

Informatik verständlich unterrichten

Vollständiges Lehrskript auf Deutsch

90 Seiten für die gemeinsame Arbeit mit einem Schüler: vom ersten Java-Ausdruck bis zu Suchbäumen und Graphalgorithmen.

Was dieses Skript abdeckt

Die Grundlage bilden 16 Klausuren und die 16 dazu erstellten Lösungen. Die daraus gewonnene Themenliste umfasst 37 Bereiche und 368 Unterpunkte. Jeder Unterpunkt erhält hier eine eigene Erklärung; die Kennungen A01 bis B19 ermöglichen den direkten Abgleich mit der Themenübersicht.

Die 82 Themenseiten verbinden Begriffe, Funktionsweisen und Begründungen. Kleine Beispiele machen die Regeln überprüfbar. Ein typischer Fehler und eine Verständnisfrage mit Antwort helfen dabei, während des Unterrichts festzustellen, ob der Gedankengang verstanden wurde.

So liest du eine Themenseite

Beginne mit dem Lernziel und lies die Erklärungen abschnittsweise. Rechne das Beispiel gemeinsam durch oder verfolge die Zustände eines Programms. Lass den Schüler danach die Kontrollfrage beantworten, bevor die kurze Antwort verglichen wird. Für längere selbstständige Übungen stehen die ursprünglichen Klausuraufgaben zur Verfügung.

Die Markierung „Ergänzung“ bezeichnet eine Präzisierung oder einen Randfall aus den Lösungen. Solche Hinweise vertiefen den Kernstoff und erklären, warum eine scheinbar naheliegende Lösung manchmal scheitert. Aufgabenabhängige Konventionen werden ausdrücklich benannt.

Arbeitsgrundlage und Beispiele

Die Erklärungen orientieren sich an den vorhandenen Aufgaben, nicht an einer bestimmten Unterrichtsstundenzahl. Viele Zahlenbeispiele sind bewusst klein und dienen der Veranschaulichung. Java-Ausschnitte zeigen den wesentlichen Teil einer Lösung; fehlende Klassenhüllen oder vorausgesetzte Daten sind jeweils aus dem Zusammenhang zu ergänzen.

Zu jedem Abschnitt nennt die Fußzeile passende Klausur-IDs. Die genaue Zuordnung zu Teilaufgaben steht in der bereits erstellten Themenübersicht; die Dateinamen aller Klausuren und Lösungen findest du auf den Seiten 89 und 90.

Programmierung mit Java

Die Einträge sind anklickbar. Jeder Bereich enthält kurze Erklärungen, Beispiele und Fragen zum gemeinsamen Wiederholen.

A01 Java: Programmaufbau, Syntax und Ausführung	5-6
A02 Zahlensysteme und binäre Zahlendarstellungen	7-8
A03 Datentypen, Operatoren und sichere Auswertung	9-11
A04 Kontrollstrukturen, Konsoleneingabe und Ablaufspuren	12-13
A05 Zahlenalgorithmen, Häufigkeiten und Auswahl von Extremwerten	14-16
A06 Stringalgorithmen: Filtern, Wiederholen, Suchen und Löschen	17-19
A07 Arrays: Grundlagen, Änderungen an Ort und Stelle und Messdaten	20-22
A08 Matrizen, Bildtransformationen und zweidimensionale Muster	23-25
A09 Klassen, Konstruktoren, Kennungen und Zugriffsrechte	26-27
A10 Abstraktion, Interfaces, Vererbung und Polymorphie	28-29
A11 Referenztypen, Methodenauflösung und statische Typfehler	30-31
A12 Speicherabbilder, Referenzen und objektorientierte Ablaufanalyse	32-33
A13 UML und allgemeine Assoziationsverwaltung	34-35
A14 UML-Modelle: InternetRadio, Lager und MediaPlayer	36-37
A15 UML-Modelle: Garten, Regal und Gebäudeflächen	38-39
A16 UML-Modell: Putzplan, Fenster und Reinigungskosten	40-41
A17 UML-Modell: Veröffentlichungen und Themengebiete	42-43
A18 Polymorphe Anwendungsmodelle und Text-/Zahlenausgaben	44-45

Algorithmen und Datenstrukturen

Die Einträge sind anklickbar. Jeder Bereich enthält kurze Erklärungen, Beispiele und Fragen zum gemeinsamen Wiederholen.

B01 Algorithmusbegriff und Divide-and-Conquer	46-47
B02 Asymptotische Laufzeiten und Aufwand von Operationen	48-49
B03 Schleifenanalyse, Rekursion und Rekursionsspeicher	50-52
B04 Java-Implementierung, Referenzen und Vergleiche	53-54
B05 List, Set, Map und allgemeine Operationen verketteter Listen	55-57
B06 Prioritätslisten, Warteschlangen und Stapel	58-59
B07 Binärbäume, Suchbäume und Traversierungen	60-62
B08 AVL-Bäume und Balancebedingungen	63-64
B09 Binäre Heaps und Prüfung der Heapordnung	65-66
B10 B-Bäume: Suche, Ausgabe, Höhe und Ordnung	67-68
B11 Lineare, binäre und Interpolationssuche	69-70
B12 Hash-Tabellen und Kollisionsauflösung	71-72
B13 Sortierverfahren und Vergleich ihrer Laufzeiten	73-74
B14 Cocktail-Sort und Gnome-Sort schrittweise ausführen	75-76
B15 Graphdarstellung, Veränderungen und Zusammenhang	77-78
B16 Topologische Sortierung und gerichtete Kreisfreiheit	79-80
B17 Breitensuche mit Warteschlange	81-82
B18 Dijkstra und Rekonstruktion kürzester Wege	83-84
B19 Minimale Spannbäume, Kruskal und Union-Find	85-86

Nachschlagen

87: Java-Kurzreferenz • 88: Laufzeiten und Verfahrenswahl • 89-90: Aufgaben- und Lösungsdateien

Vom Verstehen zum selbstständigen Lösen

1. Vorwissen kurz sichtbar machen

Lass den Schüler zuerst einen kleinen Ausdruck auswerten, eine Schleife von Hand verfolgen und erklären, was eine Referenz ist. Dabei werden die nötigen Grundlagen schnell sichtbar. Beginne bei Unsicherheiten mit den passenden A-Kapiteln, bevor mehrere unbekannte Konzepte gleichzeitig eingeführt werden.

2. Einen überschaubaren Schritt erklären

Nimm pro Arbeitsabschnitt wenige zusammengehörige Unterpunkte. Eine gute Erklärung benennt die Eingabe, die gewünschte Ausgabe und den Grundgedanken der Verarbeitung. Der Schüler sollte danach den Ablauf mit eigenen Worten erklären können, bevor er ihn in Java aufschreibt.

3. Das Beispiel aktiv bearbeiten

Decke die nächste Tabellenzeile oder den nächsten Programmschritt zunächst ab. Lass Werte, Referenzen oder Kantenentscheidungen vorhersagen und überprüfe sie anschließend. Gerade bei Rekursion, dynamischer Bindung und Graphverfahren ist die Begründung einer Zustandsänderung wichtiger als das bloße Abschreiben der Endlösung.

4. Einen Randfall bewusst ändern

Verändere eine Eingabe: ein leeres Array, eine fehlende Referenz, eine volle Sammlung, ein gleicher Schlüssel oder eine schon sortierte Folge. Der Schüler begründet zuerst, ob die Vorbedingungen noch gelten. Erst danach wird entschieden, ob der Algorithmus angepasst werden muss oder der Fall vom Vertrag ausgeschlossen ist.

5. Zu einer Klausuraufgabe übergehen

Wähle aus den genannten Klausur-IDs eine passende Aufgabe. Lass den Schüler zunächst selbst einen Lösungsansatz entwickeln und nutze die vorhandene Lösung zum begründeten Vergleich. Eine abweichende Implementierung kann korrekt sein, wenn sie dieselben Anforderungen erfüllt und die erlaubten Sprachmittel verwendet.

6. Das Gelernte abrufen

Beende den Abschnitt mit einer kurzen Erklärung ohne Vorlage: Was ist die Grundidee, welche Voraussetzung ist nötig und wo liegt ein typischer Fehler? Wiederhole schwierige Themen in einer späteren Sitzung anhand eines anderen kleinen Beispiels. Die Selbsttests im Skript lassen sich dafür erneut verwenden.

Sinnvolle Reihenfolge

Java-Grundlagen → Zahlen, Strings und Arrays → Objektorientierung und UML → Laufzeitanalyse und Rekursion → Listen und Bäume → Suchen und Sortieren → Graphalgorithmen. Die Reihenfolge darf an den vorhandenen Kenntnisstand angepasst werden.

A01 Java: Programmaufbau, Syntax und Ausführung

Namen, Methoden und Fehlerarten

Du lernst, Java-Quelltext auf zulässige Bezeichnungen und Methodendeklarationen zu prüfen und dabei Übersetzbarkeit von fachlicher Richtigkeit zu unterscheiden.

A01.01 Bezeichner richtig lesen

Ein Bezeichner benennt beispielsweise eine Klasse, eine Methode oder eine Variable. Java unterscheidet Groß- und Kleinschreibung, sodass `summe` und `Summe` zwei verschiedene Namen sind. Reservierte Schlüsselwörter können nicht als gewöhnliche Bezeichner dienen; Leerzeichen und Bindestriche trennen Zeichenfolgen beziehungsweise bilden andere Sprachelemente.

A01.02 Konvention ist kein Sprachgesetz

Die übliche Schreibweise beginnt Klassennamen groß und Variablen- sowie Methodennamen klein. Eine Abweichung davon ist nicht automatisch ein Syntaxfehler, sofern der Name sprachlich zulässig bleibt. In Prüfungsfragen muss deshalb zwischen erlaubtem Namen und gut lesbarer Benennung unterschieden werden; eine Konvention ersetzt keine Grammatikregel.

A01.03 Methodenkopf und Rückgabe

Der Methodenkopf legt Namen, Parameter und genau einen Rückgabebetyp fest. Eine Methode mit `int` als Rückgabebetyp muss bei normaler Rückkehr einen passenden Wert liefern; `void` bezeichnet eine Methode ohne Ergebniswert. Eine Hilfsklasse braucht keine eigene `main`-Methode, weil ihre Methoden von anderem Programmcode aufgerufen werden können.

A01.04 Syntax und beabsichtigte Bedeutung

Ein Syntaxfehler verletzt die Formregeln der Sprache und verhindert die Übersetzung, beispielsweise die Kombination `void int` vor einem Methodennamen. Ein Programm kann dagegen syntaktisch korrekt sein und fachlich etwas Falsches berechnen. Wenn eine geforderte Addition tatsächlich subtrahiert, liegt der Fehler im beschriebenen Verhalten, nicht in der Deklarationsform.

Beispiel: Eine kleine Rechenmethode

Die Methode besitzt zwei `int`-Parameter und liefert einen `int`-Wert zurück. Ihr Name beschreibt die beabsichtigte Operation. Ersetzt man im Methodenrumpf das Pluszeichen durch ein Minuszeichen, bleibt die Methode übersetzbar, erfüllt aber ihre Beschreibung nicht mehr. Schreibt man dagegen vor dem Namen gleichzeitig `void` und `int`, ist bereits die Deklaration unzulässig. Dieser Vergleich trennt zwei Fehlerklassen, die bei einer bloßen Sichtprüfung leicht vermischt werden.

```
public static int addiere(int a, int b) {  
    return a + b;  
}
```

Typischer Fehler: Ungewöhnliche Großschreibung allein ist kein Syntaxfehler. Ein inkompatibler Rückgabewert ist dagegen ein Typfehler und verhindert die Übersetzung.

Verständnisfrage: Warum benötigt die Klasse einer solchen Hilfsmethode nicht zwingend eine `main`-Methode?

Antwort: Die Methode kann von einer anderen Klasse aus aufgerufen werden. `main` wird nur als Startpunkt benötigt, wenn gerade diese Klasse unmittelbar gestartet werden soll.

A01 Java: Programmaufbau, Syntax und Ausführung

Vom Quelltext zur laufenden Java-Anwendung

Du ordnest die Stationen der Java-Ausführung ein und prüfst Quelldateien, Laufzeitannahmen und mögliche Fehler im Aufgabenblatt getrennt.

A01.05 Übersetzen und ausführen

Der Java-Compiler übersetzt Quelltext normalerweise in Bytecode. Diesen führt eine Java Virtual Machine aus; Bytecode ist daher nicht einfach der native Maschinencode des Prozessors. Ein größeres Programm kann aus vielen Quelldateien bestehen, deren Klassen während Übersetzung und Ausführung gemeinsam verwendet werden.

A01.06 Welche Teile sind plattformabhängig?

Dasselbe geeignete Bytecodeprogramm kann auf verschiedenen Systemen laufen, wenn dort eine passende JVM und seine benötigten Bibliotheken vorhanden sind. Die JVM selbst muss zum jeweiligen Betriebssystem und Prozessor passen. Ein rein interpretierendes Modell benötigt zusätzliche Arbeit zur Verarbeitung der Bytecodebefehle, beschreibt aber nicht jede reale Ausführung vollständig.

A01.07 JIT als Präzisierung · Ergänzung

Eine JVM kann häufig ausgeführten Bytecode während der Laufzeit in nativen Maschinencode übersetzen. Diese Just-in-time-Kompilierung verringert den Aufwand wiederholter Interpretation. Aus der bloßen Existenz von Bytecode folgt deshalb keine allgemeine Aussage, dass jedes Java-Programm unter allen Bedingungen langsamer als ein anders übersetztes Programm sein müsse.

A01.08 Dateiname und öffentlicher Typ · Ergänzung

Beim üblichen Arbeiten mit javac wird ein öffentlicher oberster Typ in einer entsprechend benannten Java-Datei gespeichert. Die öffentliche Klasse Rechner gehört somit in Rechner.java. Mehrere öffentliche Klassen aus einem Aufgabenblatt werden vor dem Übersetzen auf solche Dateien verteilt; ihre Darstellung nebeneinander auf Papier ist keine Dateistruktur.

A01.09 Fehler der Vorlage kennzeichnen · Ergänzung

Nicht jede Unstimmigkeit stammt aus der eigenen Lösung: Ein Aufgabenblatt kann falsche Zeilennummern oder unterschiedlich geschriebene Methodennamen enthalten. Prüfe den tatsächlichen Text und beschreibe eine nötige Korrektur ausdrücklich. Eine korrigierte Fassung darf nicht stillschweigend als die unveränderte Originalfassung behandelt werden.

Beispiel: Zwei Klassen bilden ein Programm

Die Datei Start.java enthält die öffentliche Klasse Start und deren main-Methode. Rechner.java enthält eine öffentliche Hilfsklasse, deren Rechenmethode von Start benutzt wird. Beide werden zu Bytecode übersetzt. Beim Start ist nur Start der Einstiegspunkt; Rechner braucht deshalb keine zweite main-Methode. Die konkrete JVM übernimmt danach die Ausführung der beteiligten Klassen auf dem jeweiligen Rechner.

Typischer Fehler: Java-Compiler, Bytecode und JVM sind verschiedene Bestandteile. Ein Interpreter in der JVM interpretiert nicht unmittelbar den ursprünglichen Java-Quelltext.

Verständnisfrage: Welche Aussage bleibt beim Wechsel von einem Betriebssystem auf ein anderes grundsätzlich richtig?

Antwort: Der Bytecode kann unverändert nutzbar bleiben, die eingesetzte JVM muss jedoch zum neuen System passen; vorausgesetzte Bibliotheken müssen dort ebenfalls verfügbar sein.

A02 Zahlensysteme und binäre Zahlendarstellungen

Stellenwerte und Zweierkomplement sicher umrechnen

Du wandelst ganze Zahlen zwischen Zahlensystemen um und kontrollierst negative Bitmuster mit einer unabhängigen Rückrechnung.

A02.01 Stellen tragen Gewichte

In einem Stellenwertsystem multipliziert jede Ziffer eine Potenz der Basis. Binär sind die Gewichte von rechts eins, zwei, vier und so weiter; hexadezimal wachsen sie mit dem Faktor sechzehn. Eine Hexadezimalziffer entspricht genau vier Bits, wodurch sich beide Darstellungen stellenweise ineinander übertragen lassen.

A02.02 Division liefert die Ziffern

Bei wiederholter ganzzahliger Division durch die Zielbasis ist jeder Rest die nächste Ziffer, beginnend rechts. Deshalb werden die erhaltenen Reste anschließend rückwärts gelesen. Führende 0-Ziffern ändern den Zahlenwert nicht, wohl aber eine ausdrücklich vorgegebene Darstellungsbreite, etwa bei einem achtbittigen Speicherwort.

A02.03 Addieren mit Übertrag

Bei der hexadezimalen Addition werden Ziffern gleicher Stelle zusammen mit einem vorhandenen Übertrag addiert. Erreicht die Summe sechzehn, bleibt ihr Rest modulo sechzehn an dieser Stelle stehen; der ganzzahlige Quotient wird weitergetragen. Das Verfahren entspricht der gewohnten Dezimaladdition, lediglich die Basis ändert sich.

A02.04 Negative Bitmuster bilden

Lege zuerst die Bitbreite fest und schreibe den positiven Betrag mit genau dieser Breite auf. Für die negative Darstellung invertierst du sämtliche Bits und addierst eins; ein Übertrag außerhalb der gewählten Breite wird verworfen. Das Verfahren ist nur sinnvoll, wenn die gewünschte Zahl im vorzeichenbehafteten Wertebereich liegt.

A02.05 Ergebnis unabhängig kontrollieren

Im Zweierkomplement hat das höchste Bit ein negatives Gewicht; die übrigen Stellen behalten ihre positiven Zweierpotenzgewichte. Eine zweite Kontrolle bildet bei acht Bits die Differenz zwischen 256 und dem Betrag. Beide Rechnungen müssen zum gleichen Bitmuster führen, sofern die negative Zahl darstellbar ist.

Beispiel: Eine positive und eine negative Darstellung

Die Dezimalzahl 45 zerfällt in $32 + 8 + 4 + 1$ und lautet binär 00101101. Die beiden Vierergruppen ergeben hexadezimal 2D. Für -45 werden bei acht Bits zunächst 11010010 und nach Addition von eins 11010011 gebildet. Dessen Wert ist $-128 + 64 + 16 + 2 + 1 = -45$. Die alternative Kontrolle $256 - 45$ ergibt 211, dessen vorzeichenloses Bitmuster ebenfalls 11010011 ist.

Typischer Fehler: Invertieren ohne die vorher festgelegte Bitbreite kann zu einer falschen negativen Darstellung führen; ergänze erforderliche führende 0-Bits zuerst.

Verständnisfrage: Weshalb reichen beim Übertragen einer Hexadezimalzahl immer Vierergruppen von Bits?

Antwort: Vier Bits besitzen sechzehn mögliche Belegungen. Genau diese Zahl von Ziffernwerten umfasst eine einzelne Hexadezimalstelle, einschließlich der Ziffer 0.

A02 Zahlensysteme und binäre Zahlendarstellungen

Ganzzahlgrenzen und normalisierte Fließkommazahlen

Du unterscheidest Bitmuster, numerischen Datentyp und Fließkommaformat und erkennst, weshalb nicht jede Dezimalzahl exakt gespeichert wird.

A02.06 Feste Breite begrenzt Zahlen

Ein vorzeichenbehaftetes Zweierkomplement mit b Bits umfasst die Werte von $-2^{(b-1)}$ bis $2^{(b-1)}-1$. Acht Bits reichen daher von -128 bis 127, während Java-int 32 Bits besitzt. Die negative Seite enthält einen zusätzlichen Wert, weil das Bitmuster der 0 keinen zweiten Vorzeichenfall benötigt.

A02.07 Der Literaltyp zählt

Ein Hexadezimalliteral beginnt in Java mit 0x; ein angehängtes L kennzeichnet long. Das vollständige int-Bitmuster 0xffffffff steht für -1, während 0xffffffffL den positiven long-Wert 4294967295 bezeichnet. Dieselben geschriebenen Hexadezimalziffern können somit aufgrund unterschiedlicher Typbreite verschiedene Zahlen repräsentieren.

A02.08 Normalisierte Fließkommazahlen

Eine normalisierte binäre Fließkommazahl wird als Vorzeichen mal Signifikand mal Zweierpotenz aufgefasst. Die führende Eins des Signifikanden ist dabei implizit; im Exponentenfeld steht der tatsächliche Exponent plus Bias. Diese Beschreibung gilt nicht unverändert für 0, subnormale Zahlen, Unendlichkeiten und NaN, die besondere Kodierungen besitzen.

A02.09 Genauigkeit von float und double

float besitzt ein Vorzeichenbit, acht Exponentenbits und 23 gespeicherte Bruchbits; bei double sind es eins, elf und 52. Die Biaswerte lauten 127 beziehungsweise 1023. Endliche Binärbrüche können bei ausreichender Genauigkeit exakt gespeichert werden, während beispielsweise ein Zehntel binär periodisch ist und gerundet werden muss.

A02.10 Überlauf ist keine zusätzliche Stelle · Ergänzung

Bei int-Arithmetik bleibt das Ergebnis auf 32 Bits begrenzt. Wenn die mathematische Rechnung mehr Stellen erfordert, werden die höherwertigen Bits verworfen, wodurch sich auch das interpretierte Vorzeichen ändern kann. Anders als bei einer beliebig großen mathematischen Ganzzahl wächst die Darstellung nicht automatisch mit.

Beispiel: Ein exakt darstellbarer Bruch

5,75 lässt sich binär als 101,11 und normalisiert als $1,0111 \text{ mal } 2^2$ schreiben. Das Vorzeichen ist positiv, der tatsächliche Exponent zwei und der gespeicherte float-Exponent deshalb 129. Hinter der impliziten führenden Eins werden die Bruchbits 0111 und anschließend 0-Bits gespeichert. Hier genügt eine endliche Binärdarstellung; bei 0,1 lässt sich die entsprechende Bruchfolge dagegen nicht durch endlich viele gespeicherte Bits vollständig erfassen.

Typischer Fehler: Die implizite führende Eins darf bei 0 und den übrigen Sonderfällen nicht einfach vorausgesetzt werden.

Verständnisfrage: Warum hilft das Anhängen von L bei einem Hexadezimalliteral?

Antwort: Es legt den Literaltyp long und damit eine größere Bitbreite fest. Dadurch kann ein Bitmuster positiv darstellbar sein, das als vollständiges int-Muster negativ interpretiert wird.

A03 Datentypen, Operatoren und sichere Auswertung

Typen wählen, umwandeln und vergleichen

Du prüfst, welche Werte eine Operation tatsächlich erhält und ob ihre Berechnung schon vor einer Zuweisung Informationen verliert.

A03.01 Primitive Werte unterscheiden

int und long speichern ganze Zahlen, float und double Fließkommazahlen. boolean enthält Wahrheitswerte; char bezeichnet eine einzelne UTF-16-Codeeinheit und wird in den Übungen für einfache Zeichen verwendet. Der deklarierte Typ bestimmt erlaubte Operationen und mögliche Werte, unabhängig davon, ob die Variable lokal, als Parameter oder als Feld vorkommt.

A03.02 Konvertierungen nicht verwechseln

Eine erweiternde numerische Konvertierung ist häufig ohne Cast erlaubt, während eine verengende Konvertierung ausdrücklich verlangt werden kann. Erweiterung bedeutet dabei nicht immer vollständige Genauigkeit: Ein großer long-Wert kann bei der Umwandlung nach float Information verlieren. boolean ist kein Zahlentyp und kann nicht einfach in int oder long umgewandelt werden.

A03.03 Vor der Rechnung umwandeln

Sind beide Operanden einer Division ganzzahlig, wird auch ganzzahlig gerechnet und der Bruchteil abgeschnitten. Eine spätere Zuweisung des Ergebnisses an double stellt diesen verlorenen Anteil nicht wieder her. Ebenso muss für eine größere Produktbreite bereits vor der Multiplikation wenigstens ein geeigneter Operand nach long umgewandelt werden.

A03.04 Zuweisung und Vergleich

Das Gleichheitszeichen = weist einer Variablen einen Wert zu, während == einen Gleichheitsvergleich berechnet. != prüft Ungleichheit; <, <=, > und >= vergleichen Größen. Eine Zeichenfolge wie ?= ist kein Java-Operator, und bei gemischten numerischen Typen erfolgt vor dem Vergleich die vorgeschriebene Typanpassung.

Beispiel: Zeitpunkt einer Umwandlung

Im folgenden Fragment wird a zuerst aus zwei int-Werten berechnet. Deshalb entsteht zunächst die ganze Zahl zwei, die erst anschließend als 2.0 gespeichert wird. Bei b ist durch den Cast schon die Division eine double-Rechnung und liefert 2.5. Beim Produkt sorgt der vorangestellte long-Cast dafür, dass nicht zuerst ein zu kleines int-Zwischenergebnis entsteht. Die Zielvariable allein bestimmt die Art der vorausgehenden Rechnung also nicht.

```
double a = 5 / 2;  
double b = (double) 5 / 2;  
long flaeche = (long) 50000 * 50000;
```

Typischer Fehler: Ein Cast nach einer bereits abgeschlossenen Ganzzahldivision oder übergelaufenen Multiplikation kann die ursprünglichen Werte nicht rekonstruieren.

Verständnisfrage: Ist long nach float grundsätzlich verboten, weil dabei Genauigkeit verloren gehen kann?

Antwort: Nein. Diese numerische Erweiterung ist in Java zulässig, kann aber runden. Die Zulässigkeit einer Konvertierung ist von ihrer exakten Werterhaltung zu unterscheiden.

A03 Datentypen, Operatoren und sichere Auswertung

Ausdrücke mit Seiteneffekten Schritt für Schritt

Du wertest Verkettung, Inkremente und boolesche Bedingungen in der richtigen Reihenfolge aus, ohne Zwischenänderungen zu übersehen.

A03.05 Plus kann zwei Aufgaben haben

Bei zwei Zahlen führt + eine Addition aus; sobald die betreffende Operation einen String enthält, entsteht eine Verkettung. Da die Operatoren von links nach rechts zusammengefasst werden, kann ein früher String alle folgenden Plusoperationen beeinflussen. Klammern legen ausdrücklich fest, welche Teilrechnung vorher abgeschlossen werden muss.

A03.06 Alter oder neuer Wert?

Postinkrement liefert zunächst den alten Wert eines Operanden und erhöht danach dessen gespeicherten Wert. Präinkrement erhöht zuerst und liefert den neuen Wert; für Dekrement gilt die entsprechende Unterscheidung beim Verringern. Auch wenn eine Bedingung anschließend false ergibt, bleiben bereits ausgeführte Änderungen an ihren Operanden bestehen.

A03.07 Zusammengesetzte Zuweisungen

Bei einer Anweisung wie `erg += berechne()` wird der bisherige linke Wert für die Addition berücksichtigt und die rechte Seite vollständig ausgewertet. Während `berechne` noch läuft, ist die abschließende Zuweisung daher noch nicht erfolgt. Die Kurzschreibweise enthält außerdem eine implizite Rückkonvertierung auf den linken Typ und ist deshalb nicht in jedem Typdetail identisch mit einer ausgeschriebenen Zuweisung.

