` die Ausnahme behandelt, oder die aufrufende Methode die zu erwartende Ausnahme in der `throws`-Klausel deklariert, wie folgendes Beispiel zeigt:

```
abstract class AbstractExceptionThrower {
    void foo() throws IOException {
        throw new IOException();
    }
}

class BaseExceptionThrower extends AbstractExceptionThrower {
    void fooCaller() {
        try {
           super.foo();
        } catch (IOException e) {
            // Ausnahmebehandlung
        }
    }

    void foo() throws IOException {
```

[5] `https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.8.3` - abgerufen 17.2.2024

```
            super.foo();
        }
    }
```

3.4 Weiterführende Informationen

Ausführlich gehen die Sprachspezifikationen auf das Thema "Ausnahmebehandlung" ein, unter "Chapter 11: Exceptions": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-11.html`[6].

Ullenboom zeigt in [Ull23, 589, Abschnitt 9.3.9] zusammenfassend die Unterschiede geprüfter/ungeprüfter Ausnahmen.

Bloch empfiehlt in [Blo17, 296], geprüfte Ausnahmen in den Fällen zu verwenden, in denen das Programm trotz Ausnahme in einen stabilen Zustand zurückversetzt werden kann[7], und ungeprüfte Ausnahmen nur in den Fällen, in denen es sich um (unerwartete) Programmfehler handelt[8].

Parnas und Würges stellen bereits 1976 in [PW76] fest:

> Responsibility for diagnosis and possible recovery must be assigned to the higher levels, because the lower level program does not have sufficient knowledge to determine what was desired.

und empfehlen damit früh ein Design-Paradigma, was heute fester Bestandteil von Software ist: Auf Ausnahmen der Low-Level-API wird in der Anwendungsschicht reagiert[9].

Im Hinblick auf Wartbarkeit des Codes weist *Martin* auf Folgendes hin:

> If you throw a checked exception from a method in your code and the catch is three levels above, *you must declare that excepion in the signature of each method between you and the catch.* This means that a change at a low level of the software can force signature changes on many higher levels. ([Mar08, 107, Hervorhebungen i.O.])

und sieht darin eine Verletzung des Open/Closed Prinzips[10].

[6] abgerufen 17.2.2024

[7] "*recoverable*"; so kann bei dem Abfangen der IOException im Beispiel dieses Abschnitts dem Anwender mitgeteilt werden, dass ein Fehler aufgetreten ist, und er seinen letzten Arbeitsschritt wiederholen soll, bspw. unter Angabe eines anderen Dateinamens

[8] wie bspw. der Zugriff auf den Index eines Arrays, der nicht existiert, oder das Fehlschlagen einer Casting-Operation

[9] *Faulk*: "[...] exception handling in today's distributed and Web-based applications follows the conceptual structures first laid out in this paper." [HW01, 229]

[10] "OCP: The Open-Closed Principle - Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification." [Mar03, 99]

3.5 try... finally

Bei Exceptions, die nicht abgefangen werden, sorgt `finally` für ein Ausführen des durch *finally* eingeleiteten Anweisungsblock auch im Fall einer Exception[11], wie folgendes Beispiel zeigt:

```
public class TryCatchDemo {

    public String  m1(boolean exc) {
        try {
            System.out.println("try 1.");
            if (exc) {
                throw new Exception();
            }
            System.out.println("try 2.");
            return "foo";
        } catch (Exception e) {
            System.out.println("Exception.");
        } finally {
            System.out.println("finally.");
        }

        return "return m1.";
    }
    public static void main(String[] args) {
        TryCatchDemo demo = new TryCatchDemo();
        System.out.println(demo.m1(false));
        System.out.println();
        System.out.println(demo.m1(true));
    }
}
```

Die Ausgabe des Programms lautet:

```
try 1.
try 2.
finally.
foo

try 1.
Exception.
finally.
return m1.
```

[11] Java Language Specification - 14.20.2. Execution of try-finally and try-catch-finally : https://docs.oracle.com/javase/specs/jls/se21/html/jls-14.html#jls-14.20.2 - abgerufen 26.01.2024

3.6 Anmerkungen

1. eine **implizite** Exception wird durch die Laufzeitumgebung geworfen, bspw. eine `ClassCastException`, wenn ein Casting unerwartet fehlschlägt (s. Abschnitt 2.7)
2. eine **explizite** Exception wird durch `throw new [Ausnahmetyp]` eingeleitet
3. Exceptions werden durch die `throws`-Klausel *nach außen* (bekannt) gegeben
4. Für einen `try`-Block darf es durchaus mehrere `catch`-Blöcke geben[12]

[12] s. "The catch Blocks": `https://docs.oracle.com/javase/tutorial/essential/exceptions/catch.html` - abgerufen 11.03.2024

4

Datenstrukturen und Algorithmen

4.1 Arrays / Felder

Eckige Klammern sollten bei der Variablendeklaration bzw. bei der Typdeklaration von Feldern hinter dem Datentyp stehen, nicht hinter dem Variablennamen[1]:

```
// statt
int numbers[]
// besser
int[] numbers
```

- Die Lesbarkeit wird verbessert: Die eckigen Klammern direkt hinter dem Typ weisen darauf hin, dass die Variable/der Parameter ein **Feld** von Werten des entsprechenden Typs ist.
- Bei folgendem Code ist nicht direkt ersichtlich, was gemeint ist:
  ```
  int[] vector, matrix[];
  ```
  Die Schreibweise ist äquivalent zu
  ```
  int vector[], matrix[][];
  // bzw.
  int[] vector;
  int[][] matrix;
  ```
  Die Sprachspezifikationen weisen daraufhin, dass diese Schreibweise nicht empfohlen wird:
  > We do not recommend "mixed notation" in array variable declarations, where bracket pairs appear on both the type and in declarators; nor in method declarations, where bracket pairs appear both before and after the formal parameter list. ("10.2. Array Variables": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-10.html#jls-10.2` - abgerufen 9.2.2024)

Gegenüber folgender Schreibweise

```
int[] numbers = new int[]{1, 2, 3, 4};
```

[1] s.a. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html` - abgerufen 9.2.2024

ist folgende Schreibweise

```
int[] numbers = {1, 2, 3, 4};
```

übersichtlicher, redundante Informationen sind weggefallen.
Beides ist erlaubt und führt am Ende zur Initialisierung des `int`-Arrays `[1, 2, 3, 4]`.
Eine Übergabe von `{1, 2, 3, 4}` als Argument an eine Methode, deren Signatur bspw.

```
foo(int[])
```

entspricht, ist dagegen nicht möglich; hier muss `new int[]{1, 2, 3, 4}` verwendet werden.

Bei mehrdimensionalen Arrays muss bei der Deklaration der lokalen Variable / des formalen Parameters immer die Anzahl der eckigen Klammern `[]` mit der Anzahl der Dimensionen des Arrays übereinstimmen.
Bei der Initialisierung eines mehrdimensionalen Arrays muss immer mindestens eine führende Dimension mit der Anzahl aufzunehmender Werte bestimmt werden:

```
// funktioniert nicht:
int[][] numbers = new int[][];
int[][] numbers = new int[][4];

// funktioniert:
int[][] numbers = new int[4][];
```

In Java sind mehrdimensionale Arrays **Arrays von Arrays** - sie müssen deshalb nicht rechteckig sein, wie folgendes Beispiel zeigt (vgl. [Ull23, 273 ff.]):

```
int[][] numbers = new int[4][];

numbers[0] = new int[]{1};
numbers[1] = new int[]{1, 2};
numbers[2] = new int[]{1, 2, 3};
numbers[3] = new int[]{1, 2, 3, 4};
```

4.2 Algorithmus

Die **Effizienz** eines Algorithmus wird im wesentlichen durch seine **Laufzeit** und seinen **Speicherplatzverbrauch** bestimmt.

Bei der Betrachtung kleinerer Problemgrößen fällt bei der Wahl eines Algorithmus weniger seine Effizienz ins Gewicht, sondern eher die Einfachheit seiner Implementierung und seine Verständlichkeit (vgl. [GD18b, 5 f.]).

So läuft bspw. eine Implementierung von Insertion-Sort ($O(n^2)$), die zur Sortierung $8 * n^2$ Operationen benötigt, für Eingabemengen $n \leq 43$ schneller als eine Implementierung von Merge-Sort ($O(n\ log(n))$), die $64 * (n\ log(n))$ Schritte für die Sortierung benötigt[2].

Optimaler Algorithmus

"Ein Algorithmus heißt (asymptotisch) **optimal**, wenn die obere Schranke für seine Laufzeit mit der unteren Schranke für die Komplexität des Problems zusammenfällt" ([GD18b, 20]).

Sortieralgorithmen mit einer Laufzeit von $O(n\ log\ n)$ sind **optimal**, bspw. **Merge-Sort**.

4.2.1 Elementaroperationen

Unter **Elementaroperationen** versteht man u.a. folgende Operationen (vgl. [GD18b, 6]):

- Zuweisungen: `int b = 3`
- Vergleiche: `if (b <= a)...`
- arithmetische Operationen: `b+3`
- Arrayzugriffe: `feld[j]`

Die **Kosten** für eine einzelne Elementaroperation belaufen sich auf 1 - die Kosten eines Befehls ergeben sich aus der Summe aller für diesen Befehl durchgeführten Elementaroperationen:

- `int z = b + c;`
  1 Zuweisung, 1 arithmetische Operation → Kosten: 2
- `int z = b += feld[j];`
  2 Zuweisungen, 1 arithmetische Operation, 1 indizierter Zugriff → Kosten: 4
- `goto 10;`
  keine Elementaroperation vorhanden → Kosten: 0

4.3 Rekursion

Die **Rekursionstiefe** entspricht der **Höhe** des Aufrufbaumes - der erste Aufruf der Funktion wird i.d.R. nicht zu der Rekursionstiefe gezählt.

Anzahl gleichzeitig aktiver *rekursiver* Aufrufe; der erste Aufruf wird i.d.R. nicht dazugezählt und entspricht der "Wurzel" des Rekursionsbaums (vgl. Skript (Teil 2) S.34)[3]. Die Rekursionstiefe entspricht damit der *Höhe* des Rekursionsbaumes.

[2] Diesbzgl. hatte der Selbsttest "D+A-Selbsttest-02: O-Notation" die Frage gestellt, in welchem Fall die Beurteilung eines Algorithmus bezüglich der Komplexitätsklasse nicht unbedingt angemessen sei. Die richtige Antwort hierzu lautete: "Bei sehr kleinen Eingabemengen".

[3] bzgl. der Begriffsbestimmung siehe hierzu auch [CK75, 144 f.].

4.4 Listen

Sequenzen sind **Folgen** bzw.**Listen**, auf denen eine Ordnung definiert ist (*erstes Element*, *zweites Element*, ... *n-tes Element*).

Bei einer Liste sind Elemente miteinander *verkettet*: Die Größe einer Liste kann dadurch dynamisch wachsen und es muss nicht vorher die Größe der Liste zur Aufnahme von n Elementen festgelegt werden (wie bei Feldern).

Ein Element in einer Liste zeigt auf einen **Nachfolger** `succ` und je nach Implementierung auf einen **Vorgänger** `pred`.
Hat ein Element keinen Nachfolger oder Vorgänger, sind `succ` bzw. `pred` **Nullzeiger**[4].

4.4.1 Einfach verkettete Liste

In einer **einfach verketteten Liste** zeigen Listenelemente nur auf ihren Nachfolger.

Ein *Dummy-Element* `head` zeigt auf das erste Listenelement bzw. auf `null`, falls die Liste leer ist.

Alternativ zeigt ein *Dummy-Elemente* `head` auf das erste und ein Dummy-Element `tail` auf das letzte Listenelement - hierdurch lassen sich Operationen wie `last()` und `concat()` effizienter implementieren (s. Abbildung 4.1).

4.4.2 Doppelt verkettete Liste

In einer **doppelt verketteten Liste** zeigen Listenelemente auf ihren Nachfolger *und* ihren Vorgänger.

Ein *Dummy-Element* `head` zeigt mit `pred` auf das Ende der Liste und mit `succ` auf das erste Listenelement (s. Abbildung 4.2).

Doppelt verkettete Listen verbrauchen mehr Speicher, da sie mehr Zeiger verwenden ($2 * m + 2$, m = Anzahl der Listenelemente, +2 für die `head`- / `tail`-Zeiger).

4.4.3 Darstellung im Array

Eine Liste kann auch in einem Array dargestellt werden.
Hierfür muss in Java jedoch entsprechend Speicherplatz für das Array reserviert werden - der je nach Anzahl der Einträge nicht ausgefüllt wird oder nicht ausreicht. In letzteren Fall kann es dazu führen, dass neue Elemente nicht in das Array passen und deshalb ein neues Array angefordert werden muss. In diesem Fall müssen alle Einträge in das neue Array kopiert werden.

[4] s. a. [GD18d, 49 f.] sowie "Nullzeiger": `https://de.wikipedia.org/wiki/Zeiger_(Informatik)#Nullzeiger` - abgerufen 09.03.2024

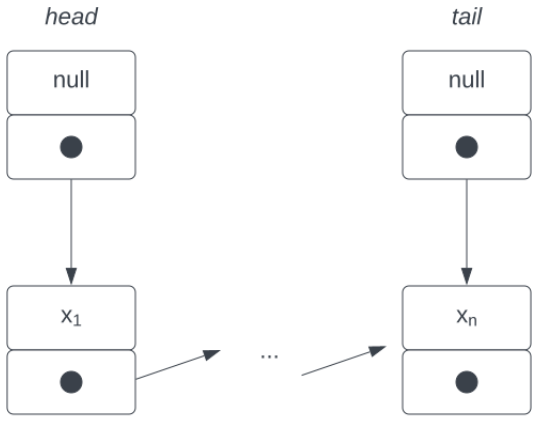

Abb. 4.1: Einfach verkettete Liste mit Zeigern auf das erste und das letzte Listenelement. (Quelle: eigene)

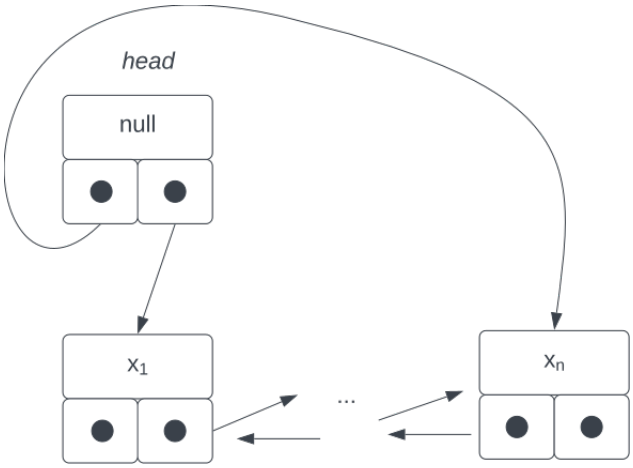

Abb. 4.2: Doppelt verkettete Liste. Das Kopfelement zeigt auf das erste und das letzte Listenelement mit *succ* bzw. *pred*. (Quelle: eigene)

4.4.4 Laufzeiten

Die Laufzeiten für die verschiedenen Listenimplementierungen sind in Tabelle 4.1 zusammengefasst.

Die doppelt verkettete Liste ist bei den Operationen *einfügen* und *entfernen* im Vorteil, da Zeiger auf die Vorgängerelemente existieren. Hierdurch muss nicht zunächst - wie in einer einfach verketteten Liste - das Vorgängerelement gesucht werden, dessen `succ`-Zeiger angepasst werden muss.
Die Laufzeiten in der Array-Implementierung sind dadurch bedingt, dass Folgeele-

mente geshifted werden müssen - wird bspw. das erste Element gelöscht, müssen n-1 Folgeelemente nach links verschoben werden.

	verkettete Liste	doppelt verkettete Liste	Liste als Array
suchen	$O(n)$	$O(n)$	$O(n)$
einfügen	$O(n)$	$O(1)$	$O(n)$
entfernen	$O(n)$	$O(1)$	$O(n)$

Tabelle 4.1: Laufzeiten für die Operationen *suchen*, *einfügen* und *entfernen* bei verschiedenen Listenimplementierungen.

4.4.5 Implementierungsbeispiele

Die folgenden Implementierungsbeispiele orientieren sich an den Praktikumsunterlagen: Dort wurde kein Dummy-Element für den Kopf der Liste verwendet.

- Einfügen eines neuen Elements vor dem Kopf einer einfach verketteten Liste: Das Einfügen funktioniert auch in einer leeren Liste, also falls `head == null` ist.

```
public void prependHead(Object data) {
    ListElement element = new ListElement(data);
    element.next = head;
    head = element;
}
```

- Einfügen eines neuen Elements hinter einem angegebenen Element: Falls das Referenz-Element `null` ist, wird eine `IllegalArgumentException` geworfen

```
public void insertAt(ListElement ref, Object data) {
    if (ref == null) {
        throw new IllegalArgumentException();
    }

    ListElement element = new ListElement(data);
    element.next = ref.next;
    ref.next = element;
}
```

- Einfügen eines Elements vor einem angegebenen Element: Falls das Referenz-Element `null` ist, wird eine `IllegalArgumentException` geworfen
  Falls das Referenz-Element das `head`-Element ist, wird die o.a. Methode `prependHead()` aufgerufen; ansonsten wird der Vorgänger des Referenz-Elementes gesucht und die Zeiger entsprechend angepaßt.

Die Implementierung berücksichtigt den Fall, dass das angegebene Referenz-Element nicht Teil der Liste ist, in dem Fall wird ebenfalls eine `IllegalArgumentException` geworfen.

```
public void insertBefore(ListElement ref, Object data) {
    if (ref == null) {
        throw new IllegalArgumentException();
    }

    if (ref == head) {
        prependHead(data);
        return;
    }

    ListElement prev = head;
    while (prev != null && prev.getNext() != ref) {
        prev = prev.getNext();
    }
    if (prev == null) {
        throw new IllegalArgumentException(
            "ref does not seem to be in this list"
        );
    }

    ListElement element = new ListElement(data);
    element.next = ref;
    prev.next = element;
}
```

- Entfernen eines Elements:
  Falls das zu entfernende Element `null` ist, wird eine `IllegalArgumentException` geworfen
  Eine Methode für die Suche nach einem Vorgänger-Element ist vorausgesetzt (s. Methode `insertBefore()`)
  Wird kein Vorgänger-Element gefunden, wird davon ausgegangen, dass das zu löschende Element das `head`-Element ist

```
public void delete(ListElement del) {
    if (del == null) {
        throw new IllegalArgumentException();
    }

    ListElement prev = getPredecessor(del);
    if (prev == null) {
        head = del.next;
        return;
    }
```

```
            prev.next = del.next;
        }
```

- Anhängen eines Elements an das Ende (`tail`) einer Liste
  Im Folgenden wird die Existenz eines `tail`-Elementes vorausgesetzt

```
        public void append(Object data) {
            ListElement element = new ListElement(data);

            if (tail == null) {
                head = element;
                head = tail;
                return;
            }

            tail.next = element;
            tail = element;
        }
```

4.5 Hashing

Um eine *Adresse* für ein zu speicherndes Elemente zu berechnen, werden **Hashverfahren** genutzt.
Der Wert des zu speichernden Elementes dient dabei als Grundlage zur Berechnung der Adresse.
Der Speicher, in dem die Elemente abgelegt werden soll, bezeichnet man dabei als *buckets*: $B_0, \ldots, B_{m-1}$ (vgl. [GD18a, 115]).

Es wird unterschieden zwischen **geschlossenem** und **offenem** Hashing (vgl. [GD18a, 116]): Bei offenem Hashing werden Überläufer verkettet, bei geschlossenem Hashing kann es zu Kollisionen mit bereits belegten Adressen kommen - über **rehashing**[5]) können Adressen neu berechnet werden, woraus sich **Sondierungsfolgen** ergeben.

Während bei offenem Hashing die m Buckets ihre Überläufer jeweils einfach in einer verketteten Liste speichern, ist die Anzahl der speicherbaren Elemente in einem Bucket bei geschlossenem Hashing durch b begrenzt. Bei *geschlossenem Hashing* wird oft der Spezialfall $b = 1$ betrachtet; trotzdem kann auch bei geschlossenem Hashing durchaus $b > 1$ sein (s. Abbildung 4.3).

4.5.1 Hashfunktionen

Für die Berechnung einer Adresse wird oft der *modulo*-Operator verwendet

[5] auch: *offene Adressierung* (vgl. [GD18a, 119])

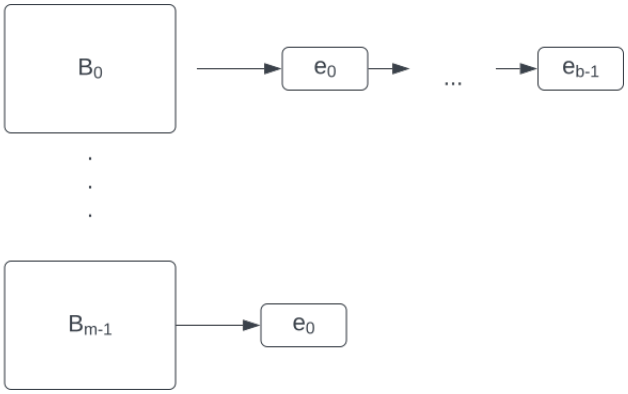

Abb. 4.3: Die Abbildung stellt schematisch geschlossenes Hashing dar. Hierbei ist die Anzahl der speicherbaren Elemente mit b pro Bucket begrenzt, sodass sich die zur Verfügung stehenden Speicherplätze aus $m * b$ berechnen. Bei offenem Hashing werden Überläufer hingegen einfach in verketteten Listen gespeichert. (Quelle: eigene)

$$b \; mod \; m \tag{4.1}$$

für den wie folgt gilt[6]:

$$b \; mod \; m = b - \lfloor \frac{b}{m} \rfloor * m \tag{4.2}$$

Beispiele

- $-1 \; mod \; 7 = -1 - \lfloor \frac{-1}{7} \rfloor * 7 = -1 - (-1 * 7) = 6$
- $19 \; mod \; 7 = 19 - \lfloor \frac{19}{7} \rfloor * 7 = 19 - (2 * 7) = 5$
- $-8 \; mod \; 6 = -8 - \frac{-8}{6} * 6 = -8 - (-2 * 6) = 4$

4.5.2 Doppel-Hashing

Bei Doppel-Hashing wird eine Sondierungsfunktion $s(k, i)$ benutzt, die eine zweite Hashfunktion $h_2(k)$ neben der eigentlichen Hashfunktion $h_1(k)$ verwendet.
Die Adresse für k wird zunächst über $h_1(k)$ berechnet. Kommt es zu einer Kollision, wird die *Sondierungsfunktion* verwendet, beginnend mit $i = 1$, bis keine weiteren Kollisionen auftreten[7].

4.5.3 Anmerkungen

Ottmann und Widmayer beschreiben unter *Offene Hashverfahren* das geschlossene Hashing (vgl. [OW17b, 203 ff.]); *Cormen et al.* behandeln dasselbe unter **open adressing** in [CL22, 293 ff.].

Ottman und Widmayer's Abschnitt *Hashverfahren mit Verkettung der Überläufer* (vgl. [OW17b, 198 ff.]) beschreibt dasselbe wie *Sedgewicks und Waynes*'s *Hashing*

[6] s. a. "Modulo": `https://de.wikipedia.org/wiki/Division_mit_Rest#Modulo` - abgerufen 09.03.2024
[7] s. a. "Doppel-Hashing": `https://de.wikipedia.org/wiki/Doppel-Hashing` - abgerufen 11.03.2024

with separate chaining in [SW11, 464 ff.]; bei Güting und Dieker wird hierzu die Bezeichnung **offenes Hashing** verwendet (vgl. [GD18a, 116]).

4.6 (Binäre) Bäume

- Der **Grad** eines Knotens ist die Anzahl seiner Nachfolger.
- Ein **innerer Knoten** ist ein Knoten mit Grad ≥ 1.
- Ein Blatt ist ein Knoten mit Grad 0.
- Der **Grad eines Baums** ist der maximale Grad eines Knotens im Baum (vgl. [GD18c, 102]).

Der **direkte linke** bzw. **direkte rechte Nachfolger** eines inneren Knotens ist sein linker bzw. rechter Nachfolger.

Die **Tiefe** eines Knotens ist der Abstand des Knotens von der Wurzel, rekursiv definiert durch

$$T(k) = \begin{cases} 0 & \text{falls k Wurzel} \\ 1 + T(k') & \text{sonst} \end{cases} \tag{4.3}$$

wobei k' der direkte Vorgänger ("Elternknoten") des Knotens k ist.

Die **Höhe** eines Baumes ist das Maximum der *Tiefe* seiner Knoten[8].

Die **Höhe** eines Knotens ist der längste Pfad von dem Knoten zu einem Blatt.

Der Kurs definiert einen Baum als **vollständig**, wenn alle inneren Knoten zwei Söhne haben und alle Blätter die gleiche Tiefe[9].

Die **maximale Höhe** eines Baumes mit n Knoten ist $n-1$ - diese wird in *entarteten* Bäumen erreicht: Hier hat jeder Knoten maximal einen Nachfolger (vgl. Skript (Teil2) S.103).