A03.08 Boolesche Verknüpfungen

`&&` überspringt die rechte Seite, wenn links bereits false steht; `||` überspringt sie bei true. Die booleschen Varianten von `&` und `|` werten normalerweise beide Seiten aus, während `^` genau dann true liefert, wenn sich die Wahrheitswerte unterscheiden. Solche Unterschiede beeinflussen nicht nur Ergebnisse, sondern auch mögliche Seiteneffekte.

Beispiel: Ein Ausdruck, mehrere Änderungen

Zuerst liefert `x++` den alten Wert drei und setzt `x` auf vier. Danach erhöht `++x` auf fünf und liefert fünf, sodass `y` schließlich acht wird. Die letzte Zeile gibt Wert: 45 aus, weil nach dem String die beiden Zahlen als Text angehängt werden. Würde man `4 + 5` einklammern, würde stattdessen zunächst neun berechnet. Schreibe bei schwierigen Ausdrücken jeden Zwischenwert auf, bevor du zur nächsten Operation gehst.

```
int x = 3;
int y = x++ + ++x;
System.out.println("Wert: " + 4 + 5);
```

Typischer Fehler: Ein späteres false macht zuvor ausgeführte Inkremente nicht rückgängig. Außerdem ist eine Textausgabe kein Beweis für numerische Addition.

Verständnisfrage: Warum kann `n != 0 && 10 / n > 1` bei einem `int n` eine Division durch 0 vermeiden?

Antwort: Ist `n` gleich 0, ergibt bereits der linke Vergleich false. Die Kurzschlussauswertung überspringt dann die Division auf der rechten Seite vollständig.

A03 Datentypen, Operatoren und sichere Auswertung

Hilfsmethoden und Grenzen maschineller Arithmetik

Du formulierst kleine Entscheidungen verständlich und sicherst Rechnungen gegen Fehler ab, die erst an großen Eingaben sichtbar werden.

A03.09 Hilfsmethode oder bedingter Ausdruck

Eine statische Hilfsmethode kann eine wiederholt benötigte Berechnung ohne eigenes Empfängerobjekt bereitstellen. Für zwei einfache Ergebnisalternativen eignet sich der Ausdruck `bedingung ? wert1 : wert2`. Nur der ausgewählte Ergebnisweig wird ausgewertet; bei größeren Verarbeitungsblöcken bleibt eine gewöhnliche Fallunterscheidung meist leichter nachvollziehbar.

A03.10 Zwischenergebnisse ausreichend breit · Ergänzung

Nicht nur das Endergebnis, sondern jedes Zwischenergebnis muss in den verwendeten Datentyp passen. Ein `long`-Akkumulator hilft nur, wenn ein bereits vorher gebildetes `int`-Produkt nicht schon übergelaufen ist. Auch Arraydimensionen und Kostenberechnungen benötigen deshalb eine bewusste Prüfung ihrer Grenzen statt bloßer Annahmen aufgrund kleiner Beispiele.

A03.11 Schleifengrenzen ohne Überlauf · Ergänzung

Für positive Werte ist `t <= rest/t` eine geeignete Alternative zur Prüfung `t*t <= rest`, weil kein möglicherweise überlaufendes Quadrat gebildet wird. Eine Schleife bis `Integer.MAX_VALUE` benötigt außerdem einen Zähler, dessen nächste Erhöhung noch darstellbar ist. Ein `long`-Zähler vermeidet hier das Zurückspringen in den negativen `int`-Bereich.

A03.12 Häufigkeitsarray und größter Index · Ergänzung

Soll ein Häufigkeitsarray jeden Wert bis maximum als Index aufnehmen, benötigt es grundsätzlich `maximum + 1` Plätze. Ist `maximum` bereits `Integer.MAX_VALUE`, ist diese positive Länge nicht als `int` darstellbar. Selbst kleinere rechnerisch gültige Längen können den verfügbaren Speicher überschreiten; Typbereich und Speicherbedarf sind getrennte Grenzen.

A03.13 Ausnahmen unterbrechen die Auswertung · Ergänzung

Die Aussage, dass `&` oder `|` beide booleschen Operanden auswertet, beschreibt einen normalen Ablauf ohne vorzeitigen Fehler. Löst bereits der linke Operand eine Ausnahme aus, wird die Auswertung abgebrochen und der rechte nicht mehr erreicht. Dies ist kein Kurzschlussverhalten, sondern eine außergewöhnliche Beendigung des Ausdrucks.

Beispiel: Ein größerer Produktbereich

Für eine rechteckige Fläche mit `int`-Abmessungen wählt man vor der Multiplikation `long`. Die Anweisung `long flaeche = (long) breite * hoehe` rechnet bereits das Produkt mit der größeren Breite. Die Variante `long flaeche = breite * hoehe` führt dagegen zunächst eine `int`-Multiplikation aus. Dieser Unterschied ist bei kleinen Seitenlängen unsichtbar, wird bei genügend großen Abmessungen aber entscheidend.

Typischer Fehler: Auch `long` ist endlich. Ein größerer Typ verschiebt die Grenze, er hebt die Pflicht zur Prüfung passender Wertebereiche nicht auf.

Verständnisfrage: Warum kann `t*t <= n` bei positiven `int`-Werten falsch entscheiden?

Antwort: Das Produkt `t*t` kann überlaufen und einen unzutreffenden, möglicherweise negativen Wert erhalten. Für positive `t` vermeidet `t <= n/t` dieses konkrete Überlaufproblem.

A04 Kontrollstrukturen, Konsoleneingabe und Ablaufspuren

Eingaben verarbeiten und Wiederholungen steuern

Du verbindest Eingaben mit Kontrollstrukturen und trennst einmalige Initialisierung von wiederholter Verarbeitung.

A04.01 Ein- und Ausgabe verbinden

Ein Scanner wird mit passendem Import und System.in eingerichtet. `nextInt` liest Ganzzahlen; Scanner-Methoden wie `next` und `nextLine` lesen Text. `print` schreibt ohne zusätzlichen Zeilenwechsel, `println` mit Zeilenwechsel; `main` bildet den Einstieg in das ausführbare Programm.

A04.02 Die passende Wiederholung wählen

`for` eignet sich für zählbare Durchläufe, `while` prüft seine Bedingung vor dem Rumpf und `do-while` danach. `if/else` und Zustandsflags steuern Entscheidungen. `break` beendet die nächste umgebende Schleife oder den `switch`, `return` den Methodenaufruf.

A04.03 Initialisierung richtig platzieren

Ein Akkumulator wird vor jener Schleife initialisiert, deren Ergebnisse er sammeln soll. Erneutes Zurücksetzen im Rumpf würde bisherige Ergebnisse löschen. Eigene Hilfsmethoden bündeln wiederholte Teilberechnungen, ohne den gesamten Algorithmus einer Bibliothek zu überlassen.

A04.04 Menü mit switch

Nach einmaliger Dateneingabe nimmt das Menü wiederholt eine Auswahl entgegen. `switch` wählt den passenden `case`; `break` verhindert das Weiterlaufen, `default` behandelt unbekannte Auswahlen. Die Abbruchauswahl wird in der Schleifenbedingung geprüft.

A04.05 Einen gesamten Versuch wiederholen

Pro Spiel erhält ein Team 3 Punkte für einen Sieg, 1 für ein Remis und 0 für eine Niederlage; die Punkte werden aufsummiert. Sind nach allen drei Spielen zwei Teamstände gleich, werden sämtliche Spiele neu eingelesen. Vor diesem Gesamtversuch werden alle Punktzähler zurückgesetzt.

A04.06 Abschlusswert ausschließen

Ein Sentinel beendet eine Eingabefolge, gehört aber nicht zu ihren Nutzdaten. Deshalb wird jeder gelesene Wert vor Addition, Zählung oder Aufnahme geprüft. Eine spätere Prüfung könnte den Abschlusswert bereits fälschlich in das Ergebnis aufgenommen haben.

Beispiel: Summe bis zur Abschlussmarke

Bei den Eingaben 4, 6 und -1 werden nur 4 und 6 addiert. Nach jeder Verarbeitung wird neu gelesen und vor der nächsten Addition geprüft. Die Summe wird einmal vor der Schleife initialisiert; ein Zurücksetzen im Rumpf würde die erste Zahl vergessen.

Typischer Fehler: Das Zurücksetzen vor einem neuen Gesamtversuch ist etwas anderes als das Zurücksetzen vor jeder einzelnen Eingabe.

Verständnisfrage: Weshalb eignet sich `do-while` für eine Menüauswahl?

Antwort: Die Auswahl muss mindestens einmal gelesen werden, bevor die Wiederholungsbedingung ihren Wert prüfen kann.

A04 Kontrollstrukturen, Konsoleneingabe und Ablaufspuren

Mittelwerte, Schleifenspuren und Eingabefallen

Du verfolgst die letzte Schleifenprüfung genauso sorgfältig wie den Rumpf und behandelst leere Folgen sowie Scanner-Eingaben ausdrücklich.

A04.07 Summe und Anzahl gemeinsam führen

Für einen Mittelwert werden sowohl die Summe als auch die Anzahl der angenommenen Werte benötigt. Ein Cast vor der Division ermöglicht ein nichtganzzahliges Ergebnis. Beendet beispielsweise die erste nichtnegative Zahl eine Folge negativer Zahlen, gehört dieser Abschlusswert nicht zu Summe oder Anzahl.

A04.08 Endzustand statt letztem Rump fzustand

Bei `for` wird nach dem letzten ausgeführten Rumpf gewöhnlich noch der Aktualisierungsschritt durchgeführt. Erst der folgende Bedingungstest beendet die Schleife. Bei ganzzahliger Halbierung werden Bruchteile abgeschnitten; die Folge der gespeicherten Werte kann deshalb anders aussehen als eine Rechnung mit Dezimalzahlen.

A04.09 Tests haben ebenfalls Seiteneffekte

Eine Spurentabelle unterscheidet den Zustand nach dem Rumpf vom Zustand nach der Bedingung. Enthält der Test `x--` oder `y++`, ändern sich diese Variablen auch dann, wenn der Vergleich `false` liefert. Nur die weitere Rumpfausführung entfällt; die bereits ausgeführten Änderungen bleiben bestehen.

A04.10 Wörter und Zeilen lesen · Ergänzung

`next` liest ein durch Leerraum getrenntes Wort, `nextLine` dagegen den Rest einer vollständigen Zeile. Nach `nextInt` kann zunächst noch ein Zeilenende übrig sein, sodass ein direkt folgendes `nextLine` einen leeren Rest liest. Die gewünschte Eingabeform muss deshalb vor der Kombination der Methoden feststehen.

A04.11 Ohne Werte kein Durchschnitt · Ergänzung

Bei einer leeren Folge ist die Summe 0, eine Division durch ihre Anzahl aber kein definierter Mittelwert. Prüfe deshalb zuerst, ob überhaupt Werte angenommen wurden. Eine Meldung oder eine ausdrücklich vereinbarte Ersatzrückgabe ist sinnvoller als ein stillschweigend erfundenes Messergebnis.

A04.12 Standardeingabe gemeinsam nutzen · Ergänzung

Das Schließen eines Scanners schließt auch seine zugrunde liegende Eingabequelle. Bei `System.in` kann dies spätere Eingaben anderer Methoden verhindern. In kleinen Aufgaben mit gemeinsam benutzter Konsole sollte die Zuständigkeit deshalb klar sein, statt jeden Scanner unmittelbar in einer Hilfsmethode zu schließen.

Beispiel: Ein letzter falscher Test verändert Werte

Starten `x` und `y` mit 5 und 4, verringert der Rumpf `x` zunächst auf 4. Der Test `x-- > y++` vergleicht dann 4 mit 4 und ist `false`. Trotzdem endet die Schleife mit `x = 3` und `y = 5`. Die Werte 4 und 4 waren Vergleichswerte, nicht der endgültige gespeicherte Zustand.

Typischer Fehler: Der letzte Test gehört zur Ausführung. Eine Spur, die unmittelbar nach dem letzten Rumpf endet, kann falsche Endwerte liefern.

Verständnisfrage: Was sollte vor der Berechnung `summe / anzahl` geprüft werden?

Antwort: Ob `anzahl` größer als 0 ist. Erst dann besitzt der Mittelwert über die angenommenen Werte eine sinnvolle Definition.

A05 Zahlenalgorithmen, Häufigkeiten und Auswahl von Extremwerten

Teiler, Primfaktoren und Dezimalziffern

Du zerlegst Zahlen durch Division und Restbildung und entscheidest, welche Informationen du für das gewünschte Ergebnis aufbewahren musst.

A05.01 Teiler erkennen

Eine positive Zahl t teilt n genau dann ganzzahlig, wenn $n \% t = 0$ ergibt. Für die Teileranzahl werden passende Kandidaten gezählt; eins und n selbst gehören dazu. Eine Primzahl besitzt genau zwei positive Teiler, weshalb insbesondere die Zahl eins keine Primzahl ist.

A05.02 Primfaktoren schrittweise entfernen

Bei der Primfaktorzerlegung werden Kandidaten aufsteigend getestet und gefundene Faktoren vollständig herausdividiert. Sobald kein Kandidat bis zur Quadratwurzel des aktuellen Rests mehr übrig ist, ist ein Rest größer als 1 selbst prim. Er wird als letzter Faktor angehängt; die Ausgabe benötigt klare Trennzeichen.

A05.03 Ziffern gewinnen

Der Rest bei Division durch zehn ist die letzte Dezimalziffer, die ganzzahlige Division entfernt sie. Zuerst lässt sich so die benötigte Arraylänge bestimmen. Soll das Ergebnis die ursprüngliche Dezimalreihenfolge besitzen, werden die anschließend gewonnenen Ziffern von rechts nach links in dieses Array eingetragen.

A05.04 Quersumme berechnen

Die Quersumme addiert sämtliche Dezimalziffern, unabhängig von ihrer Position. Man kann die Ziffern unmittelbar beim Abtrennen addieren oder zunächst ein Ziffernarray erzeugen und dessen Elemente aufsummieren. Die zweite Variante zeigt, wie zwei bereits korrekte Hilfsmethoden zu einer neuen Berechnung zusammengesetzt werden.

A05.05 Häufigste Ziffer auswählen

Für Werte von 0 bis neun genügt ein Array mit zehn Häufigkeitszählern. Jeder gelesene Wert erhöht seinen eigenen Zähler; anschließend werden die Zähler in aufsteigender Wertreihenfolge geprüft. Ersetzt man den bisherigen Gewinner nur bei strikt größerer Häufigkeit, bleibt bei Gleichstand automatisch die kleinere Zahl erhalten.

Beispiel: Zwei unterschiedliche Zerlegungen derselben Zahl

Für 84 liefert die Primfaktorzerlegung die Faktoren 2, 2, 3 und 7. Die Dezimalzerlegung derselben Zahl liefert dagegen nur die Ziffern 8 und 4. Ihre Quersumme ist 12 und darf nicht mit der Summe der Primfaktoren verwechselt werden. Bei 840 entstehen zunächst die abgetrennten Ziffern 0, 4 und 8; für eine Darstellung als Dezimalziffernarray werden sie in umgekehrter Reihenfolge abgelegt.

Typischer Fehler: Ein Primfaktor und eine Dezimalziffer sind verschiedene Begriffe. Die benutzte Division bestimmt, welche Art von Zerlegung entsteht.

Verständnisfrage: Warum muss bei gleicher Häufigkeit der Vergleich strikt größer und nicht größer oder gleich lauten?

Antwort: Bei aufsteigender Suche würde größer oder gleich den bisherigen kleineren Gewinner durch einen späteren größeren Wert ersetzen. Der strikte Vergleich erhält die verlangte Gleichstandsregel.

A05 Zahlenalgorithmen, Häufigkeiten und Auswahl von Extremwerten

Maximum, Häufigkeiten und begrenzte Auswahl

Du lernst, eine bisher beste Lösung beim Durchlaufen von Zahlen gezielt zu aktualisieren und dabei Filterbedingungen von Ersetzungsregeln zu trennen.

A05.06 Maximum und Verteilung

Ein globales Maximum über ein zweidimensionales Feld entsteht durch das Vergleichen aller tatsächlich vorhandenen Elemente. Der Startwert 0 ist nur dann allgemein geeignet, wenn negative Werte ausgeschlossen sind. Anschließend kann ein Array der Länge maximum plus eins jedem Zahlenwert seinen Häufigkeitszähler zuordnen; dafür wird das Feld erneut durchlaufen.

A05.07 Dreierexponenten zählen

Solange eine von 0 verschiedene Zahl durch drei teilbar ist, wird sie durch drei geteilt und ein Zähler erhöht. Jeder Durchlauf entfernt genau einen Dreierfaktor. Der abschließende Zähler ist deshalb der größte Exponent t , für den die ursprüngliche Zahl noch durch 3^t teilbar war.

A05.08 Nur passende Ergebnisse vergleichen

Bei der Suche nach dem größten geraden Dreierexponenten wird zunächst für jede Zahl der Exponent bestimmt. Erst wenn dieser Exponent gerade ist, darf er das bisherige Maximum ersetzen. Der Filter bezieht sich auf den berechneten Exponenten und nicht auf die Parität der ursprünglichen Zahl.

A05.09 Einen maximalen Platz ersetzen

Zuerst wird der Index eines maximalen Arrayelements gesucht. Nur wenn der neu angebotene Wert echt kleiner ist, wird genau dieser Platz überschrieben. Mehrere gleiche Maxima erfordern somit nicht mehrere Änderungen; die Aufgabe wählt eine Position, während alle übrigen Elemente ihren bisherigen Wert behalten.

A05.10 Die kleinsten Werte festhalten

Ein begrenztes Auswahlarray speichert die bisher kleinsten k Eingaben. Solange Plätze frei sind, werden neue Werte aufgenommen; danach kann nur ein Wert unterhalb des aktuellen Auswahlmaximums noch relevant sein. Doppelte Zahlen zählen als getrennte Eingaben und dürfen deshalb mehrere Plätze belegen, wenn sie zu den kleinsten gehören.

Beispiel: Eine Auswahl verändert sich

Für $k = 2$ treffen nacheinander -3, -8, -5 und -8 ein. Nach den ersten beiden Werten ist die Auswahl voll. -5 ersetzt das größere Auswahllement -3; die zweite -8 ersetzt danach -5. Die Auswahl enthält schließlich zweimal -8, weil beide Vorkommen eigenständige Eingaben sind. Eine Menge unterschiedlicher Zahlen wäre hier das falsche Modell. Die physische Reihenfolge im Auswahlarray spielt für die Auswahl selbst keine Rolle.

Typischer Fehler: Bei negativen Zahlen ist -8 kleiner als -3. Betragsgröße und gewöhnliche Zahlenordnung dürfen bei der Auswahl nicht verwechselt werden.

Verständnisfrage: Warum ist ein Maximum mit Startwert 0 für ein ausschließlich negatives Feld problematisch?

Antwort: Kein negativer Wert überschreitet 0, sodass ein nicht vorkommender Wert als Maximum stehen bliebe. Ohne Nichtnegativitätsannahme muss die Initialisierung anders erfolgen.

A05 Zahlenalgorithmen, Häufigkeiten und Auswahl von Extremwerten

Leere Fälle und belastbare Invarianten

Du erkennst, wann eine mathematische Definition einen neutralen Wert liefert und wann eine zusätzliche Aufgabenkonvention erforderlich ist.

A05.11 Weniger Werte als Plätze

Werden weniger als k Zahlen eingelesen, dürfen nur tatsächlich vorhandene Auswahlwerte summiert oder multipliziert werden. Die leere Summe hat den Wert 0, das leere Produkt den Wert eins. Diese neutralen Werte betreffen die Rechnung und bedeuten nicht, dass entsprechende Eingaben vorhanden gewesen wären.

A05.12 Primfaktoren der Eins · Ergänzung

Die Zahl eins besitzt keine Primfaktoren, weil sie nicht prim ist und keinen nichttrivialen positiven Faktor enthält. Mathematisch entspricht ihr das leere Produkt. Ob die zugehörige Textausgabe leer bleibt oder beispielsweise die Zeichenfolge 1 enthält, muss die Aufgaben- oder Lösungskonvention festlegen.

A05.13 0 als Dezimalziffer · Ergänzung

Die Dezimaldarstellung der Zahl 0 besteht aus einer Ziffer, auch wenn eine Schleife mit der Bedingung $n > 0$ keinen Durchlauf hätte. Innere 0-Ziffern anderer Zahlen werden beim Teilen ebenfalls sichtbar. Für Häufigkeiten ist ihre Reihenfolge unerheblich, ihre Anzahl aber selbstverständlich nicht.

A05.14 Leere Häufigkeitsfelder · Ergänzung

In einem leeren Eingabefeld kommt kein Wert tatsächlich vor. Ein Algorithmus kann dennoch unter lauter leeren Zählern konventionell den kleinsten zulässigen Wert auswählen. Diese Rückgabe darf nicht als vorkommendes häufigstes Element bezeichnet werden; dafür wäre eine zusätzliche Nichtleerheitsvoraussetzung nötig.

A05.15 Grenzen der Dreierbewertung · Ergänzung

Negative Zahlen können ebenfalls wiederholt durch drei geteilt werden; das Vorzeichen verändert die Anzahl ihrer Dreierfaktoren nicht. Für 0 fehlt dagegen ein größter Exponent. Gerade Exponenten entsprechen Neunerpotenzen, wodurch eine alternative Maximumsuche wiederholte Division durch neun statt Prüfung sämtlicher Zwischenzahlen nutzen kann.

A05.16 Eine Eigenschaft bleibt erhalten · Ergänzung

Ein Belegungszähler trennt echte Auswahlinträge von unbenutzten Arrayplätzen. Nach jedem Durchlauf soll die Auswahl weiterhin die bisher kleinsten aufgenommenen Werte enthalten; dies ist ihre Invariante. Für null oder leere Arrays benötigt eine ergänzte Schutzbehandlung einen ausdrücklich definierten Umgang, bevor ein Maximumindex verwendet wird.

Beispiel: Freie Plätze nicht mitmultiplizieren

Ein Auswahlarray hat vier Plätze, enthält aber erst -2 und -3. Der Belegungszähler steht deshalb auf zwei. Das Produkt wird mit eins begonnen und nur über diese zwei Einträge gebildet. Würden auch die automatisch mit 0 initialisierten freien Plätze verwendet, würde das Produkt fälschlich verschwinden.

Typischer Fehler: Ein Standardwert eines freien Speicherplatzes ist keine beobachtete Eingabe und darf nicht unbemerkt in die fachliche Berechnung eingehen.

Verständnisfrage: Ist ein leeres Feld dasselbe wie null?

Antwort: Nein. Ein leeres Array ist ein vorhandenes Objekt mit Länge 0; null bezeichnet dagegen das Fehlen einer Arrayreferenz.

A06 Stringalgorithmen: Filtern, Wiederholen, Suchen und Löschen

Strings zeichenweise aufbauen und filtern

Du setzt einfache Stringaufgaben aus Indexzugriffen zusammen und erhältst dabei die gewünschte Reihenfolge der Zeichen oder Wörter.

A06.01 Über Zeichen laufen

`length()` liefert die Anzahl der UTF-16-Codeeinheiten, `charAt(i)` liest diejenige am gültigen Index `i`. In den Übungen mit einfachen Buchstaben kann damit schrittweise ein Ergebnisstring aufgebaut werden. Wenn nur bestimmte Bibliotheksmethoden erlaubt sind, wird die Verarbeitung aus diesen kleinen Zugriffen selbst zusammengesetzt statt durch eine bequemere Gesamtoperation ersetzt.

A06.02 Wörter nach Parität ausgeben

Für eine stabile Gruppierung werden zunächst alle Wörter gespeichert. Ein erster Durchlauf gibt Wörter gerader Länge aus, ein zweiter diejenigen ungerader Länge; innerhalb jedes Durchlaufs bleibt die ursprüngliche Reihenfolge erhalten. Es ist keine alphabetische Sortierung nötig, denn die verlangte Ordnung betrifft ausschließlich zwei Längengruppen.

A06.03 Ein oder alle Treffer löschen

Sollen alle Treffer eines Zeichens entfernt werden, werden nur andere Zeichen an das Ergebnis angehängt. Für genau den ersten Treffer merkt sich dagegen ein `boolean`, ob bereits gelöscht wurde. Spätere gleiche Zeichen werden dann wieder übernommen; bei keinem Treffer bleiben sämtliche Zeichen und ihre Reihenfolge erhalten.

A06.04 Wiederholung und Präfix

Ein Wort wird `n`-mal wiederholt, indem derselbe Text in `n` Durchläufen angehängt wird. Ein Präfix der Länge `i` besteht hingegen aus genau den Zeichen an den Indizes 0 bis `i` minus eins. Die erlaubte Präfixlänge reicht bei dieser Definition vom leeren Präfix bis zur vollständigen Wortlänge.

Beispiel: Genau ein Zeichen überspringen

Für `wort = "Banane"` und `gesucht = 'a'` überspringt das Fragment nur das erste `a`. Das Ergebnis lautet `Bnane`; das spätere `a` bleibt erhalten. Das Flag wird erst beim tatsächlichen Löschen gesetzt. Würde die Bedingung allein das Zeichen vergleichen, verschwänden beide `a`. Das Beispiel zeigt, wie ein kleiner zusätzlicher Zustand zwischen zwei äußerlich ähnlichen Filteraufgaben unterscheidet, ohne die Reihenfolge aller übrigen Zeichen zu verändern.

```
String ergebnis = "";
boolean geloescht = false;
for (int i = 0; i < wort.length(); i++) {
    char c = wort.charAt(i);
    if (!geloescht && c == gesucht) geloescht = true;
    else ergebnis += c;
}
```

Typischer Fehler: Der letzte gültige Stringindex ist `length() - 1`. Ein Zugriff mit dem Index `length()` liegt bereits hinter dem Wort.

Verständnisfrage: Warum bleibt die Gruppierung nach gerader und ungerader Länge stabil?

Antwort: Jede Gruppe wird in einem vollständigen Durchlauf von links nach rechts ausgegeben. Damit bleibt die relative Reihenfolge ihrer Wörter aus der ursprünglichen Eingabe erhalten.

A06 Stringalgorithmen: Filtern, Wiederholen, Suchen und Löschen

Wiederholte Wörter und Teilwortsuche

Du nutzt Längenbedingungen zur Eingrenzung von Kandidaten und überprüfst danach die tatsächlichen Zeichen statt nur passende Zahlenverhältnisse.

A06.05 Gleiche Hälften prüfen

Ein nichtleeres Wort der Form vv hat gerade Länge und zwei gleich lange, übereinstimmende Hälften. Für mehr als zwei Wiederholungen kommen unterschiedliche Grundwortlängen infrage. Eine passende Gesamtlänge allein beweist noch keine Wiederholung; die Zeichen an den entsprechenden Positionen müssen ebenfalls übereinstimmen.

A06.06 Kandidaten systematisch bilden

Die Länge eines nichtleeren Grundworts muss die Gesamtlänge teilen. Man nimmt das Präfix dieser Kandidatenlänge und wiederholt es entsprechend oft; anschließend wird der Inhalt mit dem vollständigen Wort verglichen. So werden unmögliche Längen früh ausgeschlossen und verbleibende Kandidaten mit einer eindeutigen Prüfung bewertet.