Die **minimale Höhe eines Baumes** mit n Knoten ist $\lceil log_2(n+1) \rceil - 1$: *vollständige Bäume* besitzen minimale Höhe (vgl. Skript (Teil2) S.103).

4.6.1 Preorder

Wird ein Baum in **Preorder**-Reihenfolge traversiert, wird zunächst die Wurzel besucht, dann die Knoten des linken Teilbaums, dann die Knoten des rechten Teilbaums (*KLR*).

[8] also der längste vorkommende Pfad, von der Wurzel ausgehend
[9] s. Skript (Teil 2) S. 101.

Beispiel

Sei der Baum wie in Abbildung 4.4 gegeben[10].

Die **Preorder**-Traversierung des Baumes ergibt 3 4 2 7 1 5 9 1.

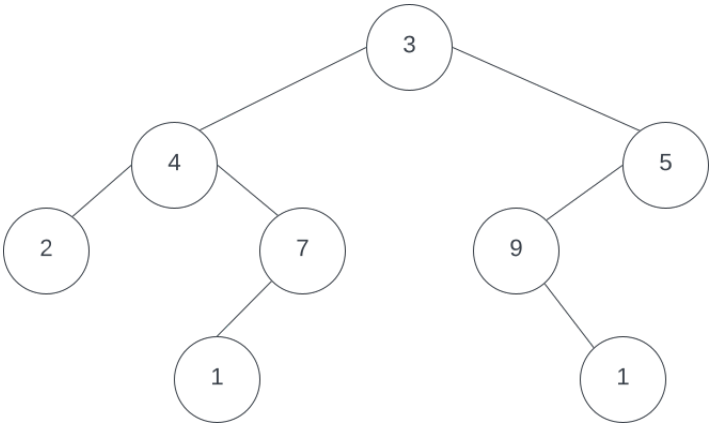

Abb. 4.4: Binärer Baum (Quelle: eigene)

4.6.2 Inorder

Wird ein Baum in **Inorder**-Reihenfolge traversiert, werden zunächst die Knoten des linken Teilbaums besucht, dann die Wurzel, dann die Knoten des rechten Teilbaums (LKR).

Beispiel

Die **Inorder**-Traversierung des Baumes in Abbildung 4.4 ergibt 2 4 1 7 3 9 1 5.

4.6.3 Postorder

Wird ein Baum in **Postorder**-Reihenfolge traversiert, werden zunächst die Knoten des linken Teilbaums besucht, dann die Knoten des rechten Teilbaums, dann die Wurzel selber (LRK) .

Beispiel

Die **Postorder**-Traversierung des Baumes in Abbildung 4.4 ergibt 2 1 7 4 1 9 5 3.

[10] die in diesem Abschnitt verwendeten Beispiele sind den Tutoraufgaben des SS2015, Lehrstuhls für Informatik 2, RWTH Aachen, entnommen (s. `https://moves.rwth-aachen.de/wp-content/uploads/SS15/dsal/Loesung2.pdf` - abgerufen 08.03.2024)

4.6.4 Rekonstruktion

Ein binärer Baum lässt sich nur mit Hilfe der Inorder- *und* der Pre- oder Postorder-Reihenfolge *eindeutig* rekonstruieren.

Ist der Baum ein **binärer Suchbaum**, reicht die Preorder- oder Postorder-Darstellung, um den Baum eindeutig zu rekonstruieren: Die Inorder-Reihenfolge kann aus der Ordnung der Schlüssel wiederhergestellt werden.
Die Inorder-Darstellung eines binären (Such-)Baumes reicht niemals zur eindeutigen Rekonstruktion desselben.
Ist nur die **level-order** Darstellung gegeben, läßt sich ein **links-vollständiger** Binärer Baum anhand dieser Linearisierung eindeutig rekonstruieren.

Beispiele

Sei folgende Inorder- und Preorder-Reihenfolge gegeben:

Inorder: 5 8 4 7 3 1 6 9 2
Preorder: 3 4 5 8 7 2 6 1 9

Der rekonstruierte Baum ist in Abbildung 4.5 dargestellt.

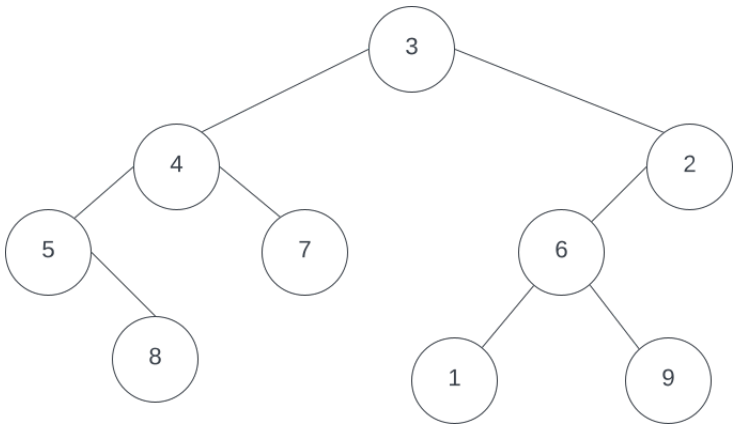

Abb. 4.5: Binärer Baum mit der Inorder-Reihenfolge 5 8 4 7 3 1 6 9 2 sowie der Preorder-Reihenfolge 3 4 5 8 7 2 6 1 9 (Quelle: eigene)

Sei folgende Inorder- und Postorder-Reihenfolge gegeben:

Inorder: 5 1 2 4 6 7 3 9 8
Postorder: 5 2 4 1 3 8 9 7 6

Der rekonstruierte Baum ist in Abbildung 4.6 dargestellt.

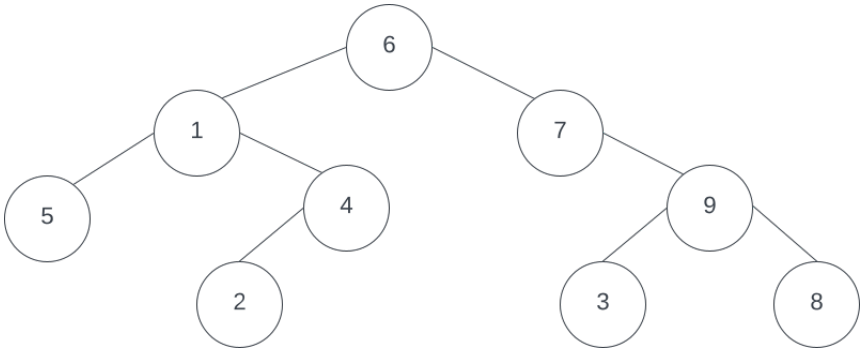

Abb. 4.6: Binärer Baum mit der Inorder-Reihenfolge 5 1 2 4 6 7 3 9 8 sowie der Postorder-Reihenfolge 5 2 4 1 3 8 9 7 6 (Quelle: eigene)

4.6.5 Einfügen / Suchen / Löschen in binären Suchbäumen

Die Kosten für die Operationen *Einfügen / Suchen / Löschen* belaufen sich auf $O(h)$, wobei h die *Höhe* des Baumes ist.
h kann zwischen $\lceil log_2\ (n+1) \rceil - 1$ (vollständiger Baum) und $n-1$ liegen, wobei n die Anzahl der Knoten des Baumes ist[11]:
Im worst-case ist ein Baum zu einer linearen Liste entartet[12], so dass alle Knoten durchlaufen werden müssen, um einen Schlüssel für eine nachfolgende *insert- / delete*-Operation zu finden.

Im Mittel wird für eine Einfügeoperation $O(log\ n)$ Zeit benötigt[13].

Der Aufwand für den Aufbau eines binären Suchbaumes mit n bereits sortierten Elementen ist $O(n^2)$ (vgl. [GD18a, 235 f.]).

4.6.6 Löschen

In einem *binären Suchbaum* gilt (rekursiv), dass der Schlüssel eines *direkten linken Nachfolgers* **kleiner** als der Schlüssel seines *direkten Vorgängers* ist.
Für den *direkten rechten Nachfolger* gilt, dass sein Schlüssel **größer** als der Schlüssel seines direkten Vorgängers ist.

Wenn in einem binären Suchbaum ein Knoten gelöscht wird, muss seine Ordnung wiederhergestellt werden.

Dies erreicht man, indem man den gelöschten Knoten durch seinen **symmetrischen Nachfolger** bzw. **symmetrischen Vorgänger** ersetzt (vgl. [OW17a, 269, 272]).
Der **symmetrische Nachfolger** ist hierbei der Knoten im *rechten* Teilbaum, der

[11] Bei *Ottmann und Widmayer* anhand der *inneren Knoten*: Dann berechnet sich h aus $\lceil log_2\ (n+1) \rceil$ (vollständiger Baum) und n (vg. [OW17a, 275])
[12] bspw. weil die einzufügenden Schlüssel schon sortiert vorliegen
[13] vgl. [GD18a, 136 ff.]

am weitesten *links* steht (→ kleinster Schlüssel im rechten Teilbaum).
Der **symmetrische Vorgänger** ist hingegen der Knoten im *linken* Teilbaum, der am weitesten *rechts* steht(→ größter Schlüssel im linken Teilbaum).

Generell wird beim Löschen so vorgegangen:

- **Fall 1**: Der zu löschende Knoten ist ein Blatt
  In diesem Fall muss lediglich die Referenz auf das Blatt entfernt werden.
- **Fall2**: Der zu löschende Knoten hat nur einen direkten Nachfolger
  Der direkte Vorgänger des zu löschenden Knoten referenziert den direkten Nachfolger des zu löschenden Knotens.
- **Fall3**: Der zu löschende Knoten hat einen linken *und* einen rechten Nachfolger
  Der zu löschende Knoten muss durch seinen symmetrischen Nachfolger oder Vorgänger ersetzt werden.

Sei bspw. wie in Abbildung 4.7 der binäre Suchbaum gegeben, dessen Knoten mit dem Schlüssel 5 gelöscht werden soll.
Der symmetrische Nachfolger für diesen Knoten ist der Knoten mit dem Schlüssel 6, der symmetrische Vorgänger ist der Knoten mit dem Schlüssel 4 - beide kommen als Ersatz für den gelöschten Knoten in Frage.

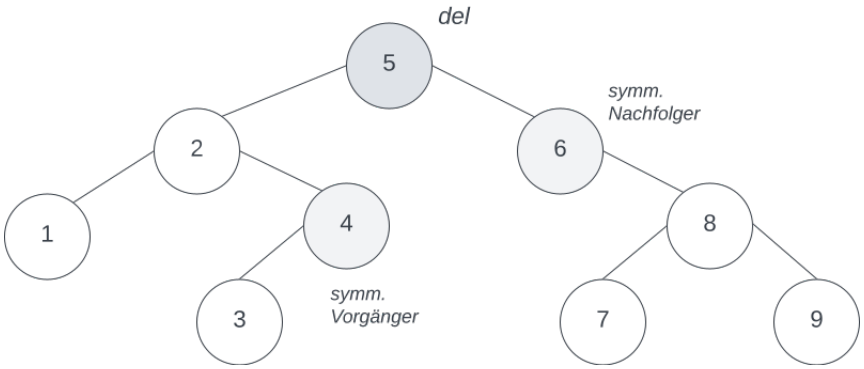

Abb. 4.7: Wenn der Knoten mit dem Schlüssel 5 gelöscht wird, muss entweder der Knoten mit dem Schlüssel 4 an Stelle des gelöschten Knotens, oder der Knoten mit dem Schlüssel 6, um die Ordnung in dem binären Suchbaum beizubehalten.(Quelle: eigene)

Es ist darauf zu achten, die Nachfolgerknoten / Vorgängerknoten beim Löschen korrekt umzuhängen:

- **Fall 1**: Der symmetrische Vorgänger wird verwendet
  Da der symmetrische Vorgänger höchstens *einen direkten linken Nachfolger* haben kann, wird dieser Knoten (in dem Beispiel der Knoten mit Schlüssel 3) nun der neue *direkte rechte Nachfolger* des direkten Vorgängers (Schlüssel 2) des symmetrischen Vorgängers. Der neue direkte linke Nachfolger des Knotens mit dem Schlüssel 4 wird der Knoten mit dem Schlüssel 2.

- **Fall 2**: der symmetrische Nachfolger wird verwendet
  Da der symmetrische Nachfolger höchstens *einen direkten rechten Nachfolger* haben kann, wird dieser Knoten nun der neue *direkte linke Nachfolger* des direkten Vorgängers des symmetrischen Nachfolgers. In dem angegebenen Beispiel muss nichts weiter geschehen, da der direkte Vorgänger des symmetrischen Nachfolgers der zu löschende Schlüssel ist. Allerdings bekommt der Knoten mit dem Schlüssel 6 nun einen neuen direkten linken Nachfolger, und zwar den Knoten mit dem Schlüssel 2.

4.7 Heap

Ein **Heap** (*Halde*) ist eine Datenstruktur, die nach *Ottmann und Widmayer* wie folgt definiert ist:

> Eine Folge $F = K_1, k_2, \ldots, k_N$ von Schlüsseln nennen wir einen Heap, wenn $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$ für $2 \leq i \leq N$ gilt. Anders ausgedrückt: $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$, sofern $2i \leq N$ bzw. $2i + 1 \leq N$. ([OW17c, 106])

Die Folge $12, 9, 6, 4, 8, 5, 1, 2$ ist ein Heap (s. Tabelle 4.2). Es gilt:

1. $12_1 \geq 9_2 \wedge 12_1 \geq 6_3$
2. $9_2 \geq 4_4 \wedge 9_2 \geq 8_5$
3. $6_3 \geq 5_6 \wedge 6_3 \geq 1_7$
4. $4_4 \geq 2_8$

i	1	2	3	4	5	6	7	8
k_i	12	9	6	4	8	5	1	2

Tabelle 4.2: Beispiel für einen Heap.

Insbesondere liefert die Beziehung $k_i \geq k_{2i}$ und $k_i \geq k_{2i+1}$ eine einfache Vorschrift für die grafische Darstellung eines Heaps als **links-vollständigen** binären Baumes[14] (s. Abbildung 4.8):

> Ein Binärbaum ist ein Heap, wenn der Schlüssel jedes Knotens mindestens so groß ist wie die Schlüssel seiner beiden Söhne (falls es diese gibt). ([OW17c, 107])

[14] alle Ebenen bis auf die letzte sind voll besetzt; auf der letzten Ebene sitzen die Knoten so weit links wie möglich (vgl. [GD18a, 154])

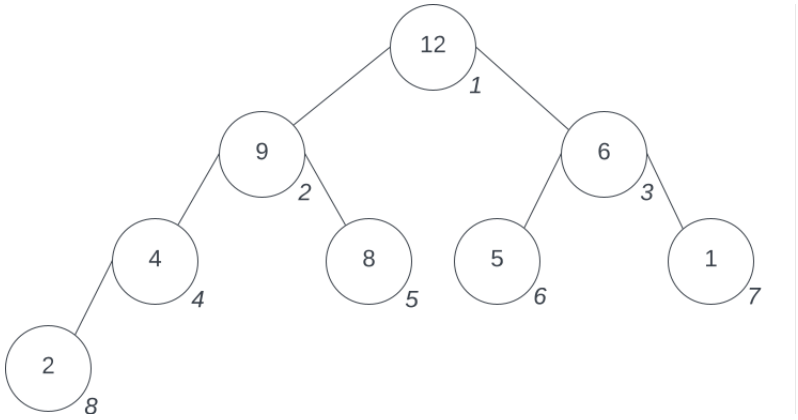

Abb. 4.8: Der Heap $12, 9, 6, 4, 8, 5, 1, 2$ als binärer Baum (Quelle: eigene)

4.7.1 Min-Heap / Max-Heap

Man kann zwischen `Min-Heaps` und `Max-Heaps` unterscheiden.
Bisherige Betrachtungen bezogen sich auf einen `Max-Heap` - die Schlüssel der direkten Nachfolger eines Knotens sind kleiner-gleich des Schlüssels des Vaterknotens.
Bei **Min-Heaps** ist die Bedingung, dass die Schlüssel der direkten Nachfolger eines Knotens größer-gleich des Schlüssels des Vaterknotens sind.
Sofern nicht anders angegeben, beziehen sich die nachfolgenden Beispiele und Anwendungen der Heap-Algorithmen auf Max-Heaps.

4.7.2 Algorithmen für Heaps

Sind nach dem Einfügen eines Schlüssels die Heap-Bedingungen verletzt, müssen diese wiederhergestellt werden.

4.7.2.1 swim

Bottom-up reheapify (swim) (vgl. [SW11, 315]) kann angewendet werden, um einen neu einzufügenden Schlüssel an seine richtige Position in dem Heap zu bewegen.
In dem o.a. Beispiel soll ein neuer Schlüssel 20 eingefügt werden.
Entsprechend der Implementierung als Folge wird der Schlüssel an das Ende angehangen: $12, 9, 6, 4, 8, 5, 1, 2, 20$

Die Heap-Bedingung für k_4 ist nun verletzt wegen $4_4 \geq 2_8 \wedge 4_4 \ngeq 20_9$. Um die Heap-Bedingung wieder herzustellen, muss der Schlüssel an seine richtige Position wandern.
Hierzu wird er sukzessive mit seinem direkten Vorgänger (dem Vaterknoten $k_{\lfloor \frac{i}{2} \rfloor}$) ausgetauscht, bis $k_{\lfloor \frac{i}{2} \rfloor}$ größer k_9 ist oder die Wurzel des Heaps erreicht ist.
Anwendung dieses Verfahrens liefert die Folge $20, 12, 6, 9, 8, 5, 1, 2, 4$ (s. Abbildung 4.9).

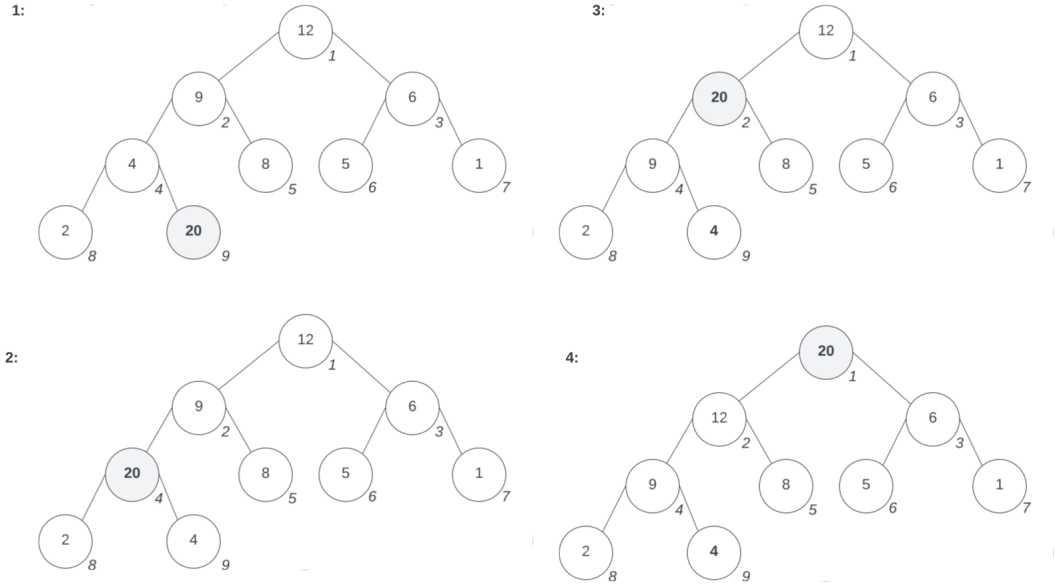

Abb. 4.9: Wiederherstellung der Heap-Bedingungen nach Einfügen des Schlüssels 20 durch **bottom-up reheapify**(Quelle: eigene)

4.7.2.2 sink

Sollte ein Schlüssel *kleiner* als seine direkten Nachfolger sein, kann **top-down reheapify (sink)**[15] angewendet werden.

Hierbei tauscht der Schlüssel die Position mit dem jeweils *größeren* seiner direkten Nachfolger, bis die Heap-Bedingungen wieder hergestellt sind (s. Abbildung 4.10).

4.8 Anmerkungen

- die Klasse `Vector` hat 4 verschiedene Konstruktoren[16], von denen 3 in den Testfragen abgefragt wurden:
- Der parameterlose Konstruktor erzeugt ein Objekt mit einer Kapazität von 10
- Der Konstruktor `Vector(initialCapacity: int)` erzeugt ein Objekt mit der angegebenen Kapazität
- Der Konstruktor `Vector(initialCapacity:int, capacityIncrement: int)` erzeugt ein Objekt mit der angegebenen Kapazität und dem angegebenen Kapazitäts-Inkrement

[15] *sift down* (*versickern*) bei *Ottmann und Widmayer* (vgl. [OW17c, 107 f.])
[16] s. "Class Vector<E> - Constructor Summary:" `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Vector.html#constructor-summary` - abgerufen 11.03.2024

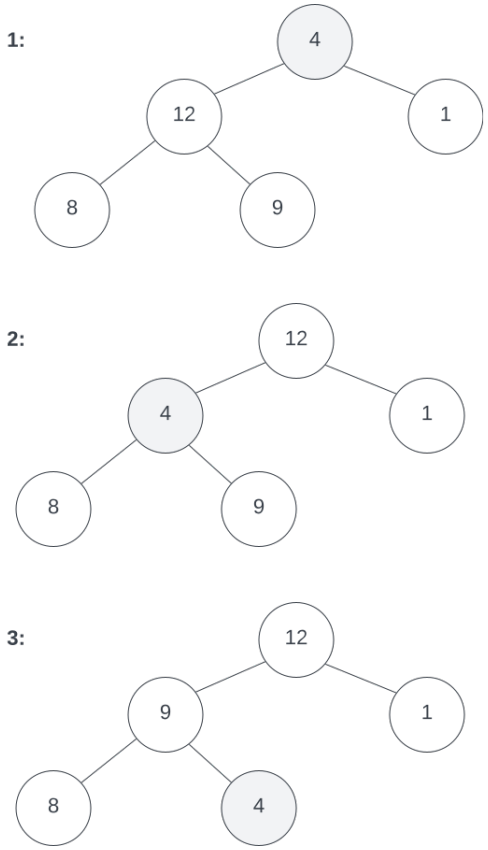

Abb. 4.10: Herstellung der Heap-Bedingungen für die Folge $4, 12, 1, 8, 9$ mittels **top-down reheapify** (Quelle: eigene)

5

Sortierverfahren

5.1 Grundlagen

Der Abschnitt wiederholt wesentliches zu *einfachen* Sortierverfahren[1] wie **Selection-Sort**, **Insertion-Sort** und **Bubblesort**, deren Laufzeitkomplexität in $O(n^2)$ liegt[2].
Außerdem werden **Divide-and-Conquer**-Verfahren wie **Merge-Sort** ($O(n\,log\,n)$) und **Quicksort** ($O(n^2)$) wiederholt.

5.1.1 Allgemeine Sortierverfahren

Unter **allgemeinen Sortierverfahren** fasst man diejenigen Sortierverfahren zusammen, die Informationen über die relative Anordnung von zu sortierenden Schlüsseln nur über paarweise Schlüsselvergleiche gewinnen (vgl. [OW17c, 130]).
Alle in diesem Abschnitt behandelten Sortierverfahren sind *allgemeine* Sortierverfahren[3].

5.1.2 Klassifikation von Sortieralgorithmen

Güting und Dieker klassifizieren Sortieralgorithmen neben der Effizienz und den angewandten Methoden u.a. nach folgenden Kriterien (vgl. [GD18e, 169 f.]):

- **intern / extern**
  interne Verfahren halten alle Daten im Hauptspeicher, *externe* laden nur einen Teil davon in den Speicher (bspw. beim Sortieren von Daten, die auf Diskette / Platte / Band vorliegen (vgl. [OW17c, 80])).
- **in place (in situ)**
  Sortierverfahren, die keinen zusätzlichen Speicherplatz zum Sortieren der Daten benötigen, sortieren place.
  Die Eingangsfolge und die Ausgangsfolge werden im selben Array dargestellt, Vertauschungen von Schlüsseln erfolgen in genau diesem Array.

[1] vgl. *Güting und Dieker* in [GD18e, 170]; bei *Ottmann und Widmayer* auch *elementare* Sortierverfahren (vgl. [OW17c, 82 ff.])
[2] im Folgenden bezieht sich O immer auf den **worst-case**, sofern nicht anders angegeben
[3] ein Sortierverfahren, dass nicht über Schlüsselvergleiche sortiert, ist *Radixsort* (vgl. [OW17c, 121])

Alle in diesem Kapitel behandelten Sortierverfahren fallen in diese Kategorie, bis auf **Merge-Sort**.

- **stabil**
  Sortierverfahren, bei denen die Ergebnisfolge gleicher Schlüssel dieselbe ist wie die Ausgangsfolge, nennt man **stabil**:
  > [...] die Reihenfolge von Elementen mit gleichem Sortierschlüssel wird während des Sortierverfahrens nicht vertauscht. ([OW17c, 164]).