A06.07 An einem Index suchen

Vor dem Zeichenvergleich muss das Suchwort ab dem vorgesehenen Index vollständig in das Ausgangswort passen. Danach werden entsprechende Zeichenpaare verglichen. Für das erste Vorkommen werden mögliche Startpositionen aufsteigend geprüft; die erste erfolgreiche Position ist der kleinste Trefferindex, ansonsten gilt der vereinbarte Fehlerwert.

A06.08 Ein Intervall auslassen

Ein gefundenes Teilwort belegt ein zusammenhängendes Indexintervall. Zum Löschen werden die Zeichen davor und danach übernommen, während genau dieses Intervall übersprungen wird. Wenn kein Vorkommen existiert, muss die Methode ihre vereinbarte Rückgabe verwenden; eine null-Rückgabe ist eine Schnittstellenentscheidung und kein leerer String.

A06.09 Nach dem Löschen neu suchen

Das Entfernen eines Teilworts kann zuvor getrennte Zeichen zusammenbringen und dadurch einen neuen Treffer erzeugen. Deshalb genügt ein einmaliger Durchlauf über die ursprünglichen Positionen nicht immer. Bei mehreren Löschwörtern ist das Ende erst erreicht, wenn ein vollständiger Durchlauf keine weitere Löschung mehr ermöglicht.

Beispiel: Eine Löschung schafft einen neuen Treffer

Das Wort `abXcd` enthält zunächst das Löschwort `X`. Nach dessen Entfernung entsteht `abcd`; darin kann jetzt das weitere Löschwort `bc` entfernt werden, sodass `ad` übrig bleibt. Wer `bc` nur vor dem Entfernen von `X` geprüft und danach nie wieder gesucht hätte, würde diesen Treffer übersehen. Das Beispiel begründet das Wiederholen der Suche und zeigt zugleich, warum gespeicherte alte Indexpositionen nach einer Löschung nicht ungeprüft weiterverwendet werden dürfen.

Typischer Fehler: Eine Länge als Teiler der Wortlänge ist nur eine notwendige Bedingung für Wiederholung, noch kein Beweis für gleichen Inhalt.

Verständnisfrage: Welche Kandidatenbedingung muss vor `charAt` beim Teilwortvergleich geprüft werden?

Antwort: Der Startindex muss zulässig sein und das gesamte Suchwort muss bis zu seinem Ende innerhalb des Ausgangswortes liegen; andernfalls droht ein Zugriff außerhalb des Strings.

A06 Stringalgorithmen: Filtern, Wiederholen, Suchen und Löschen

Leerwort, Referenzen und Ende einer Löschung

Du unterscheidest inhaltliche Stringfragen von Referenzfragen und begründest, unter welchen Voraussetzungen wiederholte Verarbeitung tatsächlich endet.

A06.10 Reihenfolge kann das Ergebnis ändern

Wenn sich löschbare Teilwörter überschneiden, kann eine erste Löschung die Möglichkeit einer anderen beseitigen. Unterschiedliche Reihenfolgen müssen deshalb nicht zum gleichen Endwort führen. Eine Aufgabe kann mehrere solche Ergebnisse zulassen; entscheidend ist dann, dass die gewählte Reihenfolge erlaubt ist und am Ende kein Löschwort mehr vorkommt.

A06.11 Inhalt ist nicht Identität · Ergänzung

`equals` vergleicht bei Strings ihren Inhalt, während `==` prüft, ob beide Referenzen dasselbe Objekt bezeichnen. Zwei getrennte Stringobjekte können daher inhaltlich gleich sein. Ist der mögliche Methodenempfänger null, darf `equals` darauf nicht aufgerufen werden; zuerst muss dieser Fall separat behandelt werden.

A06.12 Das leere Wort einordnen · Ergänzung

Das leere Wort besitzt Länge 0 und enthält kein Zeichen; es ist trotzdem ein vorhandener String. Ein Präfix der Länge 0 und eine 0-fache Wiederholung liefern dieses Wort. Formal kann das Leerwort als Verdopplung seiner selbst gelten, sofern die Aufgabe nicht ausdrücklich ein nichtleeres Grundwort voraussetzt.

A06.13 Parametergrenzen festlegen · Ergänzung

Negative Wiederholungszahlen und Präfixlängen außerhalb des erlaubten Bereichs passen nicht zur gewöhnlichen Definition. Eine Methode kann solche Aufrufe ausschließen oder mit einer dokumentierten Rückgabe behandeln. Ein leeres Suchwort passt an jede gültige Grenze einschließlich hinter das letzte Zeichen, benötigt bei Löschaufgaben jedoch eine gesonderte Betrachtung.

A06.14 Fortschritt beweist Terminierung · Ergänzung

Jede erfolgreiche Löschung eines nichtleeren Teilworts verkürzt den aktuellen String mindestens um ein Zeichen. Da die Länge eine nichtnegative ganze Zahl ist, können nicht unbegrenzt viele solche Löschungen erfolgen. Ein leeres Löschwort verändert die Länge dagegen nicht; die gleiche Fortschrittsbegründung funktioniert dann nicht.

Beispiel: Zwei gültige Endwörter

Aus `abc` sollen die Löschwörter `ab` und `bc` entfernt werden, bis keines mehr vorhanden ist. Löscht man zunächst `ab`, bleibt `c`. Beginnt man mit `bc`, bleibt `a`. Beide Endwörter enthalten keines der Löschwörter. Es liegt deshalb kein Widerspruch vor, wenn die Aufgabe keine feste Löschr Reihenfolge vorschreibt. Anders wäre es, wenn ausdrücklich stets das zuerst genannte Löschwort oder stets der früheste Treffer gewählt werden müsste.

Typischer Fehler: `null` bezeichnet keinen String. Der leere String ist dagegen ein gültiges Objekt mit wohldefinierter Länge und darf damit nicht verwechselt werden.

Verständnisfrage: Warum kann wiederholtes Löschen eines leeren Suchworts eine Endlosschleife verursachen?

Antwort: Das Suchwort wird weiterhin gefunden, aber seine Entfernung verändert den Text nicht. Ohne Sonderbehandlung wird damit dieselbe Situation immer wieder erreicht.

A07 Arrays: Grundlagen, Änderungen an Ort und Stelle und Messdaten

Arrayaufbau, Summen und Umkehrung

Du erkennst die Struktur eines Arrays und veränderst Elemente gezielt, ohne Länge, Index und gespeicherten Wert miteinander zu verwechseln.

A07.01 Länge und Inhalt trennen

Ein Array mit Länge n besitzt die gültigen Indizes 0 bis n minus eins. Seine Länge bleibt nach der Erzeugung fest, während die einzelnen Werte verändert werden können. Die Reihenfolge der Indizes bedeutet keine automatische Sortierung des Inhalts; auch das Inkrementieren eines Elements fügt keinen neuen Platz hinzu.

A07.02 Arrays können Arrays enthalten

Ein `int[][]` ist in Java ein Array, dessen Elemente Referenzen auf `int`-Arrays sind. Seine Zeilen dürfen deshalb unterschiedliche Längen haben oder sogar null sein, sofern die Aufgabe dies nicht ausschließt. Weitere Dimensionen entstehen nach demselben Prinzip; die häufig gezeichnete rechteckige Matrix ist lediglich ein besonderer Anwendungsfall.

A07.03 Alle Elemente genau einmal besuchen

Zur Summe wird ein Akkumulator mit 0 begonnen und jedes Element genau einmal addiert. Die Schleifengrenze orientiert sich an der tatsächlichen Arraylänge. Eine Methode kann den fertigen Wert zurückgeben, damit der Aufrufer ihn weiterverarbeitet; eine bloße Konsolenausgabe wäre für eine verlangte Rückgabemethode kein gleichwertiger Ersatz.

A07.04 Ohne neues Array umkehren

Für die Umkehrung werden das erste und letzte Element, dann das zweite und vorletzte vertauscht. Eine temporäre Variable bewahrt den ersten Wert auf, bevor er überschrieben wird. Nach der Hälfte der Paare ist das Feld umgekehrt; ein Weiterlaufen würde bereits bearbeitete Paare erneut zurücktauschen.

Beispiel: Ein Tausch verändert zwei Positionen

Das Array beginnt mit den Werten 2, 5, 8 und 11. Im ersten Schritt werden die äußeren Werte vertauscht, sodass 11, 5, 8, 2 entsteht. Der nächste Schritt vertauscht die beiden inneren Werte und liefert 11, 8, 5, 2. Die Arrayreferenz und die Länge vier bleiben dabei dieselben. Verändert werden ausschließlich die Inhalte der vorhandenen Positionen; es wird weder ein größeres Array angelegt noch eine zweite vollständige Wertfolge benötigt.

```
int[] a = {2, 5, 8, 11};
for (int links = 0; links < a.length / 2; links++) {
    int rechts = a.length - 1 - links;
    int temp = a[links];
    a[links] = a[rechts];
    a[rechts] = temp;
}
```

Typischer Fehler: Wer das erste Element direkt mit dem letzten überschreibt, ohne seinen alten Wert zu sichern, kann den zweiten Teil des Tauschs nicht mehr korrekt ausführen.

Verständnisfrage: Welche Länge muss die innere Schleife bei einem nichtrechteckigen `int[][]` verwenden?

Antwort: Die Länge der jeweils aktuellen, nicht-null-Zeile. Die Länge der ersten Zeile darf nicht ungeprüft für alle anderen Zeilen übernommen werden.

A07 Arrays: Grundlagen, Änderungen an Ort und Stelle und Messdaten

Präfixsummen und wachsende Tagesmessungen

Du unterscheidest Änderungen innerhalb eines bestehenden Arrays vom Austausch einer Zeilenreferenz und formulierst die nötigen Vorbedingungen.

A07.05 Präfixe von links nach rechts

Bei der Präfixsummenbildung soll jede Position die Summe aller ursprünglichen Werte bis dorthin enthalten. Deshalb wird ab dem zweiten Element der bereits aktualisierte Vorgänger addiert. Ein Durchlauf von rechts nach links hätte diese vorberechneten Summen noch nicht verfügbar und würde nur benachbarte Originalwerte zusammenfassen.

A07.06 Jeder Tag besitzt seine Zeile

Temperaturmessungen lassen sich nach Tagen gruppieren, wobei jede Tageszeile unterschiedlich viele Messungen enthalten kann. Eine zunächst leere Zeile besitzt Länge 0 und ist dennoch ein gültiges Array. Das äußere Array organisiert nur die Zuordnung der Tage; die inneren Arrays speichern ihre jeweiligen Messwerte.

A07.07 Eine Zeile gezielt verlängern

Da eine vorhandene Arraylänge unveränderlich ist, wird zum Anhängen ein neues Tagesarray mit einem zusätzlichen Platz erzeugt. Die bisherigen Messwerte werden kopiert und der neue Wert am Ende eingetragen. Erst danach ersetzt die neue Referenz die bisherige Tageszeile im äußeren Array; andere Tage bleiben unverändert.

A07.08 Form und Tagesindex prüfen

Bevor auf einen Tag zugegriffen wird, müssen der Index und die erwartete Anzahl von Tageszeilen passen. Der Mittelwert dieses Tages ergibt sich aus seiner eigenen Summe und seiner eigenen Messwertanzahl. Werte anderer Tage dürfen weder zum Zähler noch zum Nenner dieser Tagesberechnung beitragen.

A07.09 Vertrag der Methode festhalten

Vorbedingungen beschreiben, für welche Eingaben ein Algorithmus ausgelegt ist, etwa nicht-null, rechteckige Form oder positive Größen. `assert` kann solche Annahmen während einer entsprechend gestarteten Java-Ausführung prüfen. Assertions sind jedoch standardmäßig nicht zwingend aktiv und ersetzen keine immer erforderliche Eingabevalidierung einer allgemeinen Schnittstelle.

Beispiel: Ein neuer Wert verändert nur einen Tag

Angenommen, die erste Tageszeile enthält 10.0 und 14.0, die zweite ist leer. Das Anhängen von 18.0 an den ersten Tag erzeugt eine neue Zeile mit drei Werten; die zweite Zeile bleibt leer. Der erste Tagesmittelwert beträgt anschließend 14.0. Wäre der neue Messwert versehentlich an jede Zeile angehängt worden, würde sich nicht nur die Datenstruktur, sondern auch die fachliche Zuordnung der Messung ändern.

Typischer Fehler: Eine neue längere Zeile allein reicht nicht: Solange ihre Referenz nicht im äußeren Array gespeichert wird, zeigt dieses weiterhin auf die alte Zeile.

Verständnisfrage: Warum beginnt die einfache Präfixsummenschleife beim Index eins?

Antwort: Das erste Element ist bereits die Summe des ersten Präfixes. Außerdem benötigt jeder weitere Schritt den existierenden Vorgänger am Index i minus eins.

A07 Arrays: Grundlagen, Änderungen an Ort und Stelle und Messdaten

Leere Arrays, Invarianten und geteilte Zeilen

Du behandelst fehlende Daten ausdrücklich und begründest Arrayänderungen durch Eigenschaften, die während der Verarbeitung erhalten bleiben.

A07.10 Leer, fehlend oder gemessen? · [Ergänzung](#)

Eine Zeile mit Länge 0 enthält keine Messungen; eine Zeile mit dem einzigen Wert 0.0 enthält dagegen genau eine Messung. null bedeutet zusätzlich, dass überhaupt keine Zeilenreferenz vorhanden ist. Wer null wie eine leere Zeile behandeln möchte, muss dies als zusätzliche Konvention festlegen, statt beide Zustände unbegründet gleichzusetzen.

A07.11 Keine Daten bedeuten keinen Mittelwert · [Ergänzung](#)

Für einen Tag ohne Messungen ist der arithmetische Mittelwert nicht definiert. Eine Ersatzrückgabe darf nur als solche dokumentiert werden, damit sie nicht als echte Temperatur verstanden wird. Bei Maximum- und Häufigkeitsberechnungen werden leere Zeilen übersprungen und unterschiedlich lange Zeilen jeweils mit ihrer eigenen Länge durchlaufen.

A07.12 Natürliche Grenzfälle der Schleife · [Ergänzung](#)

Bei einem leeren oder einelementigen Array muss eine Umkehrung nichts tauschen. Bei ungerader Länge bleibt das mittlere Element ebenfalls unverändert, weil es sein eigener Spiegelpartner ist. Auch die Präfixsummenschleife kann solche kurzen Arrays ohne Sonderrechnung behandeln, wenn Startindex und obere Grenze passend gewählt sind.

A07.13 Eine Invariante erklärt das Verfahren · [Ergänzung](#)

Vor einem Präfixsummenschritt enthält die vorherige Position bereits die Summe aller ursprünglichen Elemente bis dorthin. Der aktuelle Wert ist noch unverändert und wird hinzugefügt. Dadurch entsteht genau die nächste Präfixsumme; dieselbe Aussage gilt anschließend für den folgenden Schritt und begründet den gesamten Durchlauf.

A07.14 Eine Referenz ist keine Kopie · [Ergänzung](#)

Die Rückgabe `matrix[i]` liefert das vorhandene Zeilenarray. Änderungen über diese Referenz verändern deshalb auch die Matrix. Eine Spalte liegt dagegen nicht als einzelnes inneres Array vor und muss aus den entsprechenden Elementen mehrerer Zeilen neu zusammengestellt werden, wodurch ein eigenständiges Ergebnisarray entsteht.

Beispiel: Die Änderung ist an beiden Namen sichtbar

Das Fragment speichert in `zeile` dieselbe Referenz wie `matrix[0]`. Deshalb ändert die Zuweisung anschließend den Matrixeintrag an Zeile 0, Spalte eins auf neun. Ein zweiter Variablenname erzeugt keine zweite Zahlenfolge. Würden stattdessen die Werte der Zeile in ein neues Array kopiert, könnte dessen Änderung unabhängig bleiben. Bei Rückgabemethoden muss deshalb klar sein, ob sie Zugriff auf Originaldaten oder eine eigenständige Kopie versprechen.

```
int[][] matrix = {{1, 2}, {3, 4}};
int[] zeile = matrix[0];
zeile[1] = 9;
```

Typischer Fehler: Prüfe vor einer Verwendung von `zeile.length` zunächst die nicht-null-Voraussetzung; Länge 0 und null-Referenz lösen unterschiedliche Fragen aus.

Verständnisfrage: Welcher Zustand begründet, dass `feld[i] += feld[i-1]` die nächste Präfixsumme erzeugt?

Antwort: `feld[i-1]` enthält bereits die vollständige vorherige Präfixsumme, während `feld[i]` noch seinen ursprünglichen Wert besitzt.

A08 Matrizen, Bildtransformationen und zweidimensionale Muster

Vektoren und Matrizen mit Arrays berechnen

Du leitest Ergebnisdimensionen und Indexbeziehungen aus der mathematischen Operation ab, bevor du die nötigen Schleifen formulierst.

A08.01 Dimensionen zuerst bestimmen

Ein Vektor kann durch ein eindimensionales Array und eine Matrix durch ein rechteckiges Array von Zeilen dargestellt werden. Für eine Matrix mit r Zeilen und s Spalten läuft der Zeilenindex bis r minus eins und der Spaltenindex bis s minus eins. Die beiden Grenzen dürfen nicht einfach vertauscht werden.

A08.02 Skalarprodukt verknüpft Komponenten

Das Skalarprodukt zweier gleich langer Vektoren addiert die Produkte zusammengehöriger Komponenten. Für eine Matrixzeile ist die passende Referenz unmittelbar vorhanden; eine Spalte wird aus derselben Spaltenposition aller Zeilen zusammengesetzt. Damit können dieselben Vektormethoden auch als Bausteine für Matrixoperationen verwendet werden.

A08.03 Zeile mal Spalte

Beim Matrixprodukt entsteht jeder Ergebniseintrag aus einer Zeile der linken und einer Spalte der rechten Matrix. Deshalb müssen deren Längen übereinstimmen: Eine r -mal- k -Matrix kann mit einer k -mal- s -Matrix multipliziert werden. Das Ergebnis hat r Zeilen und s Spalten, nicht automatisch dieselbe Form wie einer der beiden Faktoren.

A08.04 Transposition vertauscht Rollen

Die Transponierte besitzt an Position $[s][z]$ genau den ursprünglichen Wert von $[z][s]$. Entsprechend werden auch Zeilen- und Spaltenanzahl vertauscht. Wird ein neues Ergebnisarray erzeugt, bleibt die ursprüngliche Matrix erhalten; eine bloße Änderung der Schleifenreihenfolge allein führt noch keine Transposition aus.

Beispiel: Ein Produkt mit unterschiedlichen Formen

Die linke Matrix besteht aus den Zeilen (1, 2, 0) und (0, 1, 1), also aus zwei Zeilen mit je drei Einträgen. Die rechte Matrix hat drei Zeilen: (2, 1), (1, 0) und (3, 2). Das Ergebnis ist zwei mal zwei. Seine erste Zeile lautet (4, 1), die zweite (4, 2), denn jeder Eintrag entsteht durch ein eigenes Zeilen-Spalten-Skalarprodukt. Dass beide Faktoren unterschiedliche Formen besitzen, ist erlaubt; entscheidend sind die gleichen inneren Dimensionen.

Typischer Fehler: Das Matrixprodukt ist kein elementweises Multiplizieren gleich positionierter Einträge. Schon seine Ergebnisform folgt einer anderen Regel.

Verständnisfrage: Welche Form besitzt die Transponierte einer Matrix mit zwei Zeilen und fünf Spalten?

Antwort: Sie besitzt fünf Zeilen und zwei Spalten. Jedes ursprüngliche Element wechselt dabei von seiner Zeilen-Spalten-Position zur vertauschten Position.

A08 Matrizen, Bildtransformationen und zweidimensionale Muster

Symmetrien, Drehungen und Sternmuster

Du übersetzt geometrische Regeln in Indexbedingungen und unterscheidest die Spiegelung innerhalb einer Zeile von der Symmetrie an der Hauptdiagonalen.

A08.05 Symmetrie an der Diagonalen

Eine Matrix ist symmetrisch zur Hauptdiagonalen, wenn sie quadratisch ist und `feld[z][s]` stets `feld[s][z]` entspricht. Ein Vergleich nur oberhalb der Diagonale genügt, weil jedes Paar sonst doppelt geprüft würde. Die Diagonale selbst stimmt immer mit sich überein und liefert keine neue Information.

A08.06 Horizontale Bildsymmetrie

Bei horizontaler Symmetrie werden innerhalb jeder einzelnen Zeile die erste und letzte, dann die zweite und vorletzte Position verglichen. Das Bild muss dafür nicht quadratisch sein. Bei ungerader Breite besitzt die mittlere Position keinen anderen Spiegelpartner und muss daher nicht gesondert verglichen werden.

A08.07 Nach links drehen

Bei einer Linksdrehung um 90 Grad wird aus der alten Spaltenposition eine umgekehrt gezählte neue Zeilenposition. Für ein Bild mit `s` Spalten gilt `neu[s-1-spalte][zeile] = alt[zeile][spalte]`. Die neue Form vertauscht Höhe und Breite; ein neues Array erhält diese gedrehten Werte.

A08.08 Eine Treppe durch Intervalle

Ein Treppmuster kann in jeder Zeile einen zusammenhängenden Bereich von `n` Sternen enthalten. Beginnt dieser Bereich in Zeile `z` bei Spalte `z`, verschiebt er sich mit jeder weiteren Zeile um genau eine Position nach rechts. Alle Positionen außerhalb dieses Bereichs werden ausdrücklich mit Leerzeichen gefüllt.

A08.09 Gitter aus Paritätsbedingungen

Wenn jede zweite Zeile oder Spalte Sterne enthält, entscheidet die Parität der 0-basierten Indizes. Für die in den Beispielen verwendete Lage sind die ungeraden Indizes vollständig markiert. Das logische Oder ist wesentlich: Ein Stern entsteht bereits durch eine markierte Zeile oder durch eine markierte Spalte, nicht nur an ihren Kreuzungen.

Beispiel: Zwei verschiedene Symmetrien

Das Feld mit den Zeilen (1, 2, 1) und (3, 4, 3) ist horizontal symmetrisch, weil jede Zeile von beiden Seiten gleich gelesen wird. Es ist jedoch keine symmetrische Matrix im Sinn der Transposition, da es nicht quadratisch ist. Diese Unterscheidung verhindert, dass eine passende Prüfung für das eine Problem unbemerkt als Lösung für das andere eingesetzt wird.

Typischer Fehler: Das Wort symmetrisch allein genügt nicht. Bestimme zuerst, welche Achse beziehungsweise welche Indexabbildung die Aufgabe vorgibt.

Verständnisfrage: Warum erzeugt ein logisches Und statt Oder beim Gitter ein anderes Muster?

Antwort: Und markiert nur Positionen, deren Zeile und Spalte beide ausgewählt sind. Dadurch entstehen einzelne Kreuzungspunkte statt vollständig markierter Gitterlinien.

A08 Matrizen, Bildtransformationen und zweidimensionale Muster

Blockmuster und besondere Darstellungsfälle

Du behandelst Blockgrenzen und Speicherrepräsentationen bewusst und prüfst Beispiele mit der Definition statt durch ungeprüfte Übernahme.

A08.10 Blockkoordinaten berechnen

Bei Blockhöhe s und Blockbreite t geben $zeile/s$ und $spalte/t$ die zugehörigen Blockindizes an. Ihre Paritäten entscheiden, ob an einer Position eine 0 oder Eins steht. Die eigentliche Feldgröße ist davon unabhängig; ein Randblock darf deshalb kleiner erscheinen, wenn das Feld mitten im Block endet.

A08.11 Graustufe trotz negativem byte · Ergänzung

Java-byte ist vorzeichenbehaftet und besitzt Werte von -128 bis 127. Ein gewünschter Grauwert oberhalb dieses Bereichs kann dennoch über sein achtbittiges Muster gespeichert werden. Mit dem Ausdruck `wert & 0xff` wird dieses Muster als nichtnegative int-Zahl interpretiert; die Bildgeometrie benötigt dagegen häufig nur Kopieren und Gleichheitsvergleiche.

A08.12 Leere äußere Dimension · Ergänzung

Hat ein zweidimensionales Java-Array keine Zeile, lässt sich daraus keine innere Spaltenzahl ablesen. Eine Methode muss deshalb festlegen, wie sie die Transposition eines solchen Eingabewerts behandelt. Ebenso kann das leere Feld konventionell als symmetrisch gelten, sofern die Aufgabenstellung keine andere Definition verlangt.

A08.13 Initialwerte und sichere Blockrechnung · Ergänzung

Neue char-Arrays enthalten zunächst das Zeichen mit dem Codewert 0 und keine Leerzeichen. Musteralgorithmen müssen ihre Hintergrundzellen deshalb ausdrücklich beschreiben und positive Blockgrößen vor einer Division sicherstellen. Werden beide Blockindizes bereits vor ihrer Addition modulo zwei reduziert, kann ihre Summe nicht durch sehr große Indexwerte überlaufen.

A08.14 Ein Beispiel ist kein Beweis · Ergänzung

Ein angegebenes Zahlenbeispiel kann der danebenstehenden mathematischen Definition widersprechen. Beim Skalarprodukt werden dann die komponentenweisen Produkte unabhängig gebildet und addiert. Eine begründete Korrektur macht den Vorlagenfehler sichtbar; das Verfahren sollte nicht verändert werden, nur um eine falsche Beispielzahl nachzuahmen.

Beispiel: Welche Blockfarbe erhält eine Position?

Für Blöcke mit zwei Zeilen und drei Spalten liegt die Position mit 0-basierten Indizes $zeile = 3$ und $spalte = 4$ in Blockzeile eins und Blockspalte eins. Beide Blockparitäten sind ungerade; ihre Summe modulo zwei ist 0. Die Position erhält deshalb im Muster $(Blockzeile + Blockspalte) \bmod 2$ die 0. Die benachbarte Blockspalte zwei hätte an derselben Blockzeile die andere Farbe.

Typischer Fehler: Eine Blockgröße 0 verursacht keine leeren Blöcke, sondern macht die notwendige Division unzulässig. Vorbedingungen müssen deshalb vor der Rechnung gelten.

Verständnisfrage: Warum kann byte $b = (\text{byte}) 255$ beim Ausgeben negativ erscheinen?

Antwort: Das Bitmuster besteht aus acht Einsen und wird als byte vorzeichenbehaftet interpretiert. Mit $b \& 0xff$ erhält man die zugehörige nichtnegative Graustufenzahl.

A09 Klassen, Konstruktoren, Kennungen und Zugriffsrechte

Objekte initialisieren und Kennungen vergeben

Du trennst individuelle Objektwerte vom gemeinsamen Zustand und verfolgst die Reihenfolge ihrer Initialisierung.

A09.01 Klasse und konkrete Instanz

Eine Klasse beschreibt Attribute und Verhalten ihrer Objekte. `new` erzeugt eine Instanz und führt einen passenden Konstruktor aus. Instanzattribute gehören jeweils einem Objekt; ein `static`-Feld wird dagegen gemeinsam von der Klasse verwendet.