  Die Sortierverfahren **Bubblesort**, **Insertion-Sort** und **Merge-Sort** sind allesamt stabile Sortierverfahren.
  Nicht stabil sind **Selection-Sort** und **Quicksort**.
- **natürlich**
  natürliche Sortierverfahren arbeiten bei vorsortierten Daten schneller als bei unsortierten (bspw. **Insertion-Sort**, wohingegen **Selection-Sort** i.d.R. auch bei vorsortierten Daten die gleiche Laufzeitkomplexität aufweist, wie bei unsortierten)

5.1.3 Untere Schranke

Jedes allgemeine Sortierverfahren benötigt zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel wenigstens $\Omega(N\ log\ N)$ Schlüsselvergleiche. ([OW17c, 154, Satz 2.4]).

5.1.4 divide-and-conquer-Paradigma

Güting und Dieker beschreiben das **divide-and-conquer**-Paradigma in [GD18e, 174]:

- Ist die Objektmenge klein genug, löse das Problem direkt.
- *Divide* Ansonsten zerlege die Objektmenge in mehrere Teilmengen (möglichst gleicher Größe)
- *Conquer* Löse das Problem rekursiv für die Teilmenge
- *Merge* Berechne aus den für die Teilmengen erhaltenen Lösungen eine Lösung für das Gesamtproblem

5.2 Bubblesort

Bubblesort ist ein **stabiles**, auf Schlüsselvergleichen basierendes Sortierverfahren, dass **in place** sortiert.

5.2.1 Methode

Bubblesort vergleicht zwei benachbarte Elemente und vertauscht diese miteinander, falls ihre relative Reihenfolge nicht der erwarteten Sortierreihenfolge entspricht.

Nach dem ersten Durchlauf steht der größte Schlüssel[4] am Ende des Feldes.
Dann wird der Vorgang wiederholt, es muss aber nicht mehr mit dem Schlüssel an Position $n-1$[5] verglichen werden, da dieser Schlüssel bereits seine endgültige Position in dem sortierten Feld eingenommen hat.
Das wird so lange wiederholt, bis keine Vertauschungen mehr aufgetreten sind oder in u.a. Implementierung das Ende der äußeren Schleife erreicht wurde.

5.2.2 Implementierung

```
boolean inversed = true;
for (i = n - 1; i >= 0; i--) {
    inversed = false;
    for (j = 0; j < i; j++) {
        if (arr[j] > arr[j + 1]) {
            swap(arr, j, j+1);
            inversed = true;
        }
    }
    if (!inversed) {
        break;
    }
}
```

5.2.3 Laufzeit

Ist das Feld absteigend sortiert, wird in jeder Iteration der `while`-Schleife eine Fehlstellung (*Inversion*) behoben (s. Abbildung 5.1). Hierfür werden in jedem Durchlauf n Schlüsselvergleiche durchgeführt.
Bei einer Eingabegröße von n mit $\sum_{i=1}^{n}(n-i)$ Fehlstellungen (vgl. [OW17c, 87]) ergibt sich somit eine **Laufzeitkomplexität** von $O(n^2)$.

- **worst-case**: **Anzahl der Vergleiche und Vertauschungen**: $\frac{n*(n-1)}{2}$ $(O(n^2))$
- **average-case**: $O(n^2)$
- **best-case**: $O(n)$

Im günstigsten Fall muss das Feld nur einmal durchlaufen werden, um zu überprüfen, ob eine Vertauschung vorgenommen werden muss.

5.3 Selection-Sort

Selection-Sort (*Sortieren durch Auswahl*) ist ein auf Schlüsselvergleichen basierendes, **nicht-stabiles** Sortierverfahren, das **in place** sortiert.

[4] oder der kleinste Schlüssel, entsprechend den Anforderungen an die Sortierreihenfolge
[5] die Anzahl der benötigten Vergleiche reduziert sich entsprechend nach jeder Iteration um mindestens 1

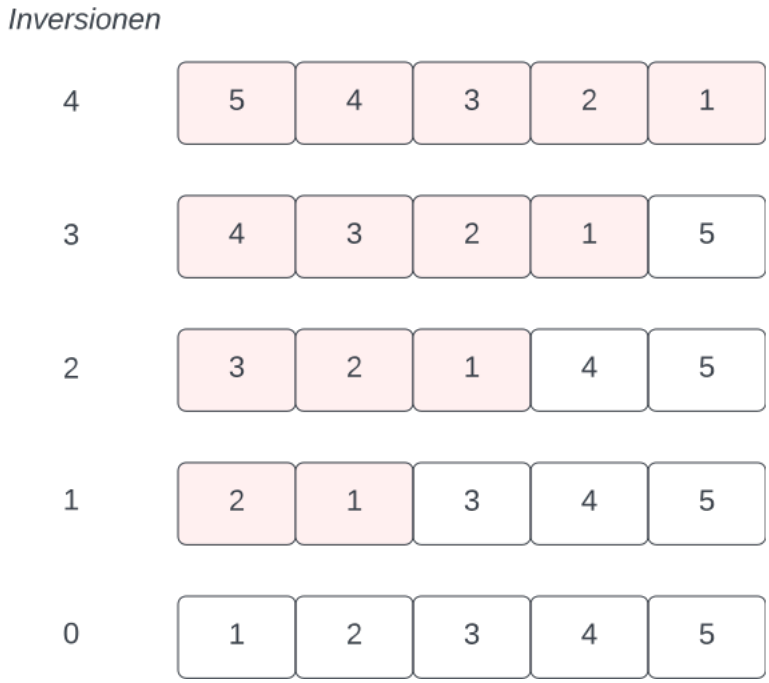

Abb. 5.1: Fehlstellungen in einem absteigend sortierten Feld. (Quelle: eigene)

5.3.1 Methode

Das Sortierverfahren vergleicht zwei Teilfolgen des zu sortierenden Feldes miteinander.
Hierbei ist eine Teilfolge sortiert, die andere unsortiert. Die sortierte Teilfolge ist zunächst leer.
Das jeweils kleinste Element aus der *unsortierten* Teilfolge wird an das Ende der sortierten Teilfolge eingefügt[6].
Das ganze wird so lange wiederholt, bis die unsortierte Teilfolge nur noch aus einem Element besteht, danach ist das Feld sortiert.

5.3.2 Implementierung

```
for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[minIndex]) {
            minIndex = j;
        }
    }
    int tmp = arr[minIndex];
    arr[minIndex] = arr[i];
```

[6] bzw. das größte, je nach Anforderungen an die Sortierreihenfolge

```
        arr[i] = tmp;
    }
```

5.3.3 Laufzeit

Die äußere Schleife wird $n-1$-mal durchlaufen, die innere - in Abhängigkeit von i, jeweils $n-(i+1)$ mal.
Mit

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \tag{5.1}$$

folgt die Laufzeitkomplexität $O(n^2)$.

- **Anzahl der Vergleiche**: $\frac{n*(n-1)}{2}$
- **Vertauschungen**: n - jedes Element höchstens ein mal[7]

- **worst-case**: $O(n^2)$
- **average-case**: $O(n^2)$
- **best-case**: $O(n^2)$

Da in der o.a. Implementierung keine Abbruchbedingung vorliegt, gilt die Laufzeitkomplexität auch für den **best**- sowie **average**-case.

5.4 Insertion-Sort

Insertion-Sort ist ein auf Schlüsselvergleichen basierendes, **stabiles** Sortierverfahren, das **in place** sortiert.

5.4.1 Methode

Bei **Insertion-Sort** wird das zu sortierende Feld in zwei Teilfolgen unterteilt - eine unsortierte und eine sortierte.
Die sortierte Folge besteht zu Beginn nur aus einem Element[8].
Aus der unsortierten Teilfolge werden Elemente der Reihe nach entnommen und in die sortierte Folge an der richtigen Stelle eingefügt, wodurch Schlüssel ihre Position in der sortierten Folge ggf. ändern müssen.

[7] s. Skript (Teil 2) "7.3.1 Sortieren durch direktes Auswählen"
[8] womit diese als bereits sortiert gilt

5.4.2 Implementierung

```
for (int i = 1; i < n; i++) {
    int min = arr[i];
    int j = i;
    while (j > 0 && arr[j - 1] > min) {
        arr[j] = arr[j - 1];
        j--;
    }
    arr[j] = min;
}
```

5.4.3 Laufzeit

Die äußere Schleife wird $n-1$ mal durchlaufen, die innere so oft, bis keine Inversion mehr festgestellt wird[9].
Im **worst-case** muss die innere Schleife allerdings in Abhängigkeit von i jeweils i-mal durchlaufen werden.

Mit

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i} 1 \tag{5.2}$$

folgt die Laufzeitkomplexität $O(n^2)$.

- **Anzahl der Vergleiche und Vertauschungen**: $\frac{n*(n-1)}{2}$ $(O(n^2))$
- **average-case**: $O(n^2)$
- **best-case**: $O(n)$

Lineares Laufzeitverhalten

Insertion-Sort eignet sich sehr gut für *vorsortierte* Felder, wo es ein *lineares Laufzeitverhalten* aufweist (vgl. [CL22, 188]).

5.5 Quicksort

Quicksort ist ein **nicht stabiler**, auf Schlüsselvergleichen basierender Sortieralgorithmus, der **in place** sortiert.

[9] das ist einer der wesentliche Unterschied zu Selection-Sort

5.5.1 Methode

Quicksort nutzt **divide-and-conquer**, um ein Feld zu sortieren.
Die wesentliche Arbeit erfolgt hier in den Partitionierungs-Schritten, im Gegensatz zu **Merge-Sort**, hier findet die eigentliche Arbeit im Merge-Schritt statt (vgl. [GD18e, 174]).

Bei Quicksort wird ein Schlüssel aus dem Feld als **Pivot-Element** p gewählt (im folgenden immer das rechte einer Teilfolge).
Alle Schlüssel, die kleiner als das Pivotelement sind, werden in die eine Hälfte geschrieben, alle die größer sind, in die andere. Das Pivotelement selber wird zwischen die beiden Hälften geschrieben, die dann durch Rekursion wiederholt bearbeitet werden, bis die Teilfelder die Größe 1 haben, womit das gesamte Feld als sortiert gilt:
Zwei Positionszeiger i und j wandern symmetrisch von links nach rechts (i) bzw. rechts nach links (j) in der Folge, bis ein Schlüssel $k_i > p$ und $k_j < p$ gefunden wurde.
Die Schlüssel an Position i und j werden dann miteinander getauscht.
Kreuzen sich i und j, gilt für alle Schlüssel $k_u < p$ (mit $u < i$), und für alle Schlüssel $k_v > p$ (mit $v > j$)). In dem Fall wird der Schlüssel an Position i mit dem Pivotelement p getauscht, und das Feld wird an dieser Stelle in zwei Teilfolgen aufgeteilt, wonach die Prozedur für die entstandenen Teilfolgen wiederholt wird (s. Abbildung 5.2).

5.5.2 Implementierung

```
quicksort (int[] arr, int l, int r) {
    if (l >= r) {
        return;
    }
    int i = l;
    int j = r - 1;
    int pivot = arr[r];
    while (i <= j) {
        while (i < r && arr[i] <= pivot) {
            i++;
        }
        while (j >= 0 && arr[j] > pivot) {
            j--;
        }
        if (i < j) {
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
        }
```

```
        }
        arr[r] = arr[i];
        arr[i] = pivot;

        quicksort(arr, l, i - 1);
        quicksort(arr, i + 1, r);
    }
```

5.5.3 Laufzeit

- **worst-case**: Quicksort hat eine Laufzeit von $O(n^2)$, allerdings wird in der Literatur darauf hingewiesen, dass das Sortierverfahren eins der schnellsten Sortierverfahren ist, da es im Mittel mit $O(n\ log\ n)$ sortiert (vgl. [OW17c, 92] sowie [GD18e, 173]).
  Güting und Dieker weisen in [GD18e, 182 f.] darauf hin, dass die **Effizienz** von Quicksort im Wesentlichen von der Wahl des Pivot-Elementes als auch von der *Rekursionstiefe* abhängt.
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

5.6 Merge-Sort

Merge-Sort ist ein **stabiler**, auf Schlüsselvergleichen basierender Sortieralgorithmus, der **nicht in place** sortiert.

5.6.1 Methode

Merge-Sort nutzt **divide-and-conquer**, um ein Feld zu sortieren.
Die wesentliche Arbeit erfolgt hier in den Merge-Schritten, im Gegensatz zu **Quicksort**, bei dem die eigentliche Arbeit im Partitionierungs-Schritt stattfindet (vgl. [GD18e, 174]).
I.d.R. benötigt Merge-Sort linear viel zusätzlichen Speicherplatz (vgl. [OW17c, 112].
Der Algorithmus teilt die Eingabefolge rekursiv in zwei gleich-große Folgen auf, bis n ein-elementige Folgen vorhanden sind (eine ein-elementige Menge gilt als sortiert).
Im Anschluss werden die Folgen miteinander verschmolzen, und zwar so, dass die miteinander verschmolzenen Folgen immer in sortierter Reihenfolge vorliegen.
Das ganze wird so lange wiederholt, bis die Folge mit n Elementen wieder rekonstruiert ist: Sie liegt dann in sortierter Reihenfolge vor (s. Abbildung 5.3).

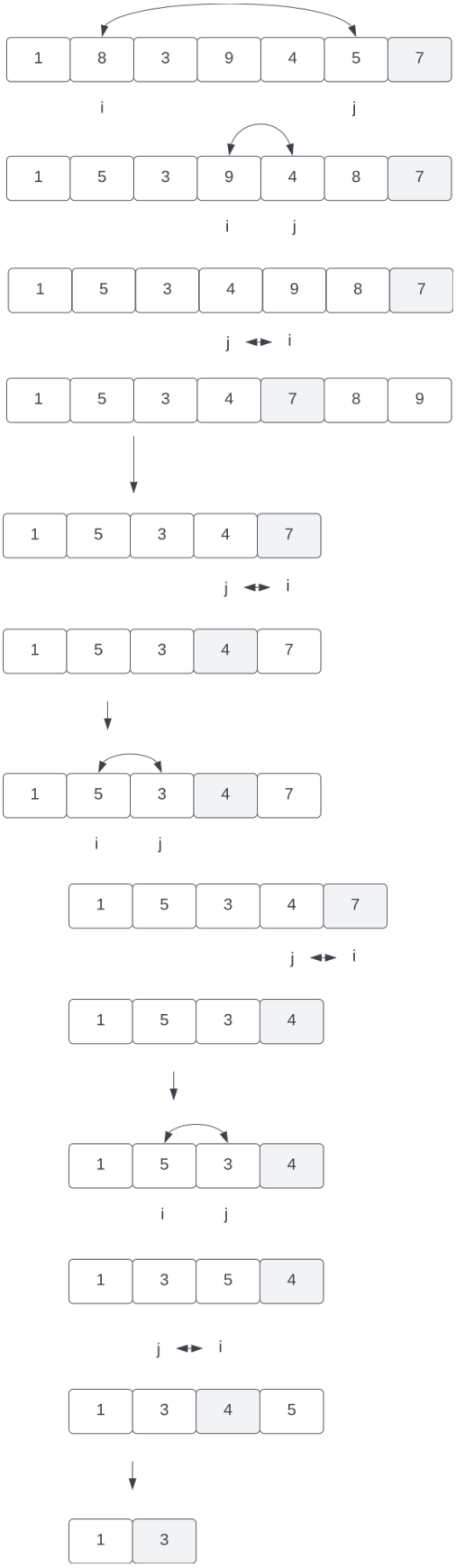

Abb. 5.2: Anwendung von Quicksort auf das Feld 1, 8, 3, 9, 4, 5, 7. Das Pivotelement ist jeweils hervorgehoben. Sobald sich i und j kreuzen, findet eine Partitionierung der Folge statt, und die Methode wird rekursiv auf die Teilfolgen angewendet. Nicht dargestellt sind bereits vollständig sortierte Teilfolgen. (Quelle: eigene)

5.6.2 Implementierung

Das folgende Beispiel beinhaltet nicht die Methoden für die Partitionierung und das Verschmelzen.

```
mergeSort(int[] arr, int start, int end) {
    if (arr.length == 1) {
        return arr;
    }

    int mid = (start + end) / 2;
    int[] left = divide(arr, start, mid);
    int[] right = divide(arr, mid + 1, end);

    left = mergeSort(left, 0, left.length - 1);
    right = mergeSort(right, 0, right.length - 1);

    arr = merge(arr, left, right, start);

    return arr;
}
```

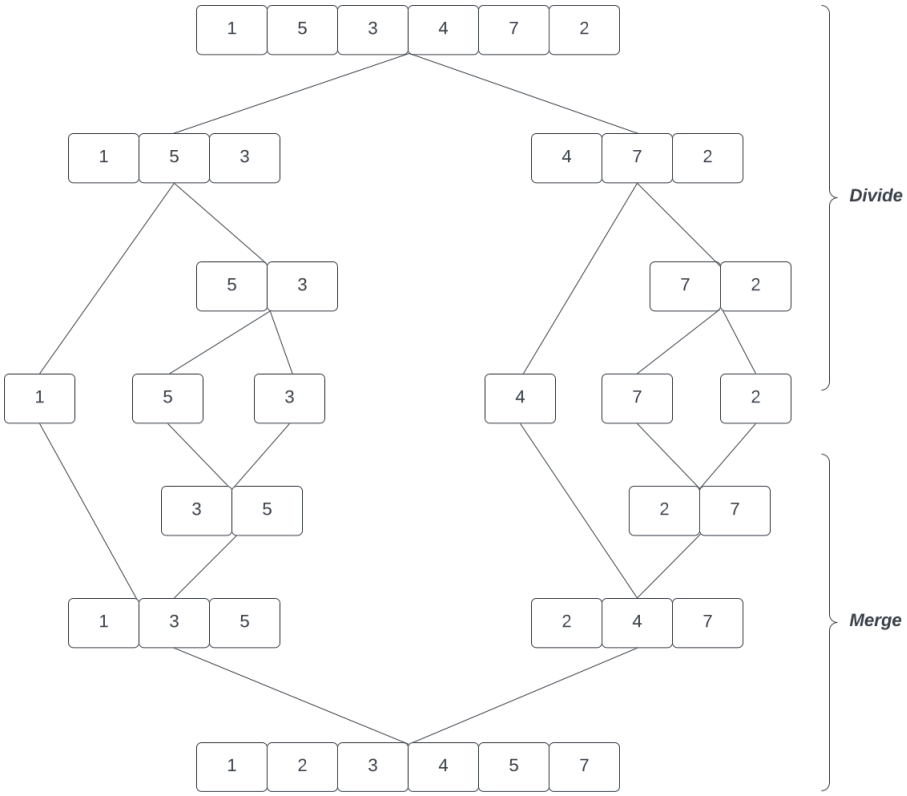

Abb. 5.3: Anwendung von Merge-Sort auf das Feld 1, 5, 3, 4, 7, 2. (Quelle: eigene)

5.6.3 Laufzeit

- **worst-case**: Merge-Sort hat eine Laufzeit von $O(n\ log\ n)$.
  Das Aufteilen des Feldes benötigt $O(log\ n)$, in jedem Schritt werden Operationen mit $O(n)$ Zeit durchgeführt (aufteilen oder verschmelzen) [10]
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

> Die Rekursionstiefe ist bei **Merge-Sort** logarithmisch beschränkt (mit $\lceil log\ n \rceil$), im Gegensatz zu **Quicksort** (vgl. [OW17c, 116]).

5.7 Heapsort

Sortieren funktioniert nach der Annahme, dass die Wurzel eines Heaps immer den größten Schlüssel enthält.
Der Schlüssel kann nun entnommen werden, woraufhin zwei Teail-Heaps entstehen. Der entnommene Schlüssel wird durch den Schlüssel mit dem höchsten Index der Folge ersetzt (die Folge verkleinert sich somit um ein Element) - sollten die Heap-Bedingungen hierdurch verletzt werden, können mittels **top-down reheapify** (**sink**) die Heap-Bedingungen wiederhergestellt werden (vgl. [OW17c, 107 f.]).

Ebenda zeigen **Ottmann und Widmayer**, wie **Heapsort** auch **in place** umgesetzt werden kann: Anstatt immer den größten Schlüssel aus dem Heap zu entfernen und in ein zusätzliches Feld zu schreiben, wird der entfernte Schlüssel einfach an das Ende der Folge geschrieben und bei nachfolgenden Operationen nicht mehr als Teil des Heaps berücksichtigt.

Allerdings müssen in einer Folge von Schlüsseln zunächst Heap-Bedingungen hergestellt werden, damit Verfahren wie **swim** und **sink** (s. Abschnitt 4.7) auf die Folge angewendet werden können. Hierzu findet sich bei *Ottmann und Widmayer* ein einfaches Verfahren[11]:

> Eine gegebene Folge $F = k_1, k_2, \ldots, k_N$ von N Schlüsseln wird in einen Heap umgewandelt, indem die Schlüssel $k_{\lfloor \frac{N}{2} \rfloor}, k_{\lfloor \frac{N}{2} \rfloor - 1}, \ldots, k_1$ (in dieser Reihenfolge) in F versickern. ([OW17c, 111])

Beispiel:
Sei die Folge $F = 1, 4, 2, 5, 12, 8, 9, 6$ mit $N = 8$ gegeben.
Anwendung der o.a. Methode liefert für die einzelnen Schritte:

1. k_4: $1, 4, 2, 6, 12, 8, 9, 5$
2. k_3: $1, 4, 9, 6, 12, 8, 2, 5$

[10] Das Verschmelzen von zwei Teilfeldern erfolgt durch paralleles Durchlaufen mti anschliessendem Einfügen in ein sortiertes Teilfeld, wozu $O(n)$ Zeit benötigt wird (vgl. Skript (Teil 2) "7.4.1 MergeSort").
[11] das Verfahren ist zurückzuführen auf *Floyd*, [Flo64]

3. k_2: $1, 12, 9, 6, 4, 8, 2, 5$
4. k_1: $12, 1, 9, 6, 4, 8, 2, 5$
   $\rightarrow 12, 6, 9, 1, 4, 8, 2, 5$
   $\rightarrow 12, 6, 9, 5, 4, 8, 2, 1$

Da nun ein Heap vorliegt, kann dieser sortiert werden.
Es gilt $k_1 = 12$, $k_{N_F} = 1$, mit $N_F = 8$.
Austauschen von k_1 mit k_{N_F} liefert $1, 6, 9, 5, 4, 8, 2, 12$, N_F ist nun 7, woraus sich eine *unsortierte* und eine *sortierte* Teilfolge ergeben:
$[1, 6, 9, 5, 4, 8, 2], [12]$
Die Heap-Bedingungen werden nun in der unsortierten Folge $k_1, \ldots, k_7$ wiederhergestellt.
Danach werden die Schritte so lange wiederholt, bis $N_F = 1$.

1. **[unsortiert] / [sortiert]**: $[12, 6, 9, 5, 4, 8, 2, 1], []$
2. tauschen: $[1, 6, 9, 5, 4, 8, 2], [12]$
3. *sink* 1: $[9, 6, 8, 5, 4, 1, 2], [12]$
4. tauschen: $[2, 6, 8, 5, 4, 1], [9, 12]$
5. *sink* 2: $[8, 6, 2, 5, 4, 1], [9, 12]$
6. tauschen: $[1, 6, 2, 5, 4], [8, 9, 12]$
7. *sink* 1: $[6, 5, 2, 1, 4], [8, 9, 12]$
8. tauschen: $[4, 5, 2, 1], [6, 8, 9, 12]$
9. *sink*: 4: $[5, 4, 2, 1], [6, 8, 9, 12]$
10. tauschen: $[1, 4, 2], [5, 6, 8, 9, 12]$
11. *sink*: $[4, 1, 2], [5, 6, 8, 9, 12]$
12. tauschen: $[2, 1], [4, 5, 6, 8, 9, 12]$
13. *sink* 2: -
14. tauschen: $[1], [2, 4, 5, 6, 8, 9, 12]$
15. $N_F = 1 \rightarrow F_s = 1, 2, 4, 5, 6, 8, 9, 12$

5.7.1 Laufzeit

Das oben beschriebene Verfahren zum Umwandeln einer Folge der Länge N in einen Heap benötigt $O(N)$ Operationen.
Das Sortieren der Schlüssel benötigt insg. $O(N\ log\ N)$ Zeit (vgl. [OW17c, 112]).

Damit ist **Heapsort** ein **optimales** Sortierverfahren.

- **worst-case**: $O(n\ log\ n)$
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

Anhänge

A

Cheat Sheets

A.1 Algorithmus

Bei der Betrachtung kleinerer Problemgrößen fällt bei der Wahl eines Algorithmus weniger seine Effizienz ins Gewicht, sondern eher die Einfachheit seiner Implementierung und seine Verständlichkeit (vgl. [GD18b, 5 f.]).

So läuft bspw. eine Implementierung von Insertion-Sort ($O(n^2)$), die zur Sortierung $8 * n^2$ Operationen benötigt, für Eingabemengen $n \leq 43$ schneller als eine Implementierung von Merge-Sort ($O(n\ log(n))$), die $64 * (n\ log(n))$ Schritte für die Sortierung benötigt.

Diesbzgl. hatte der Selbsttest "D+A-Selbsttest-02: O-Notation" die Frage gestellt, in welchem Fall die Beurteilung eines Algorithmus bezüglich der Komplexitätsklasse nicht unbedingt angemessen sei. Die richtige Antwort hierzu lautete: "Bei sehr kleinen Eingabemengen".