A09.02 Feld und Parameter unterscheiden

Ein gleichnamiger Parameter kann ein Objektfeld im Methodenrumpf verdecken. `this.name = name` beschreibt ausdrücklich das Feld des aktuellen Objekts mit dem Parameterwert. Geerbte und eigene gleichnamige Felder bleiben dabei getrennte Speicherstellen.

A09.03 Konstruktoraufrufe weiterreichen

Überladene Konstruktoren besitzen unterschiedliche Parameterlisten. `super(...)` übergibt passende Werte an den Oberklassenkonstruktor, statt dessen `private` Initialisierung nachzubauen. In mehrstufigen Hierarchien wird die Weitergabe Ebene für Ebene verfolgt, weil sich die Werte unterwegs ändern können.

A09.04 Reihenfolge der Initialisierung

Im gezeigten Konstruktorschema läuft nach den Standardwerten zuerst der Oberklassenkonstruktor, dann folgen Initialisierer und Konstruktorkörper der Unterklasse. Fehlen sowohl `this(...)` als auch `super(...)`, wird implizit `super()` eingefügt. Vertauschte `super`-Argumente verändern die übergebenen Werte; `this(...)` delegiert dagegen an einen anderen eigenen Konstruktor.

A09.05 Gemeinsam zählen, einzeln speichern

Ein `static`-Zähler steigt bei jeder Erzeugung. Die dabei vergebene Nummer wird zusätzlich im Objekt gespeichert, damit spätere Erzeugungen sie nicht verändern. Würden Ausgaben nur den aktuellen Zähler lesen, zeigten ältere Objekte ebenfalls dessen jüngsten Stand.

A09.06 Kennung unveränderlich halten

Ein `private-final`-Feld erhält seine Kennung einmal im Konstruktor und keinen öffentlichen Setter. Postinkrement vergibt den alten Zählerwert und erhöht ihn danach. Der Startwert bestimmt deshalb, mit welcher Kennung die Vergabe beginnt.

Beispiel: Gemeinsamer Zähler, zwei Kennungen

Beginnt `naechste` bei eins, vergibt `id = naechste++` zuerst eins und anschließend zwei. Danach steht der gemeinsame Zähler auf drei; die erste gespeicherte Kennung bleibt eins. `final` schützt die individuelle Zuweisung, während `static` ausschließlich den Vergabezähler verbindet.

Typischer Fehler: Ein gewöhnliches Zählerfeld würde bei jeder Instanz neu beginnen und keine gemeinsamen fortlaufenden Kennungen liefern.

Verständnisfrage: Warum sind Vergabezähler und Kennung verschiedene Felder?

Antwort: Der Zähler ist gemeinsam und veränderlich. Die Kennung gehört zum einzelnen Objekt und soll nach ihrer Vergabe unverändert bleiben.

A09 Klassen, Konstruktoren, Kennungen und Zugriffsrechte

Zugriffsschutz und Grenzen gemeinsamer Zustände

Du prüfst den Zugriffsort und die Empfängertypen und erkennst Voraussetzungen einer einfachen Kennungsvergabe.

A09.07 Vier Zugriffsstufen

public, protected und private geben Zugriffsrechte an; ohne Modifizierer gilt Paketsichtbarkeit. protected erlaubt Paketzugriffe und bestimmte Zugriffe aus Unterklassen außerhalb. private beschränkt den Zugriff auf den zugehörigen Klassenkontext, nicht auf jedes Objekt mit ähnlich benannten Feldern.

A09.08 Kapselung durch Methoden

Private Attribute werden über geeignete Methoden gelesen oder verändert. Ein Setter kann dabei Bedingungen prüfen, statt beliebige Feldzuweisungen zu erlauben. Paketsichtbarkeit entsteht ohne Sichtbarkeitsmodifizierer; das Schlüsselwort package deklariert dagegen die Paketzugehörigkeit, keine Attributeigenschaft.

A09.09 Standardwerte sind nicht universell

Objekt- und Klassenfelder erhalten Standardwerte wie 0 oder null. Lokale Variablen müssen vor einem Lesen ausdrücklich zugewiesen werden. Ein geerbtes static-Feld bleibt gemeinsam, solange eine Unterklasse kein eigenes gleichnamiges Feld deklariert.

A09.10 Geschützter Zugriff außerhalb des Packages · Ergänzung

Außerhalb des Packages genügt für protected-Instanzmember nicht beliebig irgendeine Oberklassenreferenz. Im Unterklassenkontext muss auch der qualifizierende Referenztyp die entsprechende Unterklassenbedingung erfüllen. Der gesamte Zugriff samt verwendeten Typen muss zulässig sein; public allein beantwortet nicht jede Zugriffsfrage.

A09.11 Interne Gegenreferenz pflegen · Ergänzung

Eine bidirektionale Assoziation verlangt abgestimmte Änderungen beider Seiten. Eine Hilfsmethode mit Paketsichtbarkeit kann diese interne Pflege ermöglichen. Zusammengehörige Klassen kooperieren dann über die Hilfsmethode, während normale Aufrufer eine kontrollierte öffentliche Änderungsmethode verwenden.

A09.12 Einfache Zähler haben Grenzen · Ergänzung

Ein int-Zähler besitzt endlich viele Zustände und kann überlaufen. Eine einfache Erhöhung gewährleistet außerdem keine korrekte gleichzeitige Vergabe aus mehreren Ausführungssträngen. Die Klausurlösung setzt deshalb begrenzte, sequenzielle Erzeugung voraus und liefert keine uneingeschränkte Eindeutigkeitsgarantie.

Beispiel: Zugriff hängt auch vom Empfänger ab

Eine Unterklasse B in einem fremden Package kann einen passenden protected-Zugriff über this ausführen. Daraus folgt nicht, dass derselbe Zugriff über jede Variable vom Oberklassentyp A erlaubt wäre. Prüfe deshalb Zugriffsort und deklarierten Empfängertyp getrennt.

Typischer Fehler: Paketsichtbarkeit ist nicht protected: Außerhalb des Packages gelten besondere Zugriffsregeln für Unterklassen.

Verständnisfrage: Warum darf eine uninitialisierte lokale int-Variable nicht als 0 angenommen werden?

Antwort: Automatische Standardinitialisierung gilt für Felder. Ein lokales Lesen ohne vorherige sichere Zuweisung wird beim Übersetzen zurückgewiesen.

A10 Abstraktion, Interfaces, Vererbung und Polymorphie

Abstrakte Klassen und Interfaces als Verträge

Du unterscheidest einen gemeinsamen teilweise implementierten Typ von einer konkreten Instanz und erkennst, welche Methoden noch bereitzustellen sind.

A10.01 Abstraktion richtig einordnen

Eine abstrakte Methode beschreibt eine Operation ohne normalen Implementierungsrumpf in dieser Klasse. Eine Klasse mit einer solchen offenen Verpflichtung muss abstrakt sein. Umgekehrt darf eine Klasse auch dann als abstrakt gekennzeichnet werden, wenn sie ausschließlich konkrete Methoden besitzt; damit wird ihre unmittelbare Instanziierung ausgeschlossen.

A10.02 Abstrakt bedeutet nicht konstruktionslos

Abstrakte Klassen können Attribute, Konstruktoren und bereits implementierte Methoden enthalten. Ihr Konstruktor wird beim Erzeugen einer konkreten Unterklasse zur Initialisierung des geerbten Objektanteils verwendet. Verboten ist nur die direkte Erzeugung einer Instanz der abstrakten Klasse; eine Referenzvariable dieses Typs bleibt dennoch sinnvoll.

A10.03 Interfaces beschreiben Fähigkeiten

Ein Interface legt Operationen fest, auf die sich aufrufender Code verlassen darf. Eine Klasse erklärt mit `implements`, dass sie diesen Vertrag erfüllt. Interfaces haben keine Konstruktoren und werden nicht direkt mit `new` instanziiert; ein Objekt einer implementierenden Klasse kann aber über eine Interface-Referenz angesprochen werden.

A10.04 Mehrere Verträge, eine Klassenbasis

Eine Java-Klasse hat höchstens eine direkte Klassenoberklasse, kann aber mehrere Interfaces implementieren. Ein Interface kann seinerseits mehrere andere Interfaces erweitern. Damit sind mehrere Typverpflichtungen kombinierbar, ohne dass eine Klasse gleichzeitig von mehreren unterschiedlichen Klassenoberteilen erben müsste.

A10.05 Wer muss implementieren?

Eine konkrete Klasse benötigt passende Implementierungen aller noch offenen abstrakten Methoden. Diese können direkt in ihr geschrieben oder bereits aus einer Oberklasse geerbt sein. Eine abstrakte Zwischenklasse darf solche Verpflichtungen dagegen teilweise offenlassen, sodass erst eine spätere konkrete Unterklasse sie vollständig erfüllt.

Beispiel: Eine Form ohne allgemeine Flächenformel

Die abstrakte Klasse `Form` kann einen Namen speichern und eine konkrete Namensausgabe besitzen. Die Operation `flaeche` bleibt abstrakt, weil für unterschiedliche Formen verschiedene Berechnungen nötig sind. `Rechteck` implementiert sie mit `Länge mal Breite`. Der Ausdruck `new Rechteck(...)` erzeugt ein konkretes Objekt, während `new Form(...)` unzulässig ist. Eine Variable `Form f` darf trotzdem auf das `Rechteck` zeigen und dessen vorhandene Flächenoperation nutzen.

Typischer Fehler: Das Wort `abstrakt` verbietet weder Konstruktoren noch konkrete Methoden. Es kennzeichnet, dass dieser Klassentyp nicht unmittelbar als Objekt erzeugt werden darf.

Verständnisfrage: Muss eine konkrete Unterklasse jede geerbte Interface-Methode erneut in ihrem eigenen Quelltext schreiben?

Antwort: Nein. Eine bereits geerbte passende öffentliche Implementierung kann die Verpflichtung erfüllen. Entscheidend ist ihre Verfügbarkeit, nicht ihre erneute Abschrift in jeder Unterklasse.

A10 Abstraktion, Interfaces, Vererbung und Polymorphie

Dynamische Bindung und Interface-Regeln

Du verfolgst die tatsächlich gewählte Methode und ordnest Interface-Regeln nach ihrer Java-Version ein.

A10.06 Der Objekttyp entscheidet

Bei überschriebenen Instanzmethoden entscheidet der tatsächliche Objekttyp über die Implementierung. Oberklassen- und Interface-Referenzen ändern diese dynamische Auswahl nicht. Ihr statischer Typ muss den Aufruf aber zunächst erlauben; dynamische Bindung ersetzt keine Signaturprüfung.

A10.07 super wirkt nur auf diesen Aufruf

`super.methode(...)` wählt die Oberklassenimplementierung für diesen Aufruf. Ruft diese anschließend eine weitere überschreibbare Methode unqualifiziert auf, kann erneut die Unterklassenimplementierung gewählt werden. `super` schaltet dynamische Bindung nicht dauerhaft für Folgeaufrufe ab.

A10.08 Verschiedene Objekte gemeinsam berechnen

Ein Obertyp-Array kann unterschiedliche Implementierungen enthalten. Pflanzen liefern etwa ihren Flächenbedarf, Bücher und CDs ihre jeweiligen Medienwerte. Die Summenschleife verwendet nur die vereinbarte Operation; das angesprochene Objekt bestimmt deren konkrete Berechnung.

A10.09 Interface-Hilfsmethoden zeitlich einordnen · Ergänzung

Seit Java 8 können default-Methoden Implementierungen bereitstellen; seit Java 9 sind private Hilfsmethoden mit Rumpf erlaubt. Private abstrakte Interface-Methoden sind unzulässig. Aussagen älterer Aufgaben müssen deshalb anhand ihrer vorausgesetzten Java-Version eingeordnet werden.

A10.10 Öffentlicher Vertrag · Ergänzung

Gewöhnliche abstrakte Interface-Methoden sind implizit öffentlich und müssen öffentlich implementiert werden. Eine Klasse darf diese Sichtbarkeit nicht einschränken. Statische und private Interface-Methoden sind dagegen keine offenen Instanzoperationen, die jede implementierende Klasse nachbilden müsste.

A10.11 Selbstaufzuruf genau verfolgen · Ergänzung

Ein unqualifizierter Selbstaufzuruf wird ebenfalls dynamisch gebunden und kann deshalb die überschreibende Unterklassenmethode erreichen. Ob von dort weitere Rekursion ausgeht, entscheidet deren Rumpf. Der gleiche Methodenname allein beweist keine fortgesetzte rekursive Ausführung.

Beispiel: Die Oberklasse ruft wieder die Unterklasse

A definiert `ausgabe()`, das `wert()` aufruft; B überschreibt `wert()` und liefert zwei. Ein B-Objekt verwendet auch innerhalb der geerbten `ausgabe`-Methode sein überschriebenes `wert()`. Der Methodenrumpf stammt zwar aus A, aber `this` bleibt dasselbe B-Objekt. Die Referenz kann dabei durchaus als A deklariert sein.

Typischer Fehler: Eine statische Interface-Hilfsmethode ist keine abstrakte Instanzoperation des implementierenden Objekts.

Verständnisfrage: Gilt „Interfaces dürfen keine privaten Methoden enthalten“ für alle Java-Versionen?

Antwort: Nein. Für Java 8 trifft es zu; seit Java 9 dürfen private Interface-Hilfsmethoden einen Rumpf besitzen.

Übungsbezug: Klausuren 09, 10, 11, 12, 15, 16, 17, 20 · Teilaufgaben siehe Themenübersicht A10

A11 Referenztypen, Methodenauflösung und statische Typfehler

Überladung, Feldverdeckung und statischer Typ

Du entscheidest zuerst, welcher Aufruf übersetzbar ist, und erst danach, welche Implementierung das konkrete Objekt zur Laufzeit verwendet.

A11.01 Parameterlisten unterscheiden Methoden

Überladene Methoden tragen denselben Namen, unterscheiden sich jedoch in Anzahl, Typen oder Reihenfolge ihrer Parameter. Unterschiedliche Rückgabetypen allein genügen nicht, weil der Aufruf dadurch nicht eindeutig einer Signatur zugeordnet werden könnte. Die Auswahl passender Überladungen erfolgt anhand der beim Übersetzen bekannten Typen der Argumente und des Empfängers.

A11.02 Zwei Auswahlstufen beachten

Zuerst wird geprüft, welche Methoden der deklarierte Empfängertyp überhaupt anbietet und welche Signatur zu den Argumenten passt. Danach kann eine überschreibende Implementierung dieser Signatur dynamisch gewählt werden. Eine nur in der Unterklasse neu ergänzte Überladung wird deshalb nicht automatisch über eine Oberklassenreferenz verfügbar.

A11.03 Felder werden nicht virtuell gewählt

Deklarieren Ober- und Unterklasse jeweils ein Feld `m`, besitzt das Objekt zwei getrennte Felder. Ein Feldzugriff über eine Referenz wählt anhand ihres statischen Typs das passende Feld. Die dynamische Auswahl überschreibender Methoden darf daher nicht auf gewöhnliche Feldzugriffe übertragen werden; gleiche Namen bedeuten keine gemeinsame Speicherstelle.

A11.04 Getter behalten ihren Feldbezug

Ein in der Oberklasse definierter Getter, der deren `privates m` zurückgibt, greift auf genau dieses Feld zu. Ein zusätzliches `m` in der Unterklasse verändert den Rumpf des geerbten Getters nicht. Nur eine tatsächliche Überschreibung des Getters könnte beim Aufruf eine andere Implementierung mit anderem Feldbezug einsetzen.

A11.05 Abstrakter Variablentyp ist erlaubt

Eine Referenzvariable kann einen abstrakten Klassen- oder Interface-Typ besitzen und auf ein konkretes Objekt zeigen. Über sie darf eine dort deklarierte abstrakte Operation aufgerufen werden, weil das konkrete Objekt die Implementierung bereitstellt. Die Deklaration einer solchen Variablen ist von der verbotenen direkten Instanziierung ihres abstrakten Typs zu unterscheiden.

Beispiel: Zwei Felder im selben Objekt

Das B-Objekt besitzt im Fragment `A.m` mit dem Wert eins und `B.m` mit dem Wert zwei. Die Variable `a` ist als `A` deklariert, daher liefert `a.m` den Wert eins. Ein Zugriff über eine `B`-Referenz würde dagegen `B.m` wählen. Ein aus `A` geerbter Getter für `A.m` würde weiterhin eins zurückgeben, sofern `B` diesen Getter nicht überschreibt. Der Referenztyp verändert keine gespeicherten Werte, sondern nur, welcher deklarierte Zugriff verwendet wird.

```
class A { int m = 1; }  
class B extends A { int m = 2; }  
A a = new B();
```

Typischer Fehler: Eine neue Überladung ist keine Überschreibung. Prüfe die vollständige Parameterliste, bevor du dynamische Bindung auf einen ähnlich benannten Aufruf anwendest.

Verständnisfrage: Kann man `f(int)` und `f(int)` allein durch Rückgabetypen `int` und `double` überladen?

Antwort: Nein. Beide besitzen dieselbe Parameterliste; verschiedene Rückgabetypen machen diese beiden Deklarationen nicht zu zulässigen unterschiedlichen Überladungen.

A11 Referenztypen, Methodenauflösung und statische Typfehler

Zuweisungen, Typprüfungen und saubere Korrekturen

Du begründest Übersetzungsfehler aus Typbeziehungen und hältst eine minimale Reparatur von einer Änderung des gesamten Programmverhaltens auseinander.

A11.06 Richtung einer Zuweisung

Ein Unterklassenobjekt kann einer Oberklassenreferenz zugewiesen werden, weil es deren Typanforderungen erfüllt. Ein beliebiges Oberklassenobjekt ist dagegen nicht automatisch ein Unterklassenobjekt. Nicht verwandte konkrete Klassen lassen sich ebenfalls nicht einfach gegenseitig zuweisen; ähnliche Attribute oder Methodennamen begründen keine Vererbungsbeziehung.

A11.07 Interface-Zugehörigkeit nachweisen

Eine Interface-Referenz akzeptiert Objekte, deren Klasse den Vertrag direkt oder über ihre Oberklasse implementiert. Der Name einer passenden Methode allein ersetzt diese erklärte Typbeziehung nicht. Auch über eine solche Referenz sind nur die an ihrem statischen Typ verfügbaren Operationen zulässig; das Interface selbst wird nicht unmittelbar mit `new` erzeugt.

A11.08 Erst prüfen, dann casten

`instanceof` kann feststellen, ob ein Objekt zu einem benötigten Laufzeittyp gehört; für `null` ist die Prüfung `false`. Nach erfolgreicher Prüfung kann ein expliziter Cast einen entsprechend typisierten Zugriff ermöglichen. Beispielsweise werden nur Fensterreinigungen auf ihre speziellen Fensterzuordnungen untersucht, während andere Einträge übersprungen werden.

A11.09 Nur den Empfänger korrigieren · Ergänzung

Eine nur in B deklarierte Überladung ist über den statischen Typ A nicht verfügbar; ohne andere passende Signatur scheitert der Aufruf an der Übersetzung. Eine am Typ A passende andere Überladung könnte dagegen gewählt werden. Ein Empfängercast nach B macht dessen Signatur zugänglich, sofern das Objekt tatsächlich ein B ist, und verändert weder Objekt noch Felder.

A11.10 Reparaturwirkung offenlegen · Ergänzung

Eine nicht kompilierbare Vorlage besitzt in ihrer wörtlichen Form keine reguläre Programmausgabe. Wird sie korrigiert, muss diese Fassung ausdrücklich benannt werden. Das Ändern einer gesamten Variablendeklaration von A zu B kann mehr Zugriffe beeinflussen als ein einzelner Cast, insbesondere die statische Auswahl verdeckter Felder.

Beispiel: Ein sicher eingegrenzter Spezialfall

Angenommen, ein Putzplan speichert Einträge verschiedener Klassen unter dem gemeinsamen Typ `Eintrag`. Benötigt eine Auswertung die Fensterliste, wird zuerst geprüft, ob der aktuelle Eintrag eine Fensterreinigung ist. Nur in diesem Zweig wird er entsprechend umgewandelt und seine besondere Methode verwendet. Eine Bodenreinigung wird dadurch nicht künstlich zu einer Fensterreinigung; sie fällt einfach nicht in den geprüften Zweig.

Typischer Fehler: Ein Cast ist keine Objekterzeugung und keine automatische Garantie. Ein unpassender Cast kann erst zur Laufzeit scheitern, obwohl er syntaktisch zulässig ist.

Verständnisfrage: Warum kann eine Änderung von `A a` zu `B a` spätere Ausgaben beeinflussen?

Antwort: Spätere Feldzugriffe über `a` werden danach anhand von `B` statt `A` gebunden. Ein einzelner Methodenempfängercast hätte diese anderen Zugriffe unverändert gelassen.

A12 Speicherabbilder, Referenzen und objektorientierte Ablaufanalyse

Ein vollständiges Speicherabbild aufbauen

Du hältst Objekte, gemeinsame Felder und aktive Aufrufe auseinander und ordnest jede lokale Referenz ihrem tatsächlichen Ziel zu.

A12.01 Drei Bereiche unterscheiden

Ein didaktisches Speicherabbild trennt gemeinsame Klassenfelder, Heapobjekte und aktive Methodenrahmen. In jedem Objekt werden sowohl eigene als auch geerbte Felder eingetragen, bei gleichem Namen mit ihrer deklarierenden Klasse. Dieses Modell erklärt Programmmustände; es ist keine Behauptung über jede interne Speicheroptimierung einer konkreten JVM.

A12.02 Pfeile statt Objektkopien

Referenzvariablen in main, Methodenparameter und this werden als Verweise auf vorhandene Objekte dargestellt. Die Übergabe einer Referenz erzeugt kein zweites Objekt. Auch `String[] args` ist eine Referenz, deren konkreter Inhalt nur dann eingetragen wird, wenn die Aufgabenstellung ihn vorgibt; er darf nicht frei erfunden werden.

A12.03 Erzeugungsreihenfolge beachten

Soll ein neues Objekt bereits auf ein anderes zeigen, muss die passende Referenz bei seinem Konstruktoraufruf verfügbar sein. So kann zuerst der Tiger und danach die Ziege mit diesem Freund erzeugt werden. `null` wird als fehlender Verweis gekennzeichnet und nicht als eigenständiges Zielobjekt gezeichnet.

A12.04 Empfänger und Parameter trennen

Beim Aufruf `a.gleich(b)` bezeichnet `this` das Objekt hinter `a`, während der entsprechende Parameter auf das Objekt hinter `b` zeigt. Unqualifizierte Instanzfelder beziehen sich auf den Empfänger. Zugriffe über den Parameter lesen dagegen das andere Objekt, sodass identische Variablennamen im umgebenden Programm nicht verwechselt werden dürfen.

A12.05 Nur aktive Rahmen eintragen

Direkt vor `return` ist der aktuelle Methodenaufruf noch aktiv, sein Aufrufer wartet gegebenenfalls auf das Ergebnis. Ein verschachtelter `super`-Aufruf besitzt einen eigenen Rahmen mit eigenen Parametern und lokalen Variablen. Objekte werden dabei nicht verdoppelt: Beide Rahmen können dasselbe `this` referenzieren, obwohl ihre lokalen Werte verschieden sind.

Beispiel: Zwei Rahmen, ein Empfänger

`B.summe` initialisiert eine lokale Variable `erg` und ruft danach `super.summe` auf. Während `A.summe` arbeitet, sind beide Rahmen aktiv. Jeder enthält eine eigene Variable namens `erg`; der Wert im wartenden B-Rahmen wurde durch die noch ausstehende Rückgabe nicht automatisch ersetzt. Beide `this`-Referenzen zeigen jedoch auf dasselbe B-Objekt. Die Felder in dessen A-Anteil und B-Anteil werden nur einmal im Heap eingezeichnet.

Typischer Fehler: Gleiche Namen in unterschiedlichen Aufrufrahmen bedeuten nicht dieselbe Variable. Gleiche Referenzziele bedeuten dagegen tatsächlich dasselbe Objekt.

Verständnisfrage: Wo steht ein `static`-Zähler im Speicherabbild?

Antwort: Im gemeinsamen Klassenbereich, nicht als getrennte Kopie in jedem Objekt. Seine Veränderungen durch mehrere Konstruktoraufrufe betreffen dieselbe Speicherstelle.

A12 Speicherabbilder, Referenzen und objektorientierte Ablaufanalyse

Aufrufspuren, Rückgaben und Wertkopien

Du rekonstruierst den Zustand zwischen Methodenaufruf und Rückgabe und erkennst, welche Änderungen nur lokale Kopien betreffen.

A12.06 Erzeugung vor Ausgabe verfolgen

Vor einer Ausgabespur werden die Konstruktoraufrufe und Feldinitialisierungen in ihrer tatsächlichen Reihenfolge abgearbeitet. Ein gemeinsamer static-Zähler muss dabei nach jeder Änderung aktualisiert werden. Erst danach lässt sich zuverlässig bestimmen, welche Werte spätere Getter oder direkte Feldzugriffe lesen.

A12.07 Jeden Ausdruck getrennt auswerten

Bei mehrstufigen Hierarchien werden Überladung, dynamische Bindung und statischer Feldzugriff für jede Ausgabestelle neu geprüft. Ein super-Aufruf und ein Präinkrement können zusätzliche Änderungen bewirken. Eine einmal gefundene Zahl darf deshalb nicht unverändert für spätere Zeilen übernommen werden, ohne die dazwischenliegenden Anweisungen zu betrachten.

A12.08 Zuweisung wartet auf das Ergebnis · Ergänzung

Während die rechte Seite einer Zuweisung noch läuft, ist der neue Zielwert noch nicht gespeichert. Eine bislang uninitialisierte lokale Variable bleibt deshalb uninitialisiert; eine bereits gesetzte behält ihren früheren Wert. Erst die normale Rückkehr des Aufrufs ermöglicht die abschließende Übernahme seines Ergebnisses.

A12.09 Beendete Aufrufe entfernen · Ergänzung

Ein Getter, der seinen Wert bereits zurückgegeben hat, besitzt am später betrachteten Zeitpunkt keinen aktiven Rahmen mehr. Wartende Aufrufer bleiben dagegen vorhanden. Bei einem zusammengesetzten Ausdruck wird deshalb genau festgehalten, welche Teilergebnisse schon berechnet sind und welche Aktualisierung noch von einem Unteraufruf abhängt.

A12.10 Geteilte Referenzen erkennen · Ergänzung

Mehrere Objekte können dasselbe String-Literal referenzieren, ohne miteinander identisch zu sein. Im Abbild stehen dann getrennte Objekte mit Pfeilen auf einen gemeinsamen String. Außerdem kann ein bereits erzeugtes Objekt weiterhin erreichbar sein, obwohl ein späterer Ausschnitt eines UML-Diagramms es nicht nochmals zeigt.

A12.11 Primitive Parameter sind Kopien · Ergänzung

Ein primitiver Parameter erhält beim Aufruf eine Wertkopie seines Arguments. Wird ein double-Parameter innerhalb der Methode verändert, verändert dies nicht automatisch ein Objektfeld, aus dem der ursprüngliche Wert stammte. Eine Feldänderung müsste durch einen gesonderten Zugriff auf genau dieses Feld erfolgen.

Beispiel: Der Parameter ändert sich, das Feld bleibt

Ein Objektfeld geschwindigkeit enthält 12.0 und wird als Argument an eine Methode ueberarbeite(double v) übergeben. Setzt diese Methode v auf 3.0, ist nur ihre lokale Kopie verändert. Nach der Rückkehr enthält geschwindigkeit weiterhin 12.0, sofern die Methode nicht zusätzlich das Feld selbst beschrieben hat.

Typischer Fehler: Eine Methodenvariable mit demselben Namen wie ein Feld wird dadurch nicht zum Feld. Prüfe, welchen Speicherort eine Zuweisung tatsächlich bezeichnet.

Verständnisfrage: Welchen Wert besitzt x während f() arbeitet, wenn zuvor $x = 7$ und anschließend $x = f()$ ausgeführt wird?

Antwort: x besitzt zunächst weiterhin den alten Wert sieben. Der von f zurückgegebene Wert wird erst nach dessen normaler Beendigung zugewiesen.

A13 UML und allgemeine Assoziationsverwaltung

Ein Klassendiagramm in Java übersetzen

Lernziel: Du liest die Bausteine eines UML-Modells und leitest daraus einen gültigen Anfangszustand deiner Objekte ab.

A13.01 Vom Diagramm zur Deklaration

Ein UML-Klassendiagramm beschreibt, welche Daten und Operationen zu einer Klasse gehören. Ein Attribut wird zu einem Java-Feld; eine Operation wird zu einer Methode mit genau den angegebenen Parametern und dem passenden Rückgabotyp. Übertrage zuerst die Deklarationen und ergänze danach die Berechnungen in den Methoden.

A13.02 Vererbung und Realisierung

Eine Vererbungsbeziehung bedeutet, dass die Unterklasse eine Spezialisierung der Oberklasse ist; in Java steht dafür `extends`. Eine Interface-Realisierung legt einen Vertrag fest und wird mit `implements` umgesetzt. Kursiv geschriebene Klassen oder Operationen kennzeichnen in den verwendeten Diagrammen Abstraktheit; konkrete Unterklassen müssen offene abstrakte Operationen erfüllen.

A13.03 Sichtbarkeit und Beziehungen

Die Zeichen `+`, `-`, `#` und `~` stehen für `public`, `private`, `protected` und Paketsichtbarkeit. Unterstrichene Attribute gehören zur Klasse und erhalten `static`. Eine Multiplizität an einem Assoziationsende sagt, wie viele Objekte dieses Endes zu einem Objekt am anderen Ende gehören; ein Navigationspfeil zeigt die vorgesehene Zugriffsrichtung.

A13.04 Ein Array als Verwaltungsspeicher

Ein Referenzarray reserviert Plätze, erzeugt aber keine Fachobjekte. Bei lückenlosem Bestand speichert ein Zähler anzahl die Belegung: Gültige Referenzen stehen zwischen 0 und `anzahl-1`. Ergänze nur die dafür notwendigen Verwaltungsattribute und die laut Aufgabe erlaubten Methoden; zusätzliche Komfortoperationen sind nicht automatisch vorgesehen.

A13.05 Mindest- und Höchstzahlen erhalten

Eine Mindestzahl muss schon nach erfolgreicher Konstruktion erfüllt sein, beispielsweise durch ein notwendiges Startobjekt als Parameter. Beim Entfernen darf der Bestand nicht unter die Untergrenze fallen. Vor dem Einfügen wird die Obergrenze geprüft; erst nach einer erfolgreichen Prüfung dürfen Array und Zähler gemeinsam geändert werden.

Beispiel: eine begrenzte Sammlung

Die Beziehung Regal zu Regalbrett kann eine Mindestzahl von zwei Brettern fordern. Ein leer konstruiertes Regal würde diese Bedingung verletzen, selbst wenn es später zwei Bretter bekäme. Deshalb nimmt sein Konstruktor gültige Startbretter entgegen und prüft deren Anzahl. Die unten gezeigten Felder beschreiben nur den Speicher; die Bedingungen müssen zusätzlich durch die Methoden gesichert werden.

```
private Regalbrett[] bretter = new Regalbrett[10];  
private int anzahl;
```

Typischer Fehler: Eine Arraykapazität von zehn bedeutet nicht, dass bereits zehn Fachobjekte vorhanden sind. Kapazität und Belegungszahl haben verschiedene Aufgaben.

Verständnisfrage: Warum reicht die Prüfung der Obergrenze allein nicht aus?

Antwort: Weil ein Modell auch Pflichtbeziehungen verlangen kann. Konstruktion und Entfernen müssen daher zusätzlich die jeweilige Mindestzahl erhalten.

A13 UML und allgemeine Assoziationsverwaltung

Assoziationen sicher verändern

Lernziel: Du hältst Referenzsammlungen konsistent und erkennst, welche Annahmen eine Verwaltungsoperation benötigt.

A13.06 Einfügen und lückenloses Entfernen

Zum Anhängen an einen belegten Präfix wird das neue Objekt an der Position `anzahl` gespeichert und anschließend der Zähler erhöht. Beim Entfernen rücken die nachfolgenden Referenzen einen Platz nach links. Danach wird der Zähler verkleinert und der nun freie letzte Platz auf `null` gesetzt, damit dort keine veraltete Referenz bleibt.

A13.07 Identität und ungültige Referenzen

Soll dasselbe Objekt nur einmal zugeordnet werden, vergleicht man Referenzen mit `==`. Zwei getrennte Objekte mit gleichen Attributwerten bleiben dabei verschiedene Elemente. Eine `null`-Referenz bezeichnet kein zuordenbares Objekt; sie wird vor einer Änderung zurückgewiesen, wenn die Schnittstelle ausschließlich echte Objekte erlaubt.

A13.08 Den Methodenvertrag einhalten

Ein boolean-Rückgabewert beschreibt gewöhnlich Erfolg oder Misserfolg; ein Index kann die Einfügestelle nennen. Die Werte `-1` und `null` werden in manchen Aufgaben als Fehlsignale vereinbart und müssen von regulären Ergebnissen unterscheidbar sein. Verwende diese Konventionen genau wie vorgegeben und ersetze geforderte eigene Arrays nicht durch verbotene Bibliothekssammlungen.

A13.09 Defensive Kopien · Ergänzung

Eine direkte Rückgabe des internen Verwaltungsarrays erlaubt fremde Änderungen an der Sammlung. Eine Kopie der Arrayhülle verhindert das; auch ein als Konstruktorargument übergebenes Array sollte bei entsprechendem Schutzbedarf in eigenen Speicher übernommen werden. Die kopierten Referenzen zeigen weiterhin auf dieselben Fachobjekte, deren Zustand durch diese flache Kopie nicht geschützt wird.

A13.10 Annahmen sichtbar machen · Ergänzung

Eine Kapazitätsrechnung setzt meist nichtnegative Größen und gültige Konstruktorargumente voraus. Ob eine ungültige Eingabe abgewiesen oder bereits durch eine Vorbedingung ausgeschlossen wird, richtet sich nach der Aufgabe. Ein Wert genau an der Kapazitätsgrenze ist zulässig, wenn lediglich eine Überschreitung verboten ist.

Beispiel: aus der Mitte entfernen

Die Sammlung enthält die Referenzen `a`, `b` und `c`; `anzahl` ist 3. Wird `b` an Index 1 entfernt, wird `c` von Index 2 nach Index 1 kopiert. Anschließend sinkt `anzahl` auf 2 und Position 2 wird geleert. `a` und `c` werden dabei nicht neu erzeugt: Nur die Verweise im Array ändern ihren Platz.

```
vorher: [a, b, c, null]   anzahl = 3
nachher: [a, c, null, null] anzahl = 2
```

Typischer Fehler: Eine defensive Arraykopie ist keine tiefe Kopie sämtlicher gespeicherter Objekte. Diese beiden Schutzstufen dürfen nicht verwechselt werden.

Verständnisfrage: Woran erkennt ein Aufrufer, dass eine Einfügemethode mit Rückgabety `int` gescheitert ist?

Antwort: An dem vereinbarten Fehlsignal, beispielsweise `-1`. Die Bedeutung muss aus dem Methodenvertrag hervorgehen; sie ist nicht automatisch durch `int` festgelegt.

Übungsbezug: Klausuren 09, 10, 11, 12, 15, 16, 17, 20 · Teilaufgaben siehe Themenübersicht A13

A14 UML-Modelle: InternetRadio, Lager und MediaPlayer

Favoriten, Behälter und Medien modellieren

Lernziel: Du überträgst dieselben Verwaltungsprinzipien auf unterschiedliche Fachmodelle und nutzt gemeinsame Operationen polymorph.

A14.01 InternetRadio und IPGerät

Das InternetRadio übernimmt gemeinsame Geräteeigenschaften von seiner Oberklasse und erfüllt zusätzlich den Vertrag des Interfaces IPGerät. Ein Interface beantwortet hier, welche Operationen verfügbar sein müssen, etwa das Bereitstellen einer IP-Adresse. Die Favoriten sind eigene Senderobjekte; ihre Bezeichnung und ihr Genre werden über passende Zugriffe bereitgestellt.

A14.02 Favoriten als belegter Präfix

Ein neuer Sender wird nur bei freier Kapazität erzeugt beziehungsweise aufgenommen und am Ende der belegten Favoriten gespeichert. Beim Löschen des ersten Favoriten rücken alle übrigen Referenzen nach links. Ein indexbasierter Getter muss den tatsächlichen Bestand prüfen; ein Index innerhalb der Arraykapazität kann trotzdem außerhalb der gültigen Favoriten liegen.

A14.03 Ein Lager verwaltet Behälter

Das Lager hält Referenzen auf Behälter und kontrolliert deren erlaubte Anzahl. Für das Gesamtvolumen durchläuft es die belegten Einträge und ruft jeweils die gemeinsame Volumenoperation auf. Welche Berechnung tatsächlich ausgeführt wird, entscheidet der konkrete Behältertyp; das Lager benötigt keine eigene Fallunterscheidung für jede Form.

A14.04 Karton und Dose spezialisieren

Eine abstrakte Behälterklasse sammelt gemeinsame Angaben wie die Aufschrift und verlangt eine konkrete Volumenberechnung. Ein Karton berechnet das Volumen aus Breite, Höhe und Tiefe. Seine Konstruktion muss sowohl den gemeinsamen Oberklassenanteil als auch die eigenen Abmessungen initialisieren; eine Dose liefert eine für ihre Form passende Implementierung.

A14.05 MediaPlayer mit Speicherbudget

Audio und Video können über Referenzen des gemeinsamen abstrakten Typs Medium verwaltet werden. Neben der Anzahl der Medien besitzt der Player ein begrenztes Speicherbudget. Eine freie Arrayposition allein genügt daher nicht zur Aufnahme: Auch der verbleibende Speicher muss für den Verbrauch des neuen Mediums ausreichen.

Beispiel: gemeinsamer Aufruf, verschiedene Berechnung

Ein Lager enthält einen Karton mit den Abmessungen 2, 3 und 4 Längeneinheiten und einen anderen Behälter mit Volumen 10. Der Karton liefert 24; die Summe des Lagers beträgt 34 Volumeneinheiten. Das Lager kennt nur den Methodenvertrag und summiert die Rückgaben. Eine neue Behälterart lässt sich dadurch ergänzen, ohne die Summenschleife nach jeder Form zu verzweigen.

Typischer Fehler: Ein gültiger Arrayindex ist nicht automatisch ein gültiger Favoritenindex. Für einen belegten Präfix ist `index < anzahl` entscheidend.

Verständnisfrage: Warum sollte das Lager das Kartenvolumen nicht selbst anhand eines `instanceof`-Tests berechnen?

Antwort: Die Volumenoperation gehört zur jeweiligen Behälterklasse. Dynamische Bindung wählt bereits die passende Berechnung und hält das Lager unabhängig von konkreten Formen.

A14 UML-Modelle: InternetRadio, Lager und MediaPlayer

Speicher buchen und Lücken verwalten

Lernziel: Du erkennst die Invarianten einer begrenzten Sammlung und buchst Ressourcen beim Einfügen und Entfernen symmetrisch.

A14.06 Speicher abbuchen und zurückgeben

Vor dem Einfügen prüft der MediaPlayer gleichzeitig eine freie Position und ausreichenden freien Speicher. Erst beim Erfolg wird der Verbrauch abgezogen und die Anzahl erhöht. Beim Entfernen werden die gespeicherte Referenz zurückgegeben, ihr Verbrauch wieder gutgeschrieben und die Anzahl verringert; ein erfolgloser Suchvorgang verändert nichts.

A14.07 Ersten Namenstreffer finden

Die Suche läuft von kleinen zu großen Indizes und beendet sich beim ersten inhaltlich passenden Namen. Dadurch wird unter mehreren Treffern der kleinste Index gewählt. Freie Positionen dürfen übersprungen und später wiederverwendet werden; der Zähler allein verrät bei einem lückenhaften Bestand nicht die Position aller Medien.

A14.08 Grenze einer einseitigen Zuordnung · Ergänzung

Kennt ein Lager lediglich seine eigene Behälterliste, kann es nicht ohne weitere Information sämtliche anderen Lager überprüfen. Die gezeigte Schnittstelle benötigt deshalb die Annahme, dass ein aufzunehmender Behälter noch keinem anderen Lager gehört. Eine umfassendere Eigentumsprüfung wäre eine zusätzliche Modellentscheidung, etwa mit einem Rückverweis im Behälter.

A14.09 Verbrauch muss zur Buchung passen · Ergänzung

Wird beim Entfernen der aktuelle Verbrauch zurückgebucht, muss er dem beim Einfügen abgezogenen Verbrauch entsprechen. Ein nachträglich veränderter Wert würde das freie Budget verfälschen. Die einfache Lösung setzt deshalb einen unveränderten, nichtnegativen Verbrauch während der Speicherung voraus; sonst müsste die ursprüngliche Buchung separat festgehalten werden.

A14.10 Präfix und Lücken unterscheiden · Ergänzung

Die Favoritenliste hält ihre belegten Referenzen lückenlos zusammen, während der MediaPlayer freie Löcher zulassen kann. Entsprechend unterscheiden sich Durchlauf, Einfügesuche und Löschoption. Vor einem Namensvergleich muss außerdem feststehen, dass ein Medium vorhanden ist; ein null-Name benötigt eine eigene erlaubte Vergleichskonvention.

Beispiel: zwei unabhängige Grenzen

Ein Player hat noch eine freie Position und 8 Speichereinheiten. Ein Medium mit Verbrauch 9 wird abgewiesen; ein Medium mit Verbrauch 8 kann aufgenommen werden und lässt null freie Speichereinheiten übrig. Wird es später mit unverändertem Verbrauch entfernt, stehen wieder 8 Einheiten zur Verfügung. Die Gleichheit mit dem Restbudget ist also ein gültiger Grenzfall.

Typischer Fehler: Nach dem ersten freien Platz abzurechnen ist bei der Mediensuche falsch: Hinter einer Lücke können weitere gültige Medien stehen.

Verständnisfrage: Ein Medium benötigt beim Einfügen 4 Einheiten und beim Entfernen angeblich 7. Welcher Fehler entsteht bei einfachem Zurückbuchen?

Antwort: Es würden 3 Einheiten zu viel gutgeschrieben. Das Budget wäre größer als vor der Aufnahme und entspräche nicht mehr dem tatsächlichen Bestand.

A15 UML-Modelle: Garten, Regal und Gebäudeflächen

Garten und Regal: Grenzen auf zwei Ebenen

Lernziel: Du modellierst Mengenbegrenzungen und erkennst, dass verschiedene Objekttypen gemeinsame messbare Eigenschaften anbieten können.

A15.01 Pflanzen über einen Vertrag verwalten

Ein Pflanze-Interface beschreibt die Operation, mit der der Flächenbedarf einer Pflanze abgefragt wird. Ein konkreter Busch implementiert diesen Vertrag und kann über eine Pflanze-Referenz gespeichert werden. Der Garten wahrt sowohl die vorgeschriebene Mindestzahl als auch die Höchstzahl seiner Pflanzen; Konstruktion, Aufnahme und gegebenenfalls Entfernen müssen dazu passen.

A15.02 Flächenbedarf addieren

Die Gartenfläche ergibt sich im einfachen Modell aus Länge mal Breite. Die Methode `testeFlaeche` prüft, ob die Summe der Pflanzenflächen diesen Wert nicht überschreitet; auch Gleichheit liefert `true`. Gefordert ist diese Bestandsprüfung, nicht automatisch eine Flächenkontrolle beim Hinzufügen. Die Flächensumme beschreibt außerdem keine konkrete räumliche Anordnung der Pflanzen.

A15.03 Regal und Regalbretter begrenzen

Das Regal besitzt eine eigene Mindest- und Höchstzahl von Brettern, während jedes Brett eine getrennte Medienkapazität hat. Diese Grenzen werden auf der jeweils zuständigen Ebene geprüft. Ein Regal kann daher noch ein Brett aufnehmen, obwohl ein vorhandenes Brett bereits voll ist; beide Sachverhalte betreffen unterschiedliche Sammlungen.

A15.04 Bücher und CDs gemeinsam behandeln

Ein Medium-Interface stellt die für das Regal benötigten Angaben zu Breite und Gewicht bereit. Bücher und CDs dürfen diese Angaben intern unterschiedlich speichern oder berechnen. Für die Brettverwaltung genügt der gemeinsame Vertrag; sie summiert die Rückgaben, ohne alle konkreten Medienklassen kennen zu müssen.

Beispiel: eine Budgetprüfung formulieren

Ein rechteckiger Garten ist 6 Einheiten lang und 5 Einheiten breit; seine Modellfläche beträgt 30. Benötigen die vorhandenen Pflanzen zusammen 22, liefert `testeFlaeche` den Wert `true`. Bei einem Gesamtbedarf von genau 30 bleibt das Ergebnis `true`; bei 31 wird es `false`. Eine Pflanzenliste kann also gültige Anzahlsgrenzen einhalten und trotzdem die Flächenprüfung nicht bestehen. Daraus darf ohne entsprechende Vorgabe kein zusätzlicher Aufnahmevertrag erfunden werden. Bei großen int-Abmessungen wird die Multiplikation rechtzeitig auf `long` erweitert, damit kein überlaufener Wert verglichen wird.

```
Pflanze[] bestand;  
Medium[] medien;  
// Die Arrays speichern Referenzen auf konkrete Objekte.
```

Typischer Fehler: `testeFlaeche` ist eine Prüfung des Bestands. Daraus folgt nicht automatisch, dass schon die Hinzufügemethode jede Überschreitung der Fläche verhindern muss.

Verständnisfrage: Warum kann ein Brett Bücher und CDs in demselben Array verwalten?

Antwort: Beide implementieren den gemeinsamen Typ `Medium`. Über dessen Methoden kann das Brett die benötigte Breite und das Gewicht jedes Elements erfragen.

Übungsbezug: Klausuren 09, 10, 11, 12, 15, 16 · Teilaufgaben siehe Themenübersicht A15

A15 UML-Modelle: Garten, Regal und Gebäudeflächen

Kapazität prüfen und gezielt summieren

Lernziel: Du verbindest mehrere Aufnahmebedingungen und unterscheidest eine Gesamtsumme von der Summe bestimmter Untertypen.

A15.05 Das erste geeignete Brett finden

Für jedes Brett werden belegte Breite und Gesamtgewicht bestimmt. Ein Medium passt nur dann, wenn nach seiner Aufnahme sowohl Breitenbudget als auch Tragkraft eingehalten sind und ein Verwaltungsplatz frei ist. Die Bretter werden in der vorgegebenen Reihenfolge geprüft; beim ersten vollständigen Erfolg endet die Suche.

A15.06 Räume einem Gebäude zuordnen

Die abstrakte Klasse Raum fasst gemeinsame Eigenschaften von Wohnraum und Garage zusammen. Das Gebäude nimmt einen Raum nur auf, wenn seine Referenzsammlung noch Platz bietet und die Summe der Raumflächen innerhalb des Flächenbudgets bleibt. Bei einem abgewiesenen Raum dürfen weder Belegungszähler noch gespeicherte Referenzen geändert werden.

A15.07 Gesamtfläche und Wohnfläche

Die Fläche eines rechteckigen Raums ist Länge mal Breite und kann als konkrete gemeinsame Methode in der abstrakten Basisklasse stehen. Zur Gesamtfläche tragen alle aufgenommenen Räume bei; zur Wohnfläche nur Objekte des Typs Wohnraum. Die Auswahl kann mittels instanceof erfolgen, während die Zahlen- und Rückgabetypen weiterhin den UML-Vorgaben entsprechen.

A15.08 Gültige Grenzen begründen · Ergänzung

Nichtnegative Abmessungen verhindern, dass zusätzliche Objekte rechnerisch freien Platz erzeugen. Ein Vergleich mit $\leq$ lässt die vollständige Nutzung eines Budgets zu, ein Vergleich mit $<$ würde sie ausschließen. Welche Vorbedingungen geprüft werden müssen und welche als gegeben gelten, sollte im Methodenvertrag deutlich werden.

Beispiel: zwei Bretter vergleichen

Brett 0 hat noch 4 Einheiten Breite und 3 Einheiten Tragreserve. Brett 1 besitzt noch 7 Einheiten Breite und 9 Einheiten Tragreserve. Ein Medium mit Breite 3 und Gewicht 5 scheitert bei Brett 0 an der Tragkraft, obwohl seine Breite ausreicht; bei Brett 1 erfüllt es beide Bedingungen. Die Suche darf daher nach der gescheiterten Gewichtsprüfung nicht insgesamt abbrechen, sondern muss mit dem nächsten Brett fortfahren. Ein zusätzlich fehlender Verwaltungsplatz würde auch Brett 1 ungeeignet machen.

Typischer Fehler: Bei der Wohnfläche einfach alle Räume zu addieren würde auch Garagen mitzählen. Zuerst wird die passende Objektgruppe ausgewählt, dann deren Fläche summiert.

Verständnisfrage: Eine Wohnung enthält Wohnräume mit Flächen 12 und 18 sowie eine Garage mit Fläche 20. Wie lauten Gesamtfläche und Wohnfläche?

Antwort: Die Gesamtfläche beträgt 50 Flächeneinheiten. Zur Wohnfläche gehören nur die beiden Wohnräume, also 30 Flächeneinheiten.

A16 UML-Modell: Putzplan, Fenster und Reinigungskosten

Putzplan und polymorphe Kosten

Lernziel: Du leitest Pflichtbeziehungen aus dem Modell ab und lässt verschiedene Reinigungsarten ihre Kosten selbst berechnen.

A16.01 Gebäude, Fenster und Plan

Ein Gebäude ist im Modell genau einem Putzplan zugeordnet und besitzt eine begrenzte, nichtleere Sammlung von Fenstern. Ein gültiger Konstruktor muss diese Pflichtbeziehungen herstellen, beispielsweise mithilfe geeigneter Parameter. Ein nachträglich eingefügter null-Verweis würde nicht als vorhandener Plan oder vorhandenes Fenster zählen.

A16.02 Einträge im Plan verwalten

Der Putzplan speichert seine Reinigungseinträge in einer begrenzten Referenzsammlung. Beim Einfügen wird die Kapazität geprüft, beim indexbasierten Zugriff der gültige Bestand. Ein solcher Getter stellt vorhandene Einträge bereit, darf aber einen unbenutzten Platz nicht ohne passende Rückgabekonvention als regulären Reinigungseintrag ausgeben.

A16.03 Gemeinsame Angaben, verschiedene Berechnung

Ein abstrakter Reinigungseintrag hält gemeinsame Angaben wie Name und Firma und deklariert eine Kostenoperation. Fensterreinigung und Bodenreinigung konkretisieren diese Operation auf unterschiedliche Weise. Der Putzplan kann die Kosten anschließend über den gemeinsamen Typ summieren; die dynamische Bindung wählt pro Objekt die passende Methode.

A16.04 Flächenkosten und Festpreis

Bei einer Fensterreinigung werden die Flächen der zugeordneten Fenster addiert und mit einem Kostensatz verrechnet. Bei einer Bodenreinigung kann die Aufgabe dagegen einen konstanten Preis vorgeben. Der Rückgabewert hat in beiden Fällen dieselbe Bedeutung, obwohl die Berechnungswege verschieden sind; aus der gemeinsamen Kostenoperation folgt keine einheitliche interne Formel.

Beispiel: Gesamtkosten nachvollziehen

Eine Fensterreinigung umfasst zwei Fenster mit Flächen 2 und 3. Bei einem Kostensatz von 4 Geldeinheiten je Flächeneinheit entstehen $(2 + 3) \cdot 4 = 20$ Geldeinheiten. Kommt eine Bodenreinigung zum Festpreis 35 hinzu, meldet der Plan insgesamt 55. Für dieses vereinfachte Klausurmodell genügt die vorgegebene numerische Darstellung. Achte im Unterricht zunächst auf die Zuordnung der Größen: Die Fensterzahl wird nicht mit dem Kostensatz multipliziert, sondern die gesamte Fensterfläche.

```
Reinigungseintrag
|-- Fensterreinigung: Fläche mal Satz
+-- Bodenreinigung: vorgegebener Festpreis
```

Typischer Fehler: Ein abstrakter Eintrag wird nicht direkt mit new erzeugt. Er dient als gemeinsamer Referenztyp für konkrete Fenster- und Bodenreinigungen.

Verständnisfrage: Muss der Putzplan wissen, ob ein Eintrag seine Kosten aus einer Fläche oder aus einem Festpreis bestimmt?

Antwort: Für die bloße Kostensumme nicht. Er ruft die gemeinsame Kostenoperation auf und addiert ihre Ergebnisse.

A16 UML-Modell: Putzplan, Fenster und Reinigungskosten

Sind alle Fenster berücksichtigt?

Lernziel: Du übersetzt eine Aussage über alle Objekte in verschachtelte Suchen und verwendest den passenden Gleichheitsbegriff.

A16.05 Für jedes Fenster einen Treffer finden

Die Aussage verlangt: Zu jedem Fenster des Gebäudes existiert mindestens ein passender Fensterreinigungseintrag. Eine äußere Schleife betrachtet deshalb jedes Gebäudefenster, eine innere Suche die Reinigungseinträge und deren zugeordnete Fenster. Bleibt ein Fenster ohne Treffer, darf sofort false zurückgegeben werden; erst nach allen erfolgreichen Suchen ergibt sich true.

A16.06 Den Laufzeittyp beachten

Nur eine Fensterreinigung besitzt die für diese Suche benötigte Fensterzuordnung. Ein instanceof-Test filtert die passenden Einträge, bevor ein Cast den Zugriff auf deren spezielle Methoden ermöglicht. Bodenreinigungen werden bei der Abdeckungsprüfung übersprungen, können aber trotzdem zur Gesamtkostensumme beitragen.

A16.07 Identität statt gleicher Abmessungen · Ergänzung

Zwei Fenster können dieselbe Fläche haben und trotzdem verschiedene bauliche Objekte darstellen. Soll ein bestimmtes Gebäudefenster gereinigt werden, wird seine Referenzidentität mit == gesucht. Ein Vergleich allein über Länge, Breite oder berechnete Fläche könnte fälschlich ein anderes Fenster als passenden Treffer akzeptieren.

A16.08 Vorlage und Annahmen auseinanderhalten · Ergänzung

Java unterscheidet Groß- und Kleinschreibung in Methodennamen. Enthält die Vorlage dazu widersprüchliche Schreibweisen, muss die beabsichtigte einheitliche Fassung vor dem Übersetzen festgelegt und die Korrektur benannt werden. Zusätzliche Regeln, etwa zu Mehrfachzuordnungen, dürfen nicht allein aus einer Vermutung eingeführt werden; maßgeblich bleiben die beschriebenen Bedingungen.

Beispiel: die Quantoren prüfen

Das Gebäude besitzt die Fenster f1, f2 und f3. Reinigung R1 enthält f1 und f2, Reinigung R2 enthält nur f2. Obwohl jedes der beiden Reinigungsobjekte mindestens ein Gebäudefenster nennt, fehlt f3: Das Gebäude ist noch nicht vollständig abgedeckt. Nimmt R2 zusätzlich f3 auf, ist die Bedingung erfüllt. Dass f2 doppelt vorkommt, widerlegt die Aussage „mindestens einmal“ nicht. Eine Einschränkung auf genau eine Reinigung wäre eine andere, strengere Anforderung und müsste gesondert formuliert sein.

Typischer Fehler: Die vertauschte Prüfung „Jede Reinigung enthält irgendein Gebäudefenster“ beantwortet eine andere Frage und kann unversorgte Fenster übersehen.

Verständnisfrage: f1 und f2 sind zwei verschiedene Fenster gleicher Größe. Im Plan steht nur f1. Ist dadurch auch f2 abgedeckt?

Antwort: Nein. Gleiche Eigenschaften ersetzen keine Zuordnung desselben Objekts. f2 muss selbst in mindestens einer passenden Fensterreinigung referenziert sein.

A17 UML-Modell: Veröffentlichungen und Themengebiete

Konten, Rollen und Veröffentlichungen

Lernziel: Du unterscheidest Spezialisierung von Zusammenarbeit zwischen Objekten und vergibst dauerhafte Kennungen.

A17.01 Rollen durch Vererbung beschreiben

In der vorgegebenen Hierarchie erbt Editor von Reviewer und Reviewer von der abstrakten Basisklasse UserAcct. Ein Editor ist damit zugleich ein Reviewer und ein UserAcct und kann über Referenzen dieser Obertypen angesprochen werden. Die Beziehung zwischen einem Autor und einem Paper ist dagegen eine Assoziation: Ein Autor ist kein Paper, sondern steht mit Veröffentlichungen in Verbindung.

A17.02 Öffentliche Schnittstelle und Zustand

Nutzername und E-Mail-Adresse gehören zum Kontozustand und werden über die vorgesehenen Getter und Setter zugänglich gemacht. Welche Operation ein Ausdruck aufrufen darf, bestimmt zunächst sein deklarerter Referenztyp. Verlangt die Aufgabe für bestimmte Operationen ausdrücklich leere Methodenrumpfe, werden dort keine erfundenen Arbeitsabläufe ergänzt; eine Methode mit erforderlichem Rückgabewert braucht dennoch eine gültige Rückgabe.

A17.03 Mindestbeziehungen bei der Konstruktion

Für jede Assoziationsrichtung wird getrennt gelesen, wie viele Partner mindestens vorhanden sein müssen. Fordert ein Paper schon bei seiner Entstehung einen Autor, muss seine Konstruktion eine solche gültige Zuordnung herstellen. Daraus folgt nicht automatisch dieselbe Mindestzahl in der Gegenrichtung; die beiden Enden des Diagramms können unterschiedliche Bedingungen tragen.

A17.04 Ein gemeinsamer Kennungszähler

Ein static-Zähler in der gemeinsamen Kontobasisklasse liefert fortlaufende Kennungen auch für neu erzeugte Unterklassenobjekte. Die einmal vergebene Kennung wird zusätzlich in einem privaten final-Instanzfeld festgehalten. Eine später erhöhte Zählervariable ändert dadurch nicht die Identität bereits vorhandener Nutzer; die einfache Vergabe setzt die beschriebenen sequenziellen Aufrufe voraus.

Beispiel: drei verschiedene Rollen erzeugen

Nehmen wir für die Erklärung einen bei 1 startenden Zähler an. Zuerst entsteht ein konkreter Author mit Kennung 1, danach ein Reviewer mit Kennung 2 und schließlich ein Editor mit Kennung 3. Der gemeinsame nächste Zählerwert ist nun 4. Der Editor übernimmt seine Kontokennung beim Aufruf des Basisklassenkonstruktors; er soll nicht für jede Stufe der Vererbung erneut nummeriert werden. Seine Rolle und seine unveränderliche Kennung sind unterschiedliche Aspekte desselben Objekts.

```
UserAcct
+-- Reviewer
+-- Editor
```

Typischer Fehler: Wird nur ein gemeinsames static-Feld als Kennung angezeigt, sehen später alle Konten denselben aktuellen Zählerwert. Jede Instanz braucht ihre eigene gespeicherte Kennung.

Verständnisfrage: Kann eine Variable vom Typ Reviewer auf ein Editor-Objekt zeigen?

Antwort: Ja. Editor ist in dieser Hierarchie eine Unterklasse von Reviewer. Welche Methoden ohne Cast aufrufbar sind, richtet sich dennoch nach dem deklarierten Typ Reviewer.

A17 UML-Modell: Veröffentlichungen und Themengebiete

Zwei Richtungen konsistent halten

Lernziel: Du erkennst die doppelte Speicherung einer Beziehung und unterscheidest eine Selbstassoziation von einem Baum.

A17.05 Eine Beziehung an zwei Stellen speichern

Bei einer bidirektionalen Editor-Paper-Assoziation hält der Editor eine begrenzte Papersammlung und das Paper einen Herausgeberverweis. Beide Stellen beschreiben dieselbe fachliche Zuordnung. Eine erfolgreiche Aufnahme muss daher beide Seiten aktualisieren; ein belegter Rückverweis darf nicht im Widerspruch zur Sammlung des Editors stehen.

A17.06 Wechsel ohne Doppelzuordnung

Ein Paper mit vorhandenem anderem Herausgeber wird nicht einfach zusätzlich in eine zweite Sammlung eingetragen. Für einen Wechsel wird zunächst die alte Zuordnung kontrolliert aufgehoben und danach die neue hergestellt, sofern sie zulässig ist. Hilfsmethoden können die interne Gegenrichtung pflegen; sie dürfen dabei keine endlose gegenseitige Aufrufkette auslösen. Die geerbte Papersammlung für die Reviewer-Rolle und die eigene Sammlung für die Editor-Rolle bleiben getrennte Beziehungen.

A17.07 Themengebiete mit Unterthemen

Eine SubjectArea-Selbstassoziation verbindet Objekte derselben Klasse miteinander. Die Referenzsammlung der Unterthemen ist daher vom Typ SubjectArea[], ohne dass eine Klasse von sich selbst erbt. Ein solches Objekt kann als Oberthema in einer Beziehung und zugleich als Unterthema in einer anderen auftreten; Getter und Kapazitätsprüfung arbeiten wie bei anderen Sammlungen.

A17.08 Selbstassoziation ist nicht automatisch ein Baum · Ergänzung

Eine begrenzte Zahl ausgehender Unterthemen beschränkt nicht zwangsläufig die Zahl der Oberthemen eines Objekts. Ebenso folgt daraus allein keine Kreisfreiheit. Soll tatsächlich eine Baumstruktur entstehen, wären zusätzliche Bedingungen nötig; in einer Umsetzung der gegebenen Aufgabe dürfen solche Regeln nicht ohne entsprechende Vorgabe angenommen werden.

Beispiel: eine Zuordnung verändern

Editor e1 enthält Paper p, und p verweist auf e1. Nach erfolgreichem Entfernen muss p aus der Sammlung von e1 verschwunden sein und sein Herausgeberverweis den nun fehlenden Herausgeber ausdrücken. Erst bei der neuen Aufnahme durch e2 werden dessen Sammlung und der Rückverweis auf e2 gemeinsam gesetzt. Falls e2 keine Kapazität hat, darf keine falsche neue Zuordnung entstehen. Ob ein kompletter Wechsel atomar angeboten wird oder aus zwei Schritten besteht, ist eine Frage des vereinbarten Methodenvertrags.

Typischer Fehler: Nur paper.editor zu ändern lässt den alten Editor das Paper weiterhin führen. Die beiden Datenstellen dürfen nicht getrennte Wahrheiten entwickeln.

Verständnisfrage: Beweist die Angabe „höchstens fünf Unterthemen“, dass Themenzyklen unmöglich sind?

Antwort: Nein. Sie begrenzt nur die Anzahl der entsprechend zugeordneten Unterthemen. Ohne zusätzliche Zyklusbedingung kann auch eine zyklische Selbstassoziation entstehen.

A18 Polymorphe Anwendungsmodelle und Text-/Zahlenausgaben

Polymorphie an Textausgaben erklären

Lernziel: Du erkennst in einer gemeinsamen Textdarstellung die Aufrufe, deren Ergebnisse vom tatsächlichen Objekttyp abhängen.

A18.01 Objekte als Text darstellen

toString liefert eine Textdarstellung des aktuellen Objekts. Bei der Ausgabe eines Objekts oder bei seiner Verknüpfung mit einem String wird eine passende Textdarstellung verwendet. Ein Fahrzeug kann dabei auch seine referenzierte Person darstellen; die Beziehung wird über einen Verweis genutzt, nicht durch das erneute Erzeugen einer gleich aussehenden Person.

A18.02 Gemeinsame Tierausgabe

Eine gemeinsame Tiermethode kann einen allgemeinen Textteil mit dem Ergebnis einer überschreibbaren Lautmethode verbinden. Zeigt die Referenz auf einen Tiger oder eine Ziege, wählt dynamische Bindung den zugehörigen Laut. Eine Freund-Beziehung beschreibt dagegen den Verweis auf ein anderes Objekt und ist unabhängig davon, welche Lautmethode die Klasse implementiert.

A18.03 Berufe mit verschiedenen Regeln

Anwalt, Politiker und Manager sind im Aufgabenmodell konkrete Spezialisierungen von Person. Sie liefern Berufsbezeichnungen und Gehälter nach ihren jeweiligen Regeln; dabei können etwa Geschlecht, individuelle Werte oder eine fortlaufende Nummer eingehen. Das ist eine Modellvorgabe der Programmieraufgabe und keine allgemeine Aussage über reale Berufsgruppen.

A18.04 Eine unveränderte Basismethode nutzen

Ruft Person.toString intern getBeruf und getGehalt auf, werden diese Methoden auf dem tatsächlichen Objekt ausgeführt. Die Unterklassen können deshalb die Ausgabe beeinflussen, ohne toString selbst zu ändern. In einem Person-Array werden unterschiedliche konkrete Berufe gemeinsam gespeichert; dieselbe Ausgabeschleife liefert trotzdem typabhängige Ergebnisse.

Beispiel: ein Aufruf in zwei Stufen

Die Variable p hat den deklarierten Typ Person, referenziert aber ein Manager-Objekt. println(p) verwendet dessen geerbte Textdarstellung. Während deren Ausführung führen Aufrufe von getBeruf und getGehalt zu den im Manager wirksamen Implementierungen. Die Auswahl endet also nicht mit dem Betreten einer Methode der Oberklasse: Jeder darin ausgeführte überschreibbare Instanzmethodenaufruf wird erneut passend gebunden. Ein gezielter super-Aufruf und ein gewöhnlicher Aufruf sind dabei zu unterscheiden.

```
Person p = einManager;  
System.out.println(p);  
// toString -> getBeruf und getGehalt des Objekts
```

Typischer Fehler: Die Herkunft des Methodentextes bestimmt nicht automatisch alle späteren Aufrufe. Eine geerbte Methode arbeitet weiterhin mit dem tatsächlichen Empfängerobjekt.

Verständnisfrage: Muss jedes konkrete Berufsobjekt eine eigene Kopie der gesamten Ausgabemethode enthalten?

Antwort: Nein. Die gemeinsame Ausgabe kann in Person bleiben und die veränderlichen Teile über überschreibbare Methoden erfragen. Dadurch wird wiederkehrender Text nur einmal implementiert.

A18 Polymorphe Anwendungsmodelle und Text-/Zahlenausgaben

Fahrzeuge, Sonderwerte und Wurfmodell

Lernziel: Du verfolgst berechnete Ausgaben und trennst eine vorgegebene Modellkonvention von ihrer technischen Umsetzung.

A18.05 Fahrzeugspezifische Eigenschaften

EBus, Pkw und Lkw teilen gemeinsame Fahrzeugangaben, unterscheiden sich aber beispielsweise in Personenanzahl, Treibstoffart und Verbrauch. Überschreibbare Methoden stellen diese Unterschiede bereit, ohne jede Eigenschaft unnötig als weiteres Feld zu speichern. Eine begrenzte erlaubte Attributzahl fordert deshalb dazu auf, ableitbare Werte bei Bedarf zu berechnen. Bei v Litern pro 100 km gilt für $v > 0$: $100/v$ Kilometer pro Liter.

A18.06 Besondere Instanzen dauerhaft erkennen

Soll nur der erste erzeugte EBus eine Sonderrolle besitzen, wird diese Eigenschaft bei seiner Konstruktion festgehalten. Ein gemeinsamer Zähler allein würde später auch die Auskunft über ältere Objekte verändern. Für Lkw können individuelle Nummern ähnlich vergeben werden; zusätzliche Ausgabetexte lassen sich an `super.toString()` anfügen.

A18.07 Werte im Wurfmodell verfolgen

Im Wurfmodell wird eine vorgegebene Dauer begrenzt und daraus mit einer berechneten mittleren Geschwindigkeit eine Weite bestimmt. Eine Unterklasse kann die Geschwindigkeitsberechnung durch einen Widerstandsfaktor verändern. Für die Ausgabespur werden die effektive Dauer, der dynamisch gewählte Methodenaufruf und alle `double`-Zwischenergebnisse in ihrer tatsächlichen Reihenfolge ausgewertet.

A18.08 Division durch einen Nullverbrauch · Ergänzung

Eine positive endliche `double`-Zahl geteilt durch `+0.0` ergibt in Java positive Unendlichkeit, dargestellt als `Infinity`. Das unterscheidet sich von einer ganzzahligen Division durch null, die eine Ausnahme verursacht. Verlangt die EBus-Aufgabe stattdessen ausdrücklich `0.0` als Ausgabe für Kilometer pro Liter, muss die betreffende Methode diese Modellkonvention gezielt umsetzen.

A18.09 null in einer Ausgabe · Ergänzung

Wird eine `null`-Referenz mit einem String verknüpft, erscheint an dieser Stelle der Text `null`. Das bedeutet weder, dass eine reale Treibstoffart namens `null` existiert, noch dass ein direkter Methodenaufruf auf `null` erlaubt wäre. Die geforderte Ausgabeform und die inhaltliche Bedeutung der dargestellten Größe werden deshalb getrennt geprüft.

Beispiel: Sonderfall bewusst unterscheiden

Der Ausdruck `100.0 / 0.0` liefert `Infinity`, während `100 / 0` einen Laufzeitfehler auslöst. In beiden Fällen sehen die Operanden ähnlich aus, ihre Typen führen aber zu unterschiedlichen Regeln. Für die geforderte EBus-Ausgabe ist keines dieser automatischen Ergebnisse die ganze Lösung: Die Spezialisierung gibt nach Aufgabenvertrag gezielt `0.0` zurück. Ein Test sollte genau diesen besonderen Fall prüfen.

Typischer Fehler: Ein lokaler Parameter und ein gleichnamiges Objektfeld sind getrennte Speicherstellen. Das Verändern des Parameters im Wurfmodell verändert nicht automatisch das Feld.

Verständnisfrage: Warum ist ein unendlicher Quotient nicht automatisch die richtige EBus-Ausgabe?

Antwort: Weil der Methodenvertrag der Aufgabe eine besondere Darstellung verlangen kann. Die technische Gleitkommaregel allein entscheidet nicht über diese fachliche Ausgabekonvention.

B01 Algorithmusbegriff und Divide-and-Conquer

Vom einzelnen Ergebnis zum allgemeinen Verfahren

Du lernst, eine konkrete Rechnung von einem Algorithmus zu unterscheiden und eine Verfahrensbeschreibung auf Ausführbarkeit zu prüfen.

B01.01 Eine ganze Problemklasse lösen

Ein Algorithmus beschreibt nicht nur die Antwort für eine bestimmte Eingabe, sondern ein Verfahren für alle zulässigen Eingaben einer Problemklasse. Bei einer Maximumsuche können sich beispielsweise Länge und Inhalt der Zahlenfolge ändern. Die Beschreibung muss trotzdem festlegen, wie aus jeder erlaubten Folge ihr größter Wert gewonnen wird.

B01.02 Schritte präzise und endlich beschreiben

Eine Verfahrensbeschreibung besteht aus endlich vielen eindeutig formulierten Anweisungen, deren einzelne Schritte ausführbar sind. Schleifen erlauben dennoch eine von der Eingabe abhängige Anzahl von Ausführungen. Eine Anweisung wie „Wähle geschickt den besten Wert“ genügt nicht, solange „geschickt“ und „best“ nicht durch konkrete Operationen erklärt werden.

Beispiel: Das Maximum einer nichtleeren Folge

Die Aussage „In der Folge 4, 9, 2 ist das Maximum 9“ nennt nur ein Ergebnis. Ein allgemeines Verfahren beginnt dagegen mit dem ersten Element als bisherigem Maximum. Danach wird jedes weitere Element verglichen. Ist es größer, ersetzt es den bisherigen Kandidaten; andernfalls bleibt dieser unverändert. Nach dem letzten Element wird der Kandidat zurückgegeben. Für 4, 9, 2 lauten die Kandidaten nacheinander 4, 9 und 9. Für eine andere Eingabe wie -8, -3, -11 funktioniert dieselbe Beschreibung und liefert -3. Auch eine Folge mit nur einem Element ist erfasst: Der erste Kandidat ist bereits das Ergebnis. Die Voraussetzung „nichtleer“ gehört zur Problemdefinition, weil bei einer leeren Folge kein größtes enthaltenes Element existiert. Das Verfahren ist präzise, weil Vergleichen, Zuweisen und Weitergehen eindeutig festgelegt sind. Seine Beschreibung bleibt gleich lang, selbst wenn die Eingabe sehr viele Elemente enthält.

Typischer Fehler: Eine einzige korrekt ausgerechnete Beispielantwort belegt noch nicht, dass ein allgemeines und für alle zulässigen Fälle definiertes Verfahren vorliegt.

Verständnisfrage: Warum wäre „Beginne die Maximumsuche immer mit dem Kandidaten 0“ für beliebige ganzzahlige Folgen ungeeignet?

Antwort: Bei einer Folge ausschließlich negativer Zahlen würde 0 unverändert bleiben, obwohl dieser Wert gar nicht vorkommt. Die Initialisierung mit dem ersten vorhandenen Element vermeidet diesen Fehler.

B01 Algorithmusbegriff und Divide-and-Conquer

Terminierung und Zerlegen in Teilprobleme

Du prüfst, ob ein Verfahren endet, und erkennst die drei Arbeitsschritte von Divide-and-Conquer.

B01.03 Ein Ende für jede zulässige Eingabe

In der hier verwendeten Klausurdefinition muss ein Algorithmus für jede zulässige Eingabe nach endlich vielen Schritten ein Ergebnis liefern. Eine Schleife oder Rekursion braucht deshalb einen nachvollziehbaren Fortschritt zum Abbruch. Dass ein Verfahren bei einem einzigen Test endet, beweist seine Terminierung für andere Eingaben noch nicht.

B01.04 Beschreibung und Umsetzung unterscheiden

Ein Algorithmus kann in Alltagssprache, Pseudocode oder einer Programmiersprache beschrieben werden. Verschiedene Java-Programme können deshalb dasselbe Verfahren umsetzen. Korrektheit bedeutet, das verlangte Ergebnis zu liefern; sie fordert nicht, dass kein anderer Algorithmus schneller wäre oder weniger Speicher benötigte.

B01.05 Teilen, lösen und zusammensetzen

Divide-and-Conquer zerlegt eine Aufgabe in kleinere Aufgaben derselben Art. Ausreichend kleine Basisfälle werden unmittelbar gelöst; für die übrigen Fälle wird das Verfahren erneut angewandt. Anschließend werden die Teilergebnisse so zusammengesetzt, dass daraus eine Lösung des ursprünglichen Problems entsteht.

Beispiel: Vier Zahlen mit Mergesort ordnen

Die Folge 7, 2, 5, 1 wird zunächst in 7, 2 und 5, 1 geteilt. Jede Hälfte wird erneut geteilt, bis nur noch einzelne Zahlen vorliegen. Eine einzelne Zahl ist bereits sortiert und bildet den Basisfall. Beim Zusammensetzen entstehen aus 7 und 2 die Folge 2, 7 sowie aus 5 und 1 die Folge 1, 5. Zum abschließenden Mischen vergleicht man jeweils die ersten noch nicht übernommenen Zahlen beider Folgen. Dadurch entsteht 1, 2, 5, 7. Die Teilprobleme werden bei jedem Teilen kleiner; deshalb wird der Basisfall erreicht. Entscheidend ist auch das Mischen: Bloßes Aneinanderhängen von 2, 7 und 1, 5 würde keine insgesamt sortierte Folge ergeben. Die Zerlegung allein ist daher noch keine vollständige Problemlösung.

Typischer Fehler: Eine rekursive Methode terminiert nicht automatisch. Ein Aufruf mit unverändertem Argument erreicht einen kleineren Basisfall möglicherweise niemals.

Verständnisfrage: Welcher Bestandteil von Divide-and-Conquer fehlt, wenn sortierte Teilfolgen nur unverändert hintereinander geschrieben werden?

Antwort: Es fehlt ein geeignetes Zusammensetzen. Die Elemente der Teilfolgen müssen miteinander verglichen und zu einer insgesamt geordneten Folge gemischt werden.

B02 Asymptotische Laufzeiten und Aufwand von Operationen

Wachstumsraten und der richtige Maßstab

Du beschreibst Laufzeit als Funktion der Eingabegröße und trennst Wachstumsordnung von konkreten Eingaben.

B02.01 Häufige Laufzeitklassen

O-Notation beschreibt eine asymptotische obere Schranke des Aufwands. Typische Klassen sind $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ und $O(2^n)$. Sie geben an, wie sich der Aufwand bei wachsendem n grundsätzlich entwickelt, ohne eine bestimmte Ausführungszeit in Sekunden festzulegen.

B02.02 Dominierende Terme erkennen

Bei einer Summe bestimmt für hinreichend große Eingaben der am stärksten wachsende Term die enge Wachstumsklasse. Daher wächst $3n^2 + 5n + 2$ quadratisch. Konstante Faktoren bleiben für praktische Messungen relevant, verändern aber nicht die asymptotische Klasse.

B02.03 Behauptete Schranken begründen

Zum Prüfen einer Schranke kann man Ausdrücke ausmultiplizieren und durch ein konstantes Vielfaches abschätzen. Alternativ betrachtet man den Quotienten zur behaupteten Schranke. Wächst dieser unbeschränkt, kann die Schranke nicht gelten; dabei ist 2^n als Potenz sorgfältig von $2n$ als Produkt zu unterscheiden.

B02.04 Beste, mittlere und schlechteste Fälle

Verschiedene Eingaben gleicher Länge können unterschiedlich viel Arbeit verursachen. Der Best Case betrachtet den günstigsten, der Worst Case den ungünstigsten Fall. Ein Average Case benötigt zusätzlich Annahmen darüber, wie wahrscheinlich die einzelnen Eingaben oder Suchsituationen sind.

B02.05 n bezeichnet die Eingabegröße

Bei einer Suche in einem Array bezeichnet n normalerweise dessen Elementzahl. Die numerische Größe eines gespeicherten Schlüssels ist damit nicht gemeint. Ein Array mit zehn sehr großen Zahlen hat weiterhin nur zehn Elemente und wird nach diesem Maßstab nicht größer als ein anderes Array gleicher Länge.

Beispiel: Eine quadratische Kostenfunktion prüfen

Für $T(n) = 3n^2 + 5n + 2$ und $n \geq 1$ gilt $5n \leq 5n^2$ sowie $2 \leq 2n^2$. Daher ist $T(n) \leq 10n^2$ und somit $T(n) = O(n^2)$. Dagegen ist $T(n)/n = 3n + 5 + 2/n$ unbeschränkt; $O(n)$ wäre keine gültige obere Schranke. Die Begründung gilt für beliebig große n und beruht nicht nur auf wenigen eingesetzten Beispielwerten.

Typischer Fehler: Ein günstiger Testlauf ist kein Nachweis einer Worst-Case-Schranke. Ebenso beweisen kleine Messwerte keine langfristige Wachstumsordnung.

Verständnisfrage: Welche Größe wird bei einer binären Suche in einem Array mit hundert Elementen durch n beschrieben?

Antwort: n beschreibt die hundert gespeicherten Elemente. Ob ihre Schlüssel klein, groß, positiv oder negativ sind, ändert diese Eingabegröße nicht.

B02 Asymptotische Laufzeiten und Aufwand von Operationen

Gesamtkosten, O und Θ richtig lesen

Du kombinierst mehrere Arbeitsphasen und liest asymptotische Aussagen so genau, dass unpräzise Klausurformulierungen erkennbar werden.

B02.06 Mehrere Arbeitsphasen zusammenrechnen

Bei aufeinanderfolgenden Arbeitsschritten werden die Kosten addiert. Für q Suchanfragen nach einem einmaligen Sortieren ergibt sich beispielsweise Sortieraufwand plus q mal Suchaufwand. Der Aufwand einer Vorbereitung kann sich lohnen, wenn dieselbe vorbereitete Datenstruktur anschließend häufig verwendet wird.

B02.07 Kosten einer Teilroutine mitrechnen

Die Zahl der Methodenaufrufe allein bestimmt nicht immer die Laufzeit. Wird eine linear teure Teilroutine n -mal aufgerufen, kann quadratischer Gesamtaufwand entstehen. Bei Baumoperationen wird häufig zunächst in Abhängigkeit von der Höhe h gerechnet; erst danach verbindet man h mit der Knotenzahl n .

B02.08 Obere und enge Schranke unterscheiden · Ergänzung

$O(g(n))$ behauptet nur, dass ein Aufwand langfristig höchstens wie ein konstantes Vielfaches von g wächst. $\Theta(g(n))$ legt zusätzlich eine passende untere Schranke fest. Ein linear wachsender Aufwand ist deshalb sowohl $O(n)$ als auch $O(n^2)$, aber seine enge Klasse lautet $\Theta(n)$.

B02.09 Gleiche Klasse ist nicht gleiche Zeit · Ergänzung

Zwei Verfahren mit $\Theta(n \log n)$ können in realen Messungen unterschiedlich schnell sein. Die Notation blendet konstante Faktoren und kleinere Terme aus. Aus der gemeinsamen Klasse allein kann deshalb weder eine identische Laufzeit noch eine eindeutige Rangfolge für kleine Eingaben abgeleitet werden.