A.1.1 Optimaler Algorithmus

Optimaler Algorithmus

"Ein Algorithmus heißt (asymptotisch) **optimal**, wenn die obere Schranke für seine Laufzeit mit der unteren Schranke für die Komplexität des Problems zusammenfällt." ([GD18b, 20])

Sortieralgorithmen mit einer Laufzeit von $O(n\ log\ n)$ sind **optimal**, bspw. **Merge-Sort**.

A.1.2 Divide-and-conquer (DAC)

Ottmann und Widmayer formulieren **DAC** wie folgt (vgl. [**OW17a**]):

Divide-and-conquer-Verfahren zur Lösung eines Problems der Größe N

1. *Divide*: Teile das Problem der Größe N in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn $N > 1$ ist; sonst löse das Problem der Größe 1 direkt
2. *Conquer*: Löse die Teilprobleme auf dieselbe Art (rekursiv).
3. *Merge*: Füge die Teillösungen zur Gesamtlösung zusammen.

A.2 Binäre Suche

Die **binäre Suche** folgt der *Divide-and-Conquer*-Strategie.
Hierbei wird in einer aufsteigend sortierten Liste mit n Einträgen nach einem Element mit Schlüssel k gesucht (vgl. [OW17d, 174 f.]).

A.2.1 Methode

1. Ist die Liste leer, war die Suche erfolglos
2. Teile die Liste in 2 (möglichst) gleich-große Teile und betrachte mittlere Position $a[m]$
3. Falls $k < a[m]$, durchsuche die linke Teilliste $0, \ldots, m-1$ nach demselben Verfahren
4. Falls $k > a[m]$, durchsuche die rechte Teilliste $m+1, \ldots, n-1$ nach demselben Verfahren
5. Sonst ist $key = a[m]$ und das gesuchte Element ist gefunden

A.2.2 Laufzeit

- **Laufzeit**: $O(log\ n)$

A.3 Dictionaries

A.3.1 Sequenziell geordnete Liste im Array

A.3.1.1 Laufzeit

- insert / delete: $O(n)$
- member : $O(log\ n)$ (binäre Suche)
- Platzbedarf: $O(n)$

A.3.2 Ungeordnete Liste

A.3.2.1 Laufzeit

- insert (duplikate ok): $O(1)$
- insert (duplikate entfernt): $O(n)$
- delete / member : $O(n)$
- Platzbedarf: $O(n)$

A.3.3 Geordnete Liste

A.3.3.1 Laufzeit

- insert / delete / member: $O(n)$
- Platzbedarf: $O(n)$

A.3.4 Bitvektor

A.3.4.1 Laufzeit

- insert / delete / member: $O(1)$
- Platzbedarf: $O(n)$

A.4 Listen

A.4.1 Laufzeiten

	verkettete Liste	doppelt verkettete Liste	Liste als Array
suchen	$O(n)$	$O(n)$	$O(n)$
einfügen	$O(n)$	$O(1)$	$O(n)$
entfernen	$O(n)$	$O(1)$	$O(n)$

Tabelle A.1: Laufzeiten für die Operationen *suchen*, *einfügen* und *entfernen* bei verschiedenen Listenimplementierungen.

A.4.2 Implementierung

A.4.3 Implementierungsbeispiele

Die folgenden Implementierungsbeispiele orientieren sich an den Praktikumsunterlagen: Dort wurde kein Dummy-Element für den Kopf der Liste verwendet.

- Einfügen eines neuen Elements vor dem Kopf einer einfach verketteten Liste:
```
public void prependHead(Object data) {
    ListElement element = new ListElement(data);
    element.next = head;
    head = element;
}
```

- Einfügen eines neuen Elements hinter einem angegebenen Element:
```
public void insertAt(ListElement ref, Object data) {
    if (ref == null) {
        throw new IllegalArgumentException();
    }

    ListElement element = new ListElement(data);
    element.next = ref.next;
    ref.next = element;
}
```

- Einfügen eines Elements vor einem angegebenen Element:
```
public void insertBefore(ListElement ref, Object data) {
    if (ref == null) {
        throw new IllegalArgumentException();
    }

    if (ref == head) {
        prependHead(data);
        return;
    }
```

```
    ListElement prev = head;
    while (prev != null && prev.getNext() != ref) {
        prev = prev.getNext();
    }
    if (prev == null) {
        throw new IllegalArgumentException(
            "ref does not seem to be in this list"
        );
    }

    ListElement element = new ListElement(data);
    element.next = ref;
    prev.next = element;
}
```

- Entfernen eines Elements:

```
    public void delete(ListElement del) {
        if (del == null) {
            throw new IllegalArgumentException();
        }

        ListElement prev = getPredecessor(del);
        if (prev == null) {
            // kein Vorgänger-Element: del entspricht
            // dem Kopf der Liste
            head = del.next;
            return;
        }

        prev.next = del.next;
    }
```

- Anhängen eines Elements an das Ende (`tail`) einer Liste:

```
public void append(Object data) {
    ListElement element = new ListElement(data);

    if (tail == null) {
        head = element;
        head = tail;
        return;
    }

    tail.next = element;
    tail = element;
}
```

A.5 Binaere (Such-)Baeume

- Der **Grad** eines Knotens ist die Anzahl seiner Nachfolger.
- Ein **innerer Knoten** ist ein Knoten mit Grad ≥ 1.
- Ein Blatt ist ein Knoten mit Grad 0.
- Der **Grad eines Baums** ist der maximale Grad eines Knotens im Baum.
- Die **Tiefe** eines Knotens ist der Abstand des Knotens von der Wurzel.
- Die **Höhe** eines Baumes ist das Maximum der *Tiefe* seiner Knoten.
- Die **maximale Höhe** eines Baumes mit n Knoten ist $n-1$.
- Die **minimale Höhe eines Baumes** mit n Knoten ist $\lceil log_2(n+1) \rceil - 1$.
- Ein *vollständiger* binärer Baum mit n *inneren* Knoten hat $n+1$ Blätter[1].

A.5.1 Einfügen / Suchen / Löschen in binären Suchbäumen

Die Kosten für die Operationen *Einfügen* / *Suchen* / *Löschen* belaufen sich auf $O(h)$, wobei h die *Höhe* des Baumes ist.

h kann zwischen $\lceil log_2\ (n+1) \rceil - 1$ (vollständiger Baum) und $n-1$ liegen, wobei n die Anzahl der Knoten des Baumes ist:
Im worst-case ist ein Baum zu einer linearen Liste entartet[2], so dass alle Knoten durchlaufen werden müssen, um einen Schlüssel für eine nachfolgende *insert-* / *delete*-Operation zu finden.

Im Mittel wird für eine Einfügeoperation $O(log\ n)$ Zeit benötigt[3].

Der Aufwand für den Aufbau eines binären Suchbaumes mit n bereits sortierten Elementen ist $O(n^2)$ (vgl. [GD18a, 235 f.]).

[1] was auch für binäre Bäume gilt, bei denen der Grad eines inneren Knoten mit mindestens 2 festgelegt ist
[2] bspw. weil die einzufügenden Schlüssel schon sortiert vorliegen
[3] vgl. [GD18a, 136 ff.]

A.6 Heap

T ist ein **Heap** genau dann, wenn gilt:

1. T ist ein *links-vollständiger* binärer Baum.
2. für jeden Teilbaum T' von T gilt, dass das Minimum des entsprechenden Teilbaums an der Wurzel steht[4].

[4] gilt für einen *Min-Heap*. Für einen *Max-Heap* gilt, dass das Maximum an der Wurzel steht.

A.7 Hashing

Bei allen dynamischen Anwendungen sollte **offenes Hashing** verwendet werden. **Geschlossenes Hashing** sollte nur dann verwendet werden, wenn die Gesamtzahl der Elemente von vornherein beschränkt ist, keine oder nur wenige Elemente entfernt werden müssen und ein Belegungsgrad von 60% - 70% nicht überschritten wird[5].

A.7.1 Quadratisches Sondieren

> Wenn m eine Primzahl der Form $4i + 3$ ist, dann ist garantiert, dass die Sondierungsfolge eine Permutation der Hashadressen 0 bis $m - 1$ ist [...] ([OW17b, 207])

A.7.2 Löschen von Schlüsseln bei geschlossenem Hashing

Wird ein Schlüssel k gelöscht, wird der Bucket für ein einzufügendes Element als *frei* betrachtet, bei der Suche nach einem Element als *belegt*. Grund: Schlüssel k', die nach k eingefügt wurden, könnten sonst nicht wiedergefunden werden (vgl. [OW17b, 203]).

A.7.3 Einfügen von Schlüsseln bei geschlossenem Hashing

Soll ein Schlüssel k eingefügt werden, wird zunächst die komplette Liste durchsucht, ob der Schlüssel bereits enthalten ist.
Wird dabei eine Speicherzelle getroffen, die als *gelöscht* markiert ist, wird diese zum Einfügen vorgemerkt.

- Ist der Schlüssel k bereits vorhanden, wird er nicht neu eingefügt (keine Duplikate!)
- Ist der Schlüssel k nicht vorhanden, wird er in die erste als gelöscht markierte Speicherzelle eingefügt (sofern durch die Sondierungsfunktion getroffen). Hierdurch wird die *gelöscht*-Markierung aufgehoben.
- ansonsten wird der Schlüssel in das erste freie Feld eingefügt

A.7.4 Laufzeit

- m: Anzahl der Buckets
- n: Anzahl der zu speichernden Einträge

[5] s. Skript (Teil 2) S.141 sowie [GD18a, 126]

A.7.4.1 Offenes Hashing

Die durchschnittliche Listenlänge ist $\frac{n}{m}$.

- *insert* / *delete* / *member*: $O(1 + \frac{n}{m}) = O(\frac{n}{m})$ (falls $n \sim m$: $O(\frac{n}{m}) = 1$)
- Worst-case: alle Schlüssel in einem Bucket mit Listenlänge $n \to O(n)$
- Platzbedarf: $O(n + m)$

A.7.4.2 Geschlossenes Hashing

Abhängig von Lastfaktor[6] und der Wahl der Hashfunktion.

- Best-case: $O(1)$
- Worst-case: $O(n)$

[6] wenn der Lastfaktor 1 ist (alle Buckets belegt) müssen im Worst-Case alle m Buckets nach einem Schlüssel durchsucht werden (vgl. [CL22, 300])

A.8 Sortierverfahren

Güting und Dieker klassifizieren Sortieralgorithmen neben der Effizienz und den angewandten Methoden u.a. nach folgenden Kriterien (vgl. [GD18e, 169 f.]):

- **intern / extern**
  interne Verfahren halten alle Daten im Hauptspeicher, *externe* laden nur einen Teil davon in den Speicher (bspw. beim Sortieren von Daten, die auf Diskette / Platte / Band vorliegen (vgl. [OW17c, 80])).
- **in place (in situ)**
  Sortierverfahren, die keinen zusätzlichen Speicherplatz zum Sortieren der Daten benötigen, sortieren place.
  Die Eingangsfolge und die Ausgangsfolge werden im selben Array dargestellt, Vertauschungen von Schlüsseln erfolgen in genau diesem Array.
  Alle in diesem Kapitel behandelten Sortierverfahren fallen in diese Kategorie, bis auf **Merge-Sort**.
- **stabil**
  Sortierverfahren, bei denen die Ergebnisfolge gleicher Schlüssel dieselbe ist wie die Ausgangsfolge, nennt man **stabil**:
  > [...] die Reihenfolge von Elementen mit gleichem Sortierschlüssel wird während des Sortierverfahrens nicht vertauscht. ([OW17c, 164]).

  Die Sortierverfahren **Bubblesort**, **Insertion-Sort** und **Merge-Sort** sind allesamt stabile Sortierverfahren.
  Nicht stabil sind **Selection-Sort** und **Quicksort**.
- **natürlich**
  natürliche Sortierverfahren arbeiten bei vorsortierten Daten schneller als bei unsortierten (bspw. **Insertion-Sort**, wohingegen **Selection-Sort** i.d.R. auch bei vorsortierten Daten die gleiche Laufzeitkomplexität aufweist, wie bei unsortierten)

A.8.1 Untere Schranke

Jedes allgemeine Sortierverfahren benötigt zum Sortieren von N verschiedenen Schlüsseln sowohl im schlechtesten Fall als auch im Mittel wenigstens $\Omega(N\ log\ N)$ Schlüsselvergleiche.