B02.10 Multiple Choice mathematisch einordnen · Ergänzung

Wenn eine Aufgabe genau zwei richtige O -Angaben erwartet, sind möglicherweise enge Wachstumsklassen gemeint. Bei wörtlicher Lesart können zusätzlich gröbere obere Schranken wahr sein. Eine saubere Antwort nennt die beabsichtigte Auswahl und erklärt diese Abweichung, statt mathematisch wahre Obergrenzen pauschal als falsch darzustellen.

Beispiel: Viele Anfragen an dieselben Daten

Ohne Vorbereitung benötigen q lineare Suchen insgesamt $O(qn)$. Sortiert man einmal mit $O(n \log n)$ und sucht danach binär, ergibt sich $O(n \log n + q \log n)$. Bei $q = n$ wird daraus $O(n \log n)$, während das erste Vorgehen $O(n^2)$ benötigt. Für nur eine Anfrage kann der Sortieraufwand dagegen unnötig sein. Diese Rechnung vergleicht Wachstum und ersetzt keine Messung der konkreten Implementierungen.

Typischer Fehler: Die Formel „Anzahl der Aufrufe mal n “ ist falsch, wenn die einzelnen Aufrufe mit unterschiedlich großen Argumenten arbeiten.

Verständnisfrage: Ist die Aussage „Ein Verfahren mit $\Theta(n)$ liegt in $O(n^3)$ “ mathematisch falsch?

Antwort: Nein. Eine kubische Funktion ist eine gültige, aber unnötig grobe obere Schranke. $\Theta(n^3)$ wäre für dieses Verfahren dagegen falsch.

B03 Schleifenanalyse, Rekursion und Rekursionsspeicher

Schleifen zählen statt Laufzeiten erraten

Du leitest exakte Aufrufzahlen aus Schleifengrenzen und Veränderungen der Zählvariablen ab.

B03.01 Zuerst die zu zählende Operation festlegen

Eine exakte Zählung beginnt mit der Frage, welche Operation gezählt wird: etwa jeder Aufruf von `tuwas`. Danach werden alle Stellen berücksichtigt, an denen diese Operation ausgeführt wird. Zunächst kann ein kleiner Eingabewert helfen; die allgemeine Formel muss jedoch aus dem Kontrollfluss folgen.

B03.02 Grenzen und zusätzliche Aufrufe beachten

Eine Schleife von 0 bis einschließlich n hat $n + 1$ Durchläufe, eine Schleife von eins bis ausschließlich n dagegen $n - 1$. Bei unabhängigen verschachtelten Schleifen multiplizieren sich die Durchlaufzahlen. Aufrufe vor dem Schleifennest oder einmal je äußerem Durchlauf müssen zusätzlich addiert werden.

B03.03 Dreieckige Schleifennester summieren

Hängt die innere Grenze von der äußeren Zählvariablen ab, darf man nicht einfach zwei gleiche Faktoren multiplizieren. Bei inneren Längen eins, zwei und so weiter bis n entsteht die Summe $1 + 2 + \dots + n$. Sie beträgt $n(n + 1)/2$ und wächst quadratisch.

B03.04 Multiplikative Schritte ergeben Logarithmen

Beim Verdoppeln wird nach k Schritten aus eins der Wert 2^k . Daher braucht man nur logarithmisch viele solche Schritte, um die Größenordnung n zu erreichen. Entsprechend führt wiederholtes Halbieren einer positiven Zweierpotenz nach $\log_2 n$ Halbierungen zum Wert eins; der mitgezählte Anfangszustand hängt von der Schleifenbedingung ab.

Beispiel: Eine wachsende innere Schleife

Im folgenden Ausschnitt läuft die äußere Schleife für $a = 1$ bis n . Für jedes a erfolgen genau a Aufrufe von `tuwas`. Bei $n = 4$ sind es daher zunächst einer, dann zwei, drei und vier, insgesamt zehn. Die Formel $n(n + 1)/2$ enthält alle Durchläufe; sie zählt nicht zusätzlich die Prüfungen der `for`-Bedingungen. Wird nach dem gesamten Schleifennest einmal `tuwas` aufgerufen, erhöht sich die Formel um eins, die quadratische Wachstumsklasse bleibt aber gleich.

```
for (int a = 1; a <= n; a++) {  
    for (int b = 0; b < a; b++) {  
        tuwas();  
    }  
}
```

Typischer Fehler: Ein Zählerstand nach der Schleife ist nicht automatisch die Zahl ihrer Durchläufe. Startwert, Schrittweite und inklusive Grenze müssen zusammen betrachtet werden.

Verständnisfrage: Wie viele Aufrufe entstehen bei $n = 5$ im gezeigten Schleifennest?

Antwort: Es entstehen $1 + 2 + 3 + 4 + 5 = 15$ Aufrufe. Da die innere Länge wächst, wäre die Rechnung $5 \text{ mal } 5$ zu groß.

B03 Schleifenanalyse, Rekursion und Rekursionsspeicher

Rekursionsbäume und Arbeit auf jeder Ebene

Du unterscheidest die Zahl rekursiver Aufrufe von der Arbeit, die innerhalb dieser Aufrufe ausgeführt wird.

B03.05 Aufrufe als Baum darstellen

Jeder Methodenaufruf wird als Knoten eines Rekursionsbaums dargestellt. Ein rekursiver Folgeaufruf erzeugt eine Kante zu einem Kind, zwei Folgeaufrufe erzeugen zwei Kinder. Die Beschriftung mit dem aktuellen Argument zeigt, wie schnell die Teilprobleme schrumpfen und wann der Basisfall erreicht wird.

B03.06 Den Ort der Arbeit beachten

Eine Hilfsoperation kann nur in den Blättern des Rekursionsbaums, nur in inneren Knoten oder in jedem Aufruf stattfinden. Diese Fälle haben verschiedene exakte Aufrufzahlen. Deshalb muss man den Basisfall und den Rekursionsschritt getrennt lesen, bevor man die Knoten des Baums zusammenzählt.

B03.07 Rekurrenz und Anfangswert zusammen angeben

Eine Rekurrenz drückt die gesuchte Anzahl durch kleinere Argumente aus. Bei zwei Folgeaufrufen kann beispielsweise $A(n) = 1 + 2A(n/2)$ gelten, ergänzt durch $A(1) = 0$. Die Ebenen enthalten geometrisch wachsende Knotenzahlen; ohne Anfangswert ist die Zählaufgabe noch nicht vollständig festgelegt.

B03.08 Aufwand aller Knoten einer Ebene addieren

Kosten die Aufrufe einer Ebene unterschiedlich viel, müssen ihre tatsächlichen Teilproblemgrößen berücksichtigt werden. Bei Halbierung gibt es auf Ebene j insgesamt 2^j Teilprobleme der Größe $n/2^j$. Ist die lokale Arbeit linear in dieser Größe, beträgt der gemeinsame Ebenenaufwand $O(n)$, nicht $O(n^2)$.

Beispiel: Zwei Halbierungen pro Aufruf

Für $n = 8$ und zwei Folgeaufrufe mit halbiertem Argument enthält der Baum zunächst einen Aufruf mit acht, dann zwei mit vier, vier mit zwei und schließlich acht Basisfälle mit eins. Eine Operation nur in inneren Knoten wird siebenmal ausgeführt; eine Operation nur im Basisfall dagegen achtmal. Werden alle Aufrufe gezählt, sind es fünfzehn. Kostet die Arbeit eines inneren Aufrufs genau so viele Schritte wie sein Argument, entstehen auf den drei inneren Ebenen jeweils acht Arbeitsschritte, zusammen vierundzwanzig. Die Verwaltung der Basisfälle kommt zum vollständigen Laufzeitmodell noch hinzu. Die unterschiedlichen Zahlen beantworten also unterschiedliche Fragen.

Typischer Fehler: Die Aufrufzahl darf nicht mit der Größe des ursprünglichen Arguments multipliziert werden, wenn die Teilroutine das jeweils verkleinerte Argument verarbeitet.

Verständnisfrage: Warum ist die maximale Tiefe dieses Rekursionsbaums kleiner als seine Gesamtzahl von Knoten?

Antwort: Die Tiefe betrachtet nur einen längsten Wurzel-Blatt-Pfad. Die Gesamtzahl enthält zusätzlich alle seitlichen Teilbäume, deren Aufrufe nacheinander bearbeitet werden.

B03 Schleifenanalyse, Rekursion und Rekursionsspeicher

Rekursionstiefe und Speicher verstehen

Du ermittelst den gleichzeitig benötigten Speicher und erkennst, wie eine iterative Umsetzung rekursive Verwaltungsarbeit ersetzt.

B03.09 Ein längster Aufrufpfad bestimmt die Tiefe

Während ein rekursiver Aufruf läuft, wartet sein aufrufender Vorgänger auf dessen Rückkehr. Die maximale Zahl solcher gleichzeitig aktiven Aufrufe liegt auf einem längsten Pfad zum Basisfall. Zwei nacheinander ausgeführte Kinder verdoppeln zwar möglicherweise die Gesamtarbeit, aber nicht die gleichzeitig benötigte Stapelhöhe.

B03.10 Rekursiv und iterativ sind Umsetzungsmöglichkeiten

Viele Verfahren lassen sich sowohl mit Selbstaufrufen als auch mit Schleifen formulieren. Fakultät und lineare Suche benötigen iterativ nur einen fortgeschriebenen Zustand. Bei Mergesort oder Baumtraversierungen muss eine iterative Fassung zusätzlich organisieren, welche Teilaufgaben noch ausstehen und in welcher Reihenfolge sie bearbeitet werden.

B03.11 Die Zählkonvention ausdrücklich nennen · Ergänzung

Manche Darstellungen zählen alle gleichzeitig aktiven Methodenaufrufe, andere nur die rekursiven Übergänge nach dem Erstaufruf. Beide Werte unterscheiden sich um eins. Wer die konkrete Aufrufkette notiert, macht eindeutig, welcher Basisfall enthalten ist und welche Zählweise der angegebenen Tiefe zugrunde liegt.

B03.12 Speicher aus aktiven Aufrufen ableiten · Ergänzung

Ein Aufrufrahmen speichert unter anderem Parameter, lokale Zustände und die Rückkehrstelle. Bei konstant großem Rahmen folgt der Zusatzspeicher der maximalen Tiefe. Eine Kette mit schrittweiser Verringerung benötigt daher linearen, Halbierungsrekursion logarithmischen und eine Baumrekursion im Allgemeinen $O(h)$ Speicher bei Baumhöhe h .

B03.13 Einen eigenen Stapel verwenden · Ergänzung

Eine iterative Traversierung kann die noch ausstehenden Knoten selbst auf einem Stapel speichern. Dadurch verschwinden die Informationen nicht; ihre Verwaltung wird nur vom Aufrufmechanismus in die Datenstruktur verlagert. Für Preorder werden beispielsweise Kinder in umgekehrter gewünschter Besuchsreihenfolge auf den Stapel gelegt.

Beispiel: Halbierungsrekursion ab acht

Der Pfad $8 \rightarrow 4 \rightarrow 2 \rightarrow 1$ enthält vier aktive Aufrufe, aber drei rekursive Übergänge. Selbst wenn jeder innere Aufruf zwei Kinder besitzt, bleibt diese maximale Zahl aktiver Aufrufe bestehen: Das erste Kind kehrt zurück, bevor das zweite ausgeführt wird. Für positive Zweierpotenzen erhält man entsprechend $\log_2 n + 1$ aktive Aufrufe. Das zählt den Basisfall mit.

Typischer Fehler: Rekursion durch einen eigenen Stapel zu ersetzen bedeutet nicht automatisch $O(1)$ Zusatzspeicher. Der explizite Stapel kann weiterhin mit der Problemgröße wachsen.

Verständnisfrage: Wie viele aktive Aufrufe und Übergänge hat der Pfad $3 \rightarrow 2 \rightarrow 1 \rightarrow 0$?

Antwort: Es sind vier aktive Aufrufe einschließlich des Basisfalls 0 und drei rekursive Übergänge zwischen ihnen.

B04 Java-Implementierung, Referenzen und Vergleiche

Datenstrukturen mit Klassen und Referenzen bauen

Du liest Java-Knotenklassen und entscheidest, ob eine Operation Dateninhalte oder die Identität eines bestimmten Knotens benötigt.

B04.01 Daten und Verknüpfungen modellieren

Ein Knotenobjekt speichert Nutzdaten und Verweise auf weitere Knoten. Der Konstruktor setzt die anfänglichen Feldwerte; die Datenstruktur selbst hält beispielsweise einen Kopfverweis. Bei einer leeren Liste ist dieser Kopf null, sodass noch kein erstes Knotenobjekt erreichbar ist.

B04.02 Schnittstelle und Hilfsmethode trennen

Eine öffentliche Methode bietet die gewünschte Operation nach außen an. Eine private Hilfsmethode kann zusätzliche Parameter wie den aktuellen Teilbaum erhalten und die eigentliche Rekursion durchführen. Die parameterlose Höhenmethode eines Baums kann deshalb einfach die Hilfsmethode mit der gespeicherten Wurzel aufrufen.

B04.03 Knoten durch Typparameter wiederverwenden

Ein generischer Knoten `Link<T>` legt den Typ seiner Nutzdaten zunächst als Typparameter fest. `Link<Person>` und `Link<Auftrag>` verwenden anschließend dieselbe Verknüpfungsstruktur mit unterschiedlichen Datentypen. Der Compiler kann dadurch unpassende Zuweisungen erkennen, ohne dass für jeden Datentyp eine eigene Knotenklasse notwendig ist.

B04.04 Vorbedingungen und Rückgaben planen

`assert` beschreibt eine vorausgesetzte Bedingung, etwa einen nichtleeren Knotenparameter; Assertions sind nicht standardmäßig immer aktiv. Ein früher `return` kann den leeren Fall gesondert behandeln. Rückgabesignale wie `null` oder `-1` müssen so vereinbart sein, dass sie vom normalen Ergebnis eindeutig unterschieden werden können.

B04.05 Identität oder Inhalt vergleichen

Mit `==` prüft man bei Referenzen, ob dasselbe Objekt gemeint ist. Eine passende `equals`-Methode kann stattdessen die fachlich relevanten Felder vergleichen. Beim Ausketten eines bereits gefundenen Knotens ist Identität gefragt; beim Vermeiden zweier Aufträge mit gleichen Daten dagegen inhaltliche Gleichheit.

Beispiel: Zwei gleiche Auftragsdaten

Zwei getrennt erzeugte Auftrag-Objekte können dieselbe Kennung und dieselbe Bezeichnung besitzen. Der Referenzvergleich liefert dann `false`, während die in der Aufgabe vorgegebene `equals`-Methode `true` liefern kann. Legt man beide Objekte in unterschiedliche `Link`-Knoten, sind außerdem die Knoten selbst verschieden. Eine Suche nach einem bestimmten Knoten darf deshalb nicht mit der Suche nach gleichen Auftragsdaten verwechselt werden. Im folgenden Schema bezeichnet `T` den jeweils eingesetzten Datentyp.

```
class Link<T> {  
    T daten;  
    Link<T> naechster;  
}
```

Typischer Fehler: Eine selbst geschriebene Gleichheitsmethode muss zu den fachlichen Anforderungen passen. Der bloße gleiche Typ zweier Objekte macht sie nicht inhaltlich gleich.

Verständnisfrage: Welcher Vergleich passt, wenn genau der zuvor von einer Suchmethode gelieferte Listenknoten entfernt werden soll?

Antwort: Der Referenzvergleich mit diesem Knoten ist passend. Er identifiziert genau das gefundene Objekt, auch wenn andere Knoten gleiche Nutzdaten enthalten.

Übungsbezug: Klausuren 38, 39, 40, 41, 42, 43, 44, 45 · Teilaufgaben siehe Themenübersicht B04

B04 Java-Implementierung, Referenzen und Vergleiche

Vergleiche und sichere Auswertung

Du unterscheidest Gleichheit von Sortiergleichheit und sicherst Java-Ausdrücke gegen ungültige Zugriffe und Überläufe.

B04.06 compareTo beschreibt eine Ordnung

Comparable vereinbart einen Vergleich mit negativem Ergebnis, 0 oder positivem Ergebnis. Diese Vorzeichen bedeuten kleiner, gemäß Ordnung gleichwertig oder größer. Der Betrag muss nicht dem Abstand zweier Werte entsprechen; entscheidend ist normalerweise nur das Vorzeichen.

B04.07 Sortiergleichheit ist ein eigener Begriff · Ergänzung

Ein Vergleich kann Personen ausschließlich nach dem Nachnamen ordnen. Zwei verschiedene Personen mit demselben Nachnamen sind dann für compareTo gleichwertig, obwohl ihre übrigen Daten verschieden sind. Daraus folgt weder zwingend equals-Gleichheit noch die Identität ihrer Objektreferenzen.

B04.08 Schutzbedingungen zuerst auswerten · Ergänzung

Bei && wird rechts nur bei wahren linken Operanden ausgewertet, bei || nur bei falschem. Deshalb prüft man zuerst, ob ein Knoten existiert oder ein Index gültig ist. Diese Reihenfolge verhindert den gefährlichen Zugriff vollständig.

B04.09 Hilfsmethoden und Zahlentypen passend wählen · Ergänzung

Math.min, Math.max und Math.abs vereinfachen Auswahl und Differenzprüfung; Math.pow liefert eine Gleitkommapotenz. Ein passender Ergebnistyp allein verhindert keinen Überlauf einer vorherigen int-Rechnung. Für große Produkte muss deshalb bereits mindestens ein Operand vor der Multiplikation nach long erweitert werden.

B04.10 Zugriff und Prüfaufwand richtig einordnen · Ergänzung

Private Zugriffsrechte beziehen sich auf die Klasse, nicht auf genau ein Objekt. Innerhalb derselben Klasse darf deshalb eine private Methode einer anderen Instanz aufgerufen werden. Aktivierte Assertions werden ausgewertet; eine vollständige Strukturprüfung darin verursacht zusätzlichen Aufwand.

B04.11 Fehler der Vorlage sichtbar behandeln · Ergänzung

Ein ungültiger for-Kopf verhindert schon die Übersetzung. Eine unpassende Zahl geforderter MC-Antworten kann dagegen auf einer fachlichen Mehrdeutigkeit beruhen. Benenne die notwendige Korrektur oder Konvention ausdrücklich, bevor du die festgelegte Aufgabe bearbeitest.

Beispiel: Vor dem Produkt erweitern

Für int-Werte $a = 50000$ und $b = 50000$ passt das Produkt nicht mehr in int. Die erste Zeile erweitert vor der Multiplikation. Die zweite rechnet zuerst als int; das anschließende Speichern in long repariert den Überlauf nicht.

```
long richtig = (long) a * b;  
long falsch = a * b;
```

Typischer Fehler: Der Vergleich durch $a - b$ kann selbst überlaufen und dadurch ein falsches Vorzeichen liefern.

Verständnisfrage: Warum schützt $p \neq \text{null} \ \&\& \ p.\text{daten} \neq \text{null}$ den Feldzugriff?

Antwort: Bei $p == \text{null}$ ist der linke Operand falsch. Wegen der Kurzschlussauswertung wird $p.\text{daten}$ dann überhaupt nicht gelesen.

B05 List, Set, Map und allgemeine Operationen verketteter Listen

Listentypen und die grundlegenden Verknüpfungen

Du unterscheidest abstrakte Sammlungen und lernst, wie wenige Referenzänderungen eine verkettete Liste verändern.

B05.01 Folge, Menge und Zuordnung unterscheiden

Eine List ordnet ihren Elementen Positionen zu und erlaubt gleiche Elemente mehrfach. Ein Set vermeidet Duplikate nach seinem Gleichheitsbegriff. Eine Map ordnet jedem vorhandenen Schlüssel einen Wert zu; verschiedene Schlüssel dürfen denselben Wert besitzen, ohne dass dadurch die Schlüsselzuordnung mehrdeutig wird.

B05.02 ArrayList und LinkedList vergleichen

Eine ArrayList speichert ihre Elemente in einem Feld und kann eine Position direkt ansprechen. Solange freie Kapazität besteht, ist auch Anhängen konstant teuer. Eine LinkedList muss für einen Indexzugriff Verknüpfungen durchlaufen; mit gespeichertem Endverweis kann sie trotzdem in konstanter Zeit hinten anhängen.

B05.03 Eine Kette mit Referenzen darstellen

Der Anfangsverweis zeigt auf den ersten Knoten, jeder Nachfolgerverweis auf den nächsten. Beim letzten erreichbaren Knoten ist der Nachfolger null. Ein zusätzlicher Endverweis kann denselben letzten Knoten direkt erreichbar machen; bei einer korrekt verwalteten leeren Liste existiert weder ein erster noch ein letzter Knoten.

B05.04 Verknüpfungen in richtiger Reihenfolge ändern

Beim Einfügen zwischen zwei Knoten muss der neue Knoten zuerst den bisherigen Nachfolger erhalten. Danach darf der Vorgänger auf ihn umgestellt werden. Beim Entfernen wird der Zielknoten übergangen; benötigte Daten werden vorher gesichert, und beim Kopf wird statt eines Vorgängers der Anfangsverweis geändert.

Beispiel: Zwischen A und B einfügen

Aus der Kette $A \rightarrow B \rightarrow C$ soll $A \rightarrow X \rightarrow B \rightarrow C$ werden. Zunächst erhält der neue Knoten X als Nachfolger den bisherigen Nachfolger von A, also B. Erst danach wird A.naechster auf X gesetzt. Die restliche Kette bleibt dadurch erreichbar. Würde man zuerst A.naechster überschreiben und den alten Wert nirgendwo sichern, könnte der Zugang zu B und C verloren gehen. Zum späteren Entfernen von X setzt man A.naechster auf X.naechster. Die Daten von X bleiben über eine zuvor gesicherte Referenz nutzbar, falls die Methode sie zurückgeben soll. An den Daten der anderen Knoten muss nichts geändert werden.

Typischer Fehler: Eine Änderung des Anfangsverweises verändert nicht das alte Knotenobjekt selbst. Sie verändert zunächst nur, von welchem Knoten aus die Liste erreichbar ist.

Verständnisfrage: Darf eine Map zwei unterschiedliche Schlüssel auf denselben Wert abbilden?

Antwort: Ja. Eindeutig sein muss die Zuordnung pro Schlüssel. Doppelte Werte sind erlaubt, während ein Schlüssel nicht gleichzeitig zwei verschiedene Zuordnungen in derselben Map besitzt.

B05 List, Set, Map und allgemeine Operationen verketteter Listen

Listen durchlaufen, verketten, teilen und kürzen

Du zerlegst größere Listenoperationen in einen Suchdurchlauf und wenige gezielte Änderungen von Nachfolerverweisen.

B05.05 Ende, Vorgänger und Länge ermitteln

Für die Länge wird ein Zähler bei jedem besuchten Knoten erhöht, bis null erreicht ist. Das letzte Element ist der Knoten ohne Nachfolger. Einen Vorgänger findet man dagegen, indem man prüft, ob sein Nachfolger der gesuchte Zielknoten ist; der Kopf besitzt innerhalb der Liste keinen Vorgänger.

B05.06 Ketten ohne Kopieren zusammenfügen

Beim Verketten wird der Nachfolger des letzten Knotens der ersten Liste auf den Kopf der zweiten gesetzt. Ist die erste Liste leer, übernimmt sie unmittelbar den zweiten Kopf. Anschließend wird die zweite Liste geleert, damit die Operation die vereinbarte Übertragung der Knoten ausdrückt und nicht zwei Listenverwalter dieselbe Kette behalten.

B05.07 Zwei getrennte Hälften herstellen

Bei n Knoten soll die vordere Hälfte aufgerundet $n/2$ Knoten erhalten. Man findet ihren letzten Knoten, merkt dessen bisherigen Nachfolger als Kopf der hinteren Hälfte und setzt dann die Verbindung auf null. Beide Teilketten müssen anschließend unabhängig erreichbar sein; bloßes Merken einer mittleren Referenz trennt noch nichts.

B05.08 Jedes zweite Element entfernen

Soll jeweils der Nachfolger eines erhaltenen Knotens gelöscht werden, zeigt dessen Verweis anschließend auf den übernächsten Knoten. Danach geht der Arbeitszeiger zum nächsten erhaltenen Knoten weiter. Ein einzelner letzter Knoten ohne Nachfolger bleibt unverändert, und bei einer leeren Liste darf kein Feld gelesen werden.

Beispiel: Eine Fünferliste teilen und kürzen

Die Liste $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ besitzt fünf Knoten. Ihre vordere Hälfte enthält daher A, B und C; die hintere beginnt bei D. Zuerst wird D als zweiter Kopf gespeichert, danach wird C.naechster auf null gesetzt. Das Ergebnis sind die getrennten Ketten $A \rightarrow B \rightarrow C$ und $D \rightarrow E$. Wird stattdessen aus der ursprünglichen Liste jedes zweite Element entfernt, übergeht A zunächst B und zeigt auf C. Anschließend übergeht C den Knoten D und zeigt auf E. Das Ergebnis ist $A \rightarrow C \rightarrow E$. Die beiden Operationen haben unterschiedliche Ziele und dürfen beim Nachführen des Arbeitszeigers nicht vermischt werden.

Typischer Fehler: Nach dem Ausketten eines Nachfolgers darf man nicht blind zwei weitere Verbindungen überspringen. Entscheidend ist, ob sich der Arbeitszeiger auf einem erhaltenen oder entfernten Knoten befindet.

Verständnisfrage: Welche Verbindung muss beim Teilen nach dem dritten Knoten einer Fünferliste getrennt werden?

Antwort: Der Nachfolerverweis des dritten Knotens muss null werden. Sein bisheriger Nachfolger muss zuvor als Kopf der zweiten Liste gesichert worden sein.

B05 List, Set, Map und allgemeine Operationen verketteter Listen

Umkehren, Randfälle und flache Kopien

Du erkennst den Unterschied zwischen einer neuen Knotenstruktur und neuen Datenobjekten und hältst Kopf- und Endverweise konsistent.

B05.09 Eine umgekehrte neue Liste aufbauen

Wer die ursprüngliche Liste von vorn nach hinten durchläuft und jedes gelesene Element am Kopf einer neuen Liste einfügt, kehrt die Reihenfolge um. Die neue Kette wächst damit in Gegenrichtung. Dabei können neue Knoten entstehen, während die ursprünglichen Knoten und ihre Nachfolgerverweise unverändert bleiben.

B05.10 Sonderfälle erhalten die Struktur

Eine leere Liste benötigt bei vielen Operationen eine eigene Behandlung, weil kein Knoten dereferenziert werden darf. Bei einem Einzelknoten fallen Anfang und Ende zusammen. Das Verketteten einer Liste mit sich selbst muss ausgeschlossen werden, sonst kann eine zyklische statt einer endlichen Kette entstehen.

B05.11 Endverweise korrekt nachführen

Mit Endverweis lässt sich der letzte Knoten unmittelbar erreichen, ohne die gesamte Liste zu durchlaufen. Dieser Vorteil gilt nur, wenn jede Änderung den Endverweis konsistent hält. Nach Entfernen des einzigen Knotens müssen beispielsweise sowohl Anfang als auch Ende den leeren Zustand anzeigen.

B05.12 Bibliotheksklasse und Eigenbau unterscheiden · Ergänzung

Die Java-LinkedList verwaltet einen Zugriff auf ihr Ende und kann hinten konstant schnell anhängen. Eine eigene einfach verkettete Liste mit ausschließlich einem Anfangsverweis besitzt diesen Vorteil nicht automatisch. Aussagen zur Laufzeit müssen deshalb zur tatsächlich gegebenen Repräsentation passen, nicht nur zum Wort „Liste“.

B05.13 Flach oder tief kopieren · Ergänzung

Bei einer flachen Kopie werden neue Listenknoten erzeugt, ihre Datenfelder dürfen aber dieselben Datenobjekte referenzieren. Eine Änderung der Kettenstruktur ist dann unabhängig, eine Änderung eines gemeinsam referenzierten veränderlichen Datenobjekts dagegen sichtbar. Eine tiefe Kopie müsste zusätzlich passende Kopien der Datenobjekte erzeugen.

Beispiel: Drei Knoten in Gegenrichtung

Nach dem Lesen von A enthält die neue Liste nur A. Nach B steht dort $B \rightarrow A$ und nach C schließlich $C \rightarrow B \rightarrow A$. Die ursprüngliche Folge $A \rightarrow B \rightarrow C$ bleibt erhalten. Speichern die Knoten Personenobjekte, darf dieselbe Person in beiden Strukturen referenziert sein. Das Entfernen ihres neuen Knotens aus der Kopie entfernt sie nicht aus der ursprünglichen Kette. Eine Namensänderung am gemeinsam verwendeten Personenobjekt betrifft jedoch beide Sichten.

Typischer Fehler: Neue Knoten bedeuten nicht automatisch neue Nutzdaten. Auch eine vollständig anders verknüpfte Liste kann dieselben veränderlichen Objekte wie die Ausgangsliste enthalten.

Verständnisfrage: Was muss beim Leeren einer Liste mit Anfangs- und Endverweis mit beiden Feldern geschehen?

Antwort: Beide müssen auf null gesetzt werden. Ein veralteter Endverweis würde weiterhin auf einen Knoten zeigen und den vereinbarten leeren Zustand verletzen.

B06 Prioritätslisten, Warteschlangen und Stapel

Prioritäten sortiert oder unsortiert verwalten

Du vergleichst zwei Prioritätslisten, bei denen sich die Arbeit zwischen Einfügen und Entfernen unterschiedlich verteilt.

B06.01 Die absteigende Ordnung bewahren

In einer sortierten Prioritätsliste müssen höhere Prioritäten vor niedrigeren stehen. Beim Einfügen wird deshalb die Stelle gesucht, an der die neue Priorität zwischen ihre Nachbarn passt. Für gleiche Prioritäten ist die in der Aufgabe festgelegte Regel maßgeblich; ohne weitere Vorgabe ist deren interne Reihenfolge beliebig.

B06.02 Das Maximum am Kopf entnehmen

Durch die absteigende Ordnung ist eine Aktion höchster Priorität direkt am Anfang erreichbar. Nach Sichern ihrer Daten wird der Kopf auf den Nachfolger gesetzt. Die Entfernung benötigt dadurch konstanten Aufwand, auch wenn die Liste sehr lang ist; die vorherige Suche nach einer Einfügestelle kann dagegen linear sein.

B06.03 Ohne Sortierung das Maximum suchen

Eine unsortierte Liste kann eine neue Aktion sofort vorne aufnehmen. Dafür muss die Entfernung höchster Priorität zunächst alle Knoten untersuchen. Wird der bisher beste Knoten nur bei echt größerer Priorität ersetzt, bleibt bei Gleichstand der erste maximale Knoten der aktuellen Listenreihenfolge erhalten.

B06.04 Den gefundenen Knoten ausketten

Liefert eine Hilfsmethode die Referenz auf das Maximum, ist zunächst zwischen Kopf und innerem Knoten zu unterscheiden. Beim inneren Knoten wird dessen Vorgänger gesucht und an ihm vorbei auf den nächsten Knoten verwiesen. Der zurückzugebende Aktionsinhalt kann anschließend aus dem gefundenen Zielknoten gelesen werden.

Beispiel: Gleiche Maximalprioritäten

Eine unsortierte Liste enthält A mit Priorität 3, danach B mit Priorität 8 und C ebenfalls mit Priorität 8. Die Suche beginnt mit A als bisherigem Kandidaten, ersetzt ihn durch B und behält B beim Vergleich mit C bei. Nach Entfernung von B verbleibt A → C. In einer absteigend sortierten Darstellung könnten dagegen B und C vor A stehen; das erste Maximum wäre bereits am Kopf. Wird eine Aktion D mit Priorität 5 ergänzt, muss die sortierte Liste D zwischen die Achtergruppe und A einfügen. Die unsortierte Variante darf D sofort vorn einsetzen, muss aber bei der nächsten Maximumsuche wieder vollständig durchlaufen werden.

Typischer Fehler: Der Vergleich $\geq$ würde bei einer Suche nach dem ersten Maximum den Kandidaten bei Gleichstand erneut ersetzen. Dadurch würde der letzte statt des ersten maximalen Knotens ausgewählt.

Verständnisfrage: Welche Variante ist beim reinen Einfügen ohne anschließende Suche besonders günstig?

Antwort: Die unsortierte Prioritätsliste kann jeden neuen Knoten unmittelbar vorn einfügen. Die sortierte Variante muss unter Umständen zunächst die passende Stelle finden.

B06 Prioritätslisten, Warteschlangen und Stapel

FIFO, LIFO und eindeutige Einträge

Du setzt Warteschlange und Stapel als Listen um und begründest die nötige Sucharbeit beim Vermeiden von Duplikaten.

B06.05 Warteschlange nach FIFO organisieren

FIFO bedeutet, dass der zuerst eingereihte Eintrag zuerst wieder entfernt wird. Eine Listenimplementierung fügt deshalb am Ende ein und nimmt am Anfang heraus. Nach dem Speichern der vorderen Daten setzt dequeue den Kopf weiter; bei einer leeren Warteschlange liefert die vereinbarte Methode null.

B06.06 Nur neue Personen hinten anstellen

Ohne Endverweis muss enqueue die Liste ohnehin bis zum letzten Knoten durchlaufen. Dabei kann gleichzeitig geprüft werden, ob eine inhaltlich gleiche Person schon enthalten ist. Bei einem equals-Treffer bleibt die Struktur unverändert; andernfalls wird der neue Knoten am ermittelten Ende angehängt.

B06.07 Stapel nach LIFO organisieren

LIFO entfernt zuerst den zuletzt oben abgelegten Eintrag. push und pop arbeiten daher grundsätzlich am Kopf. Soll push einen gleichen bereits gespeicherten Auftrag nach oben holen, wird dessen bisheriger Knoten vorher ausgekettet und der neue beziehungsweise übergebene Auftrag anschließend vorn abgelegt.

B06.08 Eindeutigkeit und Aufwand begründen

Beginnt der Stapel leer und entfernt jedes push einen vorhandenen gleichen Auftrag, bleibt höchstens ein solcher Auftrag enthalten. pop benötigt dann nur konstante Arbeit am Kopf. Die vorgeschaltete Duplikatsuche kann push trotzdem linear machen; ein schneller Einfügeschritt macht nicht automatisch die gesamte Operation konstant.

B06.09 Maximumsuche mit einer Invariante prüfen · [Ergänzung](#)

Nach jedem Suchschritt sollte der gemerkte Knoten das erste Maximum des bereits betrachteten Präfixes sein. Diese Aussage bleibt erhalten, wenn nur bei strikt größerer Priorität ersetzt wird. Ein Start ohne numerischen Scheinwert, beispielsweise mit null als „noch kein Kandidat“, funktioniert auch bei ausschließlich negativen Prioritäten.

Beispiel: Einen vorhandenen Auftrag nach oben holen

Der Stapel sei von oben nach unten $C \rightarrow B \rightarrow A$. Ein push für einen inhaltlich gleichen Auftrag A entfernt zunächst den bisherigen hinteren A-Knoten. Danach entsteht $A \rightarrow C \rightarrow B$. pop liefert nun A und lässt $C \rightarrow B$ zurück. Ein Vergleich bloßer Referenzen wäre ungeeignet, wenn der übergebene Auftrag ein neues Objekt mit denselben fachlichen Daten ist. Die vorgegebene equals-Methode entscheidet, welche Aufträge als gleich gelten.

Typischer Fehler: FIFO und LIFO beschreiben die Entnahmereihenfolge, nicht allein die verwendete Datenstruktur. Beide können durch unterschiedliche Operationen derselben Listenart umgesetzt werden.

Verständnisfrage: Warum darf die Maximumsuche in einer Liste mit Prioritäten -7 und -2 nicht mit dem Zahlenwert 0 als gültigem Maximum beginnen?

Antwort: Kein Listenelement übertrifft 0, sodass ein nicht vorhandener Wert erhalten bliebe. Der erste vorhandene Knoten oder ein ausdrücklich leerer Kandidatenzustand liefert eine korrekte Initialisierung.

B07 Binärbäume, Suchbäume und Traversierungen

Binärbäume vermessen und vollständig durchsuchen

Du beschreibst die Baumstruktur und leitest rekursive Berechnungen direkt aus den beiden Teilbäumen ab.

B07.01 Knotenrollen und Kindverweise unterscheiden

Die Wurzel besitzt keinen Elternknoten, jeder andere Knoten eines Baums genau einen. Ein Binärknoten hat höchstens ein linkes und ein rechtes Kind. Diese beiden Verweise können auch als Einträge eines Feldes gespeichert werden; ihre vereinbarte Bedeutung muss dann überall gleich verwendet werden.

B07.02 Die Höhenkonvention festlegen

Hier zählt die Höhe die Knotenebenen: Ein leerer Baum hat Höhe 0, ein Blatt Höhe eins. Für einen nichtleeren Baum wird eins zum Maximum seiner Teilbaumhöhen addiert. Bei einer Zählung in Kanten wären nichtleere Baumhöhen jeweils um eins kleiner; der Leerfall benötigt eine gesonderte Vereinbarung.

B07.03 Blätter und innere Knoten zählen

Ein Blatt besitzt kein Kind; ein innerer Knoten besitzt mindestens eines. Zum Zählen wird der aktuelle Knoten entsprechend seiner Eigenschaft bewertet und das Ergebnis der Teilbäume hinzugefügt. Ein leerer Teilbaum trägt 0 bei, sodass fehlende Kinder keine Sonderaddition verursachen.

B07.04 Maximale Größe und perfekte Form

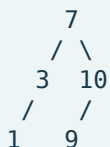
Auf jeder Ebene kann sich die Knotenzahl höchstens verdoppeln. Bei $h \geq 1$ Ebenen ergeben sich deshalb höchstens $2^h - 1$ Knoten und höchstens $2^{(h-1)}$ Blätter. Ein perfekter Binärbaum erreicht diese Grenzen, weil alle Ebenen gefüllt sind und alle Blätter dieselbe Tiefe besitzen.

B07.05 Extremwerte ohne Suchbaumannahme finden

In einem beliebigen Binärbaum sagt die Lage links oder rechts nichts über die Schlüsselgröße aus. Minimum und Maximum müssen daher den aktuellen Schlüssel und beide Teilbäume berücksichtigen. Bei einem einzigen Besuch jedes Knotens entsteht linearer Aufwand in der Knotenzahl.

Beispiel: Fünf Knoten auf drei Ebenen

Der skizzierte Baum besitzt Höhe drei. Seine Blätter sind die Knoten mit den Schlüsseln eins und neun; die übrigen drei Knoten sind innere Knoten. Perfekt ist er nicht, denn auf der letzten Ebene fehlen Positionen. Für das Minimum werden beide Teilbäume betrachtet und schließlich eins gewählt; das Maximum ist zehn. Die gleiche vollständige Suche wäre auch dann korrekt, wenn die Schlüssel anders auf die vorhandenen Positionen verteilt wären.



Typischer Fehler: Ein Knoten mit genau einem Kind ist kein Blatt. Außerdem ist ein leerer Teilbaum kein Knoten und darf nicht als zusätzliches Blatt mitgezählt werden.

Verständnisfrage: Welche Höhe hat ein Baum, der nur aus seiner Wurzel besteht, bei der hier verwendeten Konvention?

Antwort: Er hat Höhe eins. Seine beiden leeren Teilbäume haben jeweils Höhe 0; die Formel addiert eine Ebene für die Wurzel.

B07 Binärbäume, Suchbäume und Traversierungen

Suchbaumordnung gilt für ganze Teilbäume

Du prüfst die globale Ordnung und nutzt sie zum Suchen, ohne lokale Vergleiche mit einem vollständigen Nachweis zu verwechseln.

B07.06 Jede Abzweigung setzt eine Wertgrenze

Bei strenger Suchbaumordnung liegen links ausschließlich kleinere und rechts ausschließlich größere Schlüssel als an der jeweiligen Wurzel. Diese Regel betrifft alle Nachkommen im betreffenden Teilbaum. Entlang eines Pfads sammeln sich dadurch obere und untere Grenzen; eine beliebige pauschale Aussage über alle Vorfahrenschlüssel wäre zu grob.

B07.07 Die globale Ordnung rekursiv prüfen

Das Maximum des linken Teilbaums muss kleiner als der aktuelle Schlüssel sein, das Minimum rechts größer. Zusätzlich müssen beide Teilbäume selbst die Suchbaumeigenschaft besitzen. Ein Vergleich nur mit den beiden unmittelbaren Kindern übersieht mögliche Verletzungen durch weiter unten liegende Nachkommen.

B07.08 Extremwerte über Randpfade finden

In einem bereits gültigen Suchbaum kann das Minimum durch wiederholtes Folgen des linken Kindes gefunden werden. Das Maximum liegt entsprechend am Ende des rechten Randpfads. Der übrige Baum muss nicht untersucht werden, weil seine Schlüssel durch die Suchbaumordnung passend eingegrenzt sind.

B07.09 Die Höhe bestimmt den Suchaufwand

Suchen und gewöhnliches Einfügen folgen jeweils einem Weg von der Wurzel abwärts und benötigen daher $O(h)$ Zeit. Ohne Balancevorschrift kann eine ungünstige Einfügereihenfolge eine Kette erzeugen. Dann ist h linear in der Knotenzahl, sodass ein binärer Suchbaum nicht automatisch logarithmische Operationen garantiert.

B07.10 Eine entartete Listenform erkennen

Ein Baum bildet genau dann eine lineare Kette, wenn jeder seiner Knoten höchstens ein Kind hat. Zur iterativen Prüfung folgt man dem jeweils vorhandenen Kind und verwirft einen Knoten mit zwei Kindern. Links- und Rechtsabzweigungen dürfen dabei wechseln; eine Kette muss nicht durchgehend nach rechts verlaufen.

Beispiel: Lokal richtig, global falsch

Eine Wurzel mit Schlüssel zehn habe links den Knoten fünf und rechts den Knoten fünfzehn. Der Knoten fünf besitzt wiederum ein rechtes Kind mit Schlüssel zwölf. Jeder unmittelbare Eltern-Kind-Vergleich scheint richtig: fünf liegt links von zehn, zwölf rechts von fünf. Dennoch befindet sich zwölf im linken Teilbaum von zehn und ist dort unzulässig. Die Prüfung des linken Teilbaummaximums entdeckt den Fehler, weil dieses Maximum zwölf und damit größer als die Wurzel ist.

Typischer Fehler: Eine Sortierregel nur für Geschwister oder unmittelbare Kinder ist keine ausreichende Definition eines Suchbaums. Die Grenzen gelten jeweils für den gesamten Teilbaum.

Verständnisfrage: Ist eine Kette mit abwechselnd linken und rechten Kindverweisen automatisch kein Suchbaum?

Antwort: Nein. Die Kettenform erlaubt solche Wechsel. Ob zusätzlich ein Suchbaum vorliegt, hängt davon ab, ob alle Schlüssel die entlang des Pfads entstandenen Grenzen einhalten.

B07 Binärbäume, Suchbäume und Traversierungen

Traversieren und Baumkonventionen präzisieren

Du wählst Besuchsreihenfolgen bewusst und erkennst Zusatzkosten oder Mehrdeutigkeiten, die eine kurze Baumaufgabe verbergen kann.

B07.11 Drei Besuchsreihenfolgen unterscheiden

Preorder besucht die Wurzel vor ihren Teilbäumen, Inorder zwischen linkem und rechtem Teilbaum und Postorder nach beiden. Jede vollständige Traversierung besucht jeden Knoten einmal. Bei konstantem Aufwand pro Besuch sind deshalb alle drei in der Knotenzahl linear, trotz verschiedener Ausgabereihenfolge.

B07.12 Schlüssel absteigend sammeln

Im Suchbaum liefert die Reihenfolge rechts, Wurzel, links die Schlüssel absteigend. Teilfolgen werden nur dann durch Leerzeichen getrennt, wenn sie tatsächlich vorhanden sind. Der leere Baum liefert eine leere Zeichenkette und erzeugt weder ein zusätzliches Wort noch ein führendes Leerzeichen.

B07.13 Perfekt und linksbündig vollständig trennen · Ergänzung

Die Quellen verwenden „vollständig“ nicht überall eindeutig. Perfekt bedeutet vollständig gefüllte Ebenen; linksbündig vollständig erlaubt eine teilweise letzte Ebene. Bei Feldnummerierung mit Wurzelindex 0 und Kindern $2i+1$ sowie $2i+2$ belegt die zweite Variante lückenlos die ersten n Positionen.

B07.14 Gleiche Schlüssel bewusst zulassen · Ergänzung

Eine strenge Suchbaumdefinition verbietet gleiche Schlüssel. Andere Konventionen erlauben Gleichheit beispielsweise nur rechts; dann ändern sich die entsprechenden Grenzvergleiche. Die gewählte Regel muss für den gesamten Baum gelten und darf nicht an verschiedenen Knoten unterschiedlich angewandt werden.

B07.15 Wiederholte Teilbaumsuchen vermeiden · Ergänzung

Eine Prüfung über Minimum und Maximum kann dieselben Knoten mehrfach untersuchen. In einer langen Kette entstehen dadurch quadratische Gesamtkosten. Gibt man stattdessen zulässige Wertebereiche von der Wurzel zu den Kindern weiter, genügt ein einzelner Besuch jedes Knotens.

B07.16 Ausgabeaufwand gesondert betrachten · Ergänzung

Strings sind unveränderlich; eine Verkettung kann daher bereits aufgebaute Zeichen erneut kopieren. Eine lange Baumkette kann so quadratischen Kopieraufwand verursachen, obwohl die Traversierung selbst jeden Knoten nur einmal besucht. Knotenbesuche und Kosten des Ergebnisaufbaus sind deshalb getrennt zu analysieren.

Beispiel: Dieselben Knoten, andere Ausgabe

Eine Wurzel vier mit linkem Kind zwei und rechtem Kind sechs ergibt in Preorder 4, 2, 6, in Inorder 2, 4, 6 und in Postorder 2, 6, 4. Umgekehrte Inorder liefert 6, 4, 2. Entfernt man das rechte Kind, bleibt ein linksbündig vollständiger Baum zurück, aber kein perfekter Baum. Seine belegten Feldpositionen sind 0 und 1; die letzte Ebene ist nicht vollständig gefüllt.

Typischer Fehler: Die Zahl der Blätter allein entscheidet nicht über linksbündige Vollständigkeit. Für diese Eigenschaft ist auch wichtig, an welchen Positionen die Knoten liegen.

Verständnisfrage: Darf nach der Konvention „gleiche Schlüssel nur rechts“ ein gleicher Schlüssel im linken Teilbaum stehen?

Antwort: Nein. Links muss die Grenze weiterhin strikt eingehalten werden. Die erlaubte Gleichheit rechts lockert die linke Teilbaumregel nicht.

Übungsbezug: Klausuren 38, 39, 40, 41, 42, 43, 44, 45 · Teilaufgaben siehe Themenübersicht B07

B08 AVL-Bäume und Balancebedingungen

AVL-Balance an jedem Knoten prüfen

Du verbindest Suchbaumordnung mit Höhenbalance und erkennst, warum eine einzige ausgeglichene Wurzel noch keinen AVL-Baum garantiert.

B08.01 Zwei Eigenschaften gleichzeitig verlangen

Ein AVL-Baum ist ein binärer Suchbaum, bei dem sich die Höhen der beiden Teilbäume an jedem Knoten höchstens um eins unterscheiden. Suchbaumordnung und Höhenbedingung sind unabhängig zu prüfen. Ein geometrisch ausgeglichener Baum mit falsch platzierten Schlüsseln ist deshalb ebenso wenig ein AVL-Baum wie ein geordneter, stark schiefer Baum.

B08.02 Balancefaktoren konsistent definieren

Wir verwenden hier den Balancefaktor Höhe links minus Höhe rechts. Erlaubt sind damit -1, 0 und +1; Quellen mit umgekehrter Differenz tauschen nur die Vorzeichen. Der Wert gehört zu einem Knoten und seinen Kindern, nicht zu einem Elternknoten, den die Wurzel ohnehin nicht besitzt.

B08.03 Aus Balance folgt geringe Höhe

Die Höhenbedingung verhindert, dass ein großer Teilbaum dauerhaft neben einem sehr kleinen Teilbaum wächst. Für zunehmende Höhe muss ein AVL-Baum deshalb auch ausreichend viele Knoten enthalten. Seine Höhe wächst nur logarithmisch mit der Knotenzahl; das begrenzt die Länge der Pfade beim Suchen und Einfügen.

Beispiel: Ein weiterer Knoten verletzt die Wurzelbalance

Die Wurzel zehn habe links den Knoten fünf und rechts den Knoten zwanzig. Fünf besitzt die beiden Blätter drei und sieben; zwanzig ist selbst ein Blatt. Bei Höhenzählung in Ebenen hat die linke Seite der Wurzel Höhe zwei und die rechte Höhe eins. Ihr Balancefaktor ist also +1, und auch alle übrigen Knoten erfüllen die Bedingung. Fügt man nun unter drei links ein Blatt eins hinzu, erhöht sich die linke Höhe der Wurzel auf drei. Rechts bleibt die Höhe eins. Die Differenz beträgt nun zwei, sodass der Baum trotz weiterhin korrekter Schlüsselordnung kein AVL-Baum mehr ist. Die Verletzung muss nicht an dem neu eingefügten Blatt selbst auftreten; sie kann erst weiter oben sichtbar werden. Man untersucht daher den betroffenen Pfad bis zur Wurzel beziehungsweise bei einer vollständigen Prüfung alle Knoten.

Typischer Fehler: Gleiche Knotenzahlen links und rechts sind nicht die AVL-Bedingung. Gefordert ist eine beschränkte Höhendifferenz, und zwar an jedem Knoten.

Verständnisfrage: Ist ein Balancefaktor von -1 nach der hier gewählten Konvention erlaubt, und welche Seite ist höher?

Antwort: Ja. Die rechte Seite ist genau eine Ebene höher als die linke; diese Differenz liegt noch innerhalb der AVL-Bedingung.

B08 AVL-Bäume und Balancebedingungen

AVL-Prüfung mit Höhe und Fehlersignal

Du berechnest die Höhe nur einmal je Teilbaum und verstehst den Zusammenhang zwischen lokaler Balance, perfekter Form und Rotationen.

B08.04 Prüfung und Höhenberechnung verbinden

Eine rekursive Hilfsmethode kann bei Erfolg die Höhe und bei einem Fehler -1 liefern. Der leere Baum liefert 0; aus gültigen Teilbaumhöhen entsteht eins plus deren Maximum. Ein ungültiger Teilbaum oder eine zu große Höhendifferenz führt unmittelbar zum Fehlersignal, das bis zum ursprünglichen Aufruf weitergegeben wird.

B08.05 Überall Balance 0 ist besonders stark

Sind an jedem Knoten die beiden Teilbaumhöhen gleich, entsteht bei einem endlichen Binärbaum eine perfekte Form. Die Aussage muss tatsächlich für alle Knoten gelten, nicht nur für die Wurzel. Bei h Knotenebenen besitzt ein solcher Baum genau $2^h - 1$ Knoten, weil jede mögliche Position besetzt ist.

B08.06 Rotationen erhalten die Schlüsselreihenfolge · Ergänzung

Eine Rotation verändert einige Eltern-Kind-Verbindungen, während die sortierte Inorder-Reihenfolge erhalten bleibt. Dadurch kann ein nach dem Einfügen unbalancierter Suchbaum wieder eine passende Form erhalten. In den zugrunde liegenden Klausuraufgaben wird keine konkrete Rotationsfolge verlangt; hier dient das Prinzip zur Erklärung der logarithmischen AVL-Höhe.

Beispiel: Die Prüfung von unten nach oben

Der folgende Code setzt die Suchbaumeigenschaft voraus und prüft zusätzlich die AVL-Balance. Bei einem Blatt liefern die Aufrufe für beide leeren Teilbäume 0, also erhält das Blatt Höhe eins. Bei einer Kette aus drei nach links gerichteten Knoten bekommt die Wurzel links Höhe zwei und rechts Höhe 0; deshalb liefert sie -1. Ein größerer umgebender Baum übernimmt diesen Fehler, statt mit -1 als vermeintlicher regulärer Höhe weiterzurechnen. Jeder Knoten wird nur einmal besucht, sodass der Aufwand linear bleibt. Anschaulich zeigt die Schlüsselreihe 30, 20, 10 das Rotationsprinzip: Eine Rechtsrotation kann zwanzig zur Wurzel mit zehn links und dreißig rechts machen. Die Inorder-Folge bleibt 10, 20, 30, während die Höhe der Dreierkette sinkt.

```
int pruefe(Knoten k) {
    if (k == null) return 0;
    int l = pruefe(k.linkesKind);
    int r = pruefe(k.rechtesKind);
    if (l < 0 || r < 0 || Math.abs(l-r) > 1) return -1;
    return 1 + Math.max(l, r);
}
```

Typischer Fehler: Die Teilbaumhöhen sollten nicht für jeden Vergleich erneut berechnet werden. Das würde die gemeinsame Prüfung unnötig vervielfachen und kann den linearen Aufwand verlieren.

Verständnisfrage: Reicht es für einen perfekten Baum, dass nur die beiden Teilbäume seiner Wurzel gleich hoch sind?

Antwort: Nein. Diese Gleichheit muss rekursiv an jedem Knoten gelten. Zwei gleich hohe, jeweils unvollständig besetzte Teilbäume können eine ausgeglichene Wurzel besitzen, ohne perfekt zu sein.

B09 Binäre Heaps und Prüfung der Heapordnung

Heapform ist keine Suchbaumordnung

Du prüfst Form und Schlüsselregel getrennt und erkennst, welche Information ein Heap über seine Wurzel garantiert.

B09.01 Die Form eines binären Heaps

Ein binärer Heap ist bis auf möglicherweise die letzte Ebene vollständig besetzt. Auf dieser letzten Ebene stehen die Knoten ohne Lücken von links nach rechts. Diese Form erlaubt eine kompakte Darstellung in einem Array, entscheidet aber noch nicht, welche Schlüsselwerte an den einzelnen Positionen zulässig sind.

B09.02 Eltern und Kinder vergleichen

Beim Max-Heap ist jeder Elternschlüssel mindestens so groß wie die Schlüssel seiner