A.9 Bubblesort

Sortieren durch direktes Austauschen

A.9.1 Eigenschaften

- stabil
- in place

A.9.2 Methode

Bubblesort vergleicht zwei benachbarte Elemente und vertauscht diese miteinander, falls ihre relative Reihenfolge nicht der erwarteten Sortierreihenfolge entspricht.
Nach dem ersten Durchlauf steht der größte Schlüssel am Ende des Feldes.
Dann wird der Vorgang wiederholt, es muss aber nicht mehr mit dem Schlüssel an Position $n-1$ verglichen werden, da dieser Schlüssel bereits seine endgültige Position in dem sortierten Feld eingenommen hat.
Das wird so lange wiederholt, bis keine Vertauschungen mehr aufgetreten sind oder die Schleifen komplett durchlaufen sind.

A.9.3 Implementierung

```
boolean inversed = true;
for (i = n - 1; i >= 0; i--) {
    inversed = false;
    for (j = 0; j < i; j++) {
        if (arr[j] > arr[j + 1]) {
            swap(arr, j, j+1);
            inversed = true;
        }
    }
    if (!inversed) {
        break;
    }
}
```

A.9.4 Laufzeit

- **Anzahl der Vergleiche und Vertauschungen**: $\frac{n*(n-1)}{2}$ $(O(n^2))$
- **worst-case**: $O(n^2)$
- **average-case**: $O(n^2)$
- **best-case**: $O(n)$

A.10 Selection-Sort

Sortieren durch Auswahl

A.10.1 Eigenschaften

- nicht stabil
- in place

A.10.2 Methode

Das Sortierverfahren vergleicht zwei Teilfolgen des zu sortierenden Feldes miteinander.
Hierbei ist eine Teilfolge sortiert, die andere unsortiert. Die sortierte Teilfolge ist zunächst leer.
Das jeweils kleinste Element aus der *unsortierten* Teilfolge wird an das Ende der sortierten Teilfolge eingefügt.
Das ganze wird so lange wiederholt, bis die unsortierte Teilfolge nur noch aus einem Element besteht, danach ist das Feld sortiert.

A.10.3 Implementierung

```
for (int i = 0; i < n - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < n; j++) {
        if (arr[j] < arr[minIndex]) {
            minIndex = j;
        }
    }
    int tmp = arr[minIndex];
    arr[minIndex] = arr[i];
    arr[i] = tmp;
}
```

A.10.4 Laufzeit

- **Anzahl der Vergleiche**: $\frac{n*(n-1)}{2}$
- **Vertauschungen**: $O(n)$
- **worst-case**: $O(n^2)$
- **average-case**: $O(n^2)$
- **best-case**: $O(n^2)$

A.11 Insertion-Sort

Sortieren durch Einfügen

A.11.1 Eigenschaften

- stabil
- in place

A.11.2 Methode

Bei **Insertion-Sort** wird das zu sortierende Feld in zwei Teilfolgen unterteilt - eine unsortierte und eine sortierte.
Die sortierte Folge enthält zu Beginn nur ein Element.
Aus der unsortierten Teilfolge werden Elemente der Reihe nach entnommen und in die sortierte Folge an der richtigen Stelle eingefügt, wodurch Schlüssel ihre Position in der sortierten Folge ggf. ändern müssen.

A.11.3 Implementierung

```
for (int i = 1; i < n; i++) {
    int min = arr[i];
    int j = i;
    while (j > 0 && arr[j - 1] > min) {
        arr[j] = arr[j - 1];
        j--;
    }
    arr[j] = min;
}
```

A.11.4 Laufzeit

- **Anzahl der Vergleiche und Vertauschungen**: $\frac{n*(n-1)}{2}$
- **worst-case**: $O(n^2)$
- **average-case**: $O(n^2)$
- **best-case**: $O(n)$

A.12 Quicksort

A.12.1 Eigenschaften

- nicht stabil
- in place

A.12.2 Methode

Bei **Quicksort** wird ein Schlüssel aus dem Feld als **Pivot-Element** p gewählt (bspw. das rechte einer Teilfolge).
Alle Schlüssel, die kleiner als das Pivotelement sind, werden in die eine Hälfte geschrieben, alle die größer sind, in die andere. Das Pivotelement selber wird zwischen die beiden Hälften geschrieben, die dann durch Rekursion wiederholt bearbeitet werden, bis die Teilfelder die Größe 1 haben, womit das gesamte Feld als sortiert gilt.

A.12.3 Implementierung

```
quicksort (int[] arr, int l, int r) {
    if (l >= r) {
        return;
    }
    int i = l;
    int j = r - 1;
    int pivot = arr[r];
    while (i <= j) {
        while (i < r && arr[i] <= pivot) {
            i++;
        }
        while (j >= 0 && arr[j] > pivot) {
            j--;
        }
        if (i < j) {
            int tmp = arr[i];
            arr[i] = arr[j];
            arr[j] = tmp;
        }
    }
    arr[r] = arr[i];
    arr[i] = pivot;

    quicksort(arr, l, i - 1);
    quicksort(arr, i + 1, r);
}
```

A.12.4 Laufzeit

- **worst-case**: $O(n^2)$
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

A.12.5 Anmerkungen

Quicksort nutzt **divide-and-conquer**, um ein Feld zu sortieren.
Die wesentliche Arbeit erfolgt hier in den Partitionierungs-Schritten.

Im **worst-case** hat Quicksort eine Laufzeit von $O(n^2)$, allerdings wird in der Literatur darauf hingewiesen, dass das Sortierverfahren eins der schnellsten Sortierverfahren ist, da es im Mittel mit $O(n\ log\ n)$ sortiert.

Die **Effizienz** von Quicksort hängt im Wesentlichen von der Wahl des Pivot-Elementes als auch von der *Rekursionstiefe* ab.

A.13 Merge-Sort

A.13.1 Eigenschaften

- stabil
- nicht in place
- optimal

A.13.2 Methode

Der Algorithmus teilt die Eingabefolge rekursiv in zwei gleich-große Folgen auf, bis n ein-elementige Folgen vorhanden sind.
Im Anschluss werden die Folgen miteinander verschmolzen, und zwar so, dass die miteinander verschmolzenen Folgen in sortierter Reihenfolge vorliegen.

A.13.3 Implementierung

```
mergeSort(int[] arr, int start, int end) {
    if (arr.length == 1) {
        return arr;
    }

    int mid = (start + end) / 2;
    int[] left = divide(arr, start, mid);
    int[] right = divide(arr, mid + 1, end);

    left = mergeSort(left, 0, left.length - 1);
    right = mergeSort(right, 0, right.length - 1);

    arr = merge(arr, left, right, start);

    return arr;
}
```

A.13.4 Laufzeit

- **worst-case**: $O(n\ log\ n)$
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

A.13.5 Anmerkungen

Der Baum hat eine Höhe von $\sim log_2\ n$, auf jeder Ebene werden $O(n)$ Operationen (*merge* oder *divide*) durchgeführt.
Die Rekursionstiefe ist bei **Merge-Sort** logarithmisch beschränkt mit $\lceil log\ n \rceil$, im Gegensatz zu **Quicksort**.

A.14 Heapsort

A.14.1 Eigenschaften

- nicht stabil
- in place
- optimal

A.14.2 Methode

Die Eingabefolge wird in einen Max-Heap umgewandelt.
Da in einem Max-Heap immer der größte Schlüssel am Anfang der Folge steht, wird der Schlüssel mit dem letzten Element der Folge ausgetauscht und die Heap-Größe um 1 verkleinert.
Das an Position 1 stehende Element wird nun über ein **top-down reheapify** in dem Heap an seine eigentliche Position gebracht, damit die Heap-Bedingungen hergestellt sind und das größte Element wieder an Position 1 steht.
Das wird so lange wiederholt, bis der Heap nur noch aus einem Element besteht, dann ist die resultierende Folge sortiert.

Analog läßt sich das Verfahren auf einen **Min-Heap** anwenden, bei dem der kleinste Schlüssel immer am Anfang der Folge steht: Die Heap-Bedingungen sind hier, dass die Schlüssel der direkten Nachfolger eines Knotens größer oder gleich als der Schlüssel des Knotens selber sind.
Eine Vorsortierung der Daten ändert nichts an der Laufzeit von Heapsort (vgl. [OW17c, 112]). Heapsort ist also *nicht* natürlich[7].

A.14.3 Laufzeit

- **Aufbau des Heaps**: $O(n)$
- **sink**: n Aufrufe, jeweils $O(log\ n)$
- **worst-case**: $O(n\ log\ n)$
- **average-case**: $O(n\ log\ n)$
- **best-case**: $O(n\ log\ n)$

[7] ebenda verweisen *Ottmann und Widmayer* auf **Smoothsort**, eine Variante von Heapsort, die auf vorsortierten Daten schneller arbeitet und $O(n)$ erreichen kann

Literaturverzeichnis

[Blo17] Joshua Bloch. Effective Java, 3rd Edition. Addison-Wesley Professional, 2017. ISBN: 978-0134686097.

[CK75] B. J. Cornelius und G. H. Kirby. „Depth of Recursion and the Ackermann Function“. In: BIT Numerical Mathematics 15.2 (Juni 1975), S. 144–150. ISSN: 1572-9125. DOI: 10.1007/BF01932687.

[CL22] Thomas H Cormen und Charles E Leiserson. Introduction to Algorithms, fourth edition. en. London, England: MIT Press, Apr. 2022. ISBN: 9780262046305.

[Flo64] Robert W. Floyd. „Algorithm 245: Treesort“. In: Commun. ACM 7.12 (Dez. 1964), S. 701. ISSN: 0001-0782. DOI: 10.1145/355588.365103. URL: https://doi.org/10.1145/355588.365103.

[GD18a] Ralf Hartmut Güting und Stefan Dieker. „Datentypen zur Darstellung von Mengen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 109–167. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_4. URL: https://doi.org/10.1007/978-3-658-04676-7_4.

[GD18b] Ralf Hartmut Güting und Stefan Dieker. „Einführung“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 1–38. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_1. URL: https://doi.org/10.1007/978-3-658-04676-7_1.

[GD18c] Ralf Hartmut Güting und Stefan Dieker. „Grundlegende Datentypen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 63–107. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_3. URL: https://doi.org/10.1007/978-3-658-04676-7_3.

[GD18d] Ralf Hartmut Güting und Stefan Dieker. „Programmiersprachliche Konzepte für Datenstrukturen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 39–61. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_2. URL: https://doi.org/10.1007/978-3-658-04676-7_2.

[GD18e] Ralf Hartmut Güting und Stefan Dieker. „Sortieralgorithmen“. In: Datenstrukturen und Algorithmen. Wiesbaden: Springer Fachmedien Wiesbaden, 2018, S. 169–200. ISBN: 978-3-658-04676-7. DOI: 10.1007/978-3-658-04676-7_5. URL: https://doi.org/10.1007/978-3-658-04676-7_5.

[HW01] Daniel M. Hoffman und David M. Weiss, Hrsg. Software fundamentals: collected papers by David L. Parnas. USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201703696.

[Mar03] Robert Cecil Martin. Agile Software Development: Principles, Patterns, and Practices. USA: Prentice Hall PTR, 2003. ISBN: 0135974445.

[Mar08] Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. 1. Aufl. USA: Prentice Hall PTR, 2008. ISBN: 0132350882.

[Oes05] Bernd Oestereich. Analyse und Design mit UML 2. Oldenbourg Wissenschaftsverlag GmbH, 2005. ISBN: 3486576542.

[OW17a] Thomas Ottmann und Peter Widmayer. „Bäume“. In: Algorithmen und Datenstrukturen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 259–402. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_5. URL: https://doi.org/10.1007/978-3-662-55650-4_5.

[OW17b] Thomas Ottmann und Peter Widmayer. „Hashverfahren“. In: Algorithmen und Datenstrukturen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 191–258. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_4. URL: https://doi.org/10.1007/978-3-662-55650-4_4.

[OW17c] Thomas Ottmann und Peter Widmayer. „Sortieren“. In: Algorithmen und Datenstrukturen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 79–165. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_2. URL: https://doi.org/10.1007/978-3-662-55650-4_2.

[OW17d] Thomas Ottmann und Peter Widmayer. „Suchen“. In: Algorithmen und Datenstrukturen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, S. 167–189. ISBN: 978-3-662-55650-4. DOI: 10.1007/978-3-662-55650-4_3. URL: https://doi.org/10.1007/978-3-662-55650-4_3.

[PW76] D. L. Parnas und H. Würges. „Response to undesired events in software systems“. In: Proceedings of the 2nd International Conference on Software Engineering. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, S. 437–446.

[SW11] Robert Sedgewick und Kevin Wayne. Algorithms, 4th Edition. Addison-Wesley, 2011, S. I–XII, 1–955. ISBN: 978-0-321-57351-3.

[Ull23] Christian Ullenboom. Java ist auch eine Insel: Das umfassende Handbuch, 17. Auflage. Galileo Computing, 2023. ISBN: 978-3-8362-9544-4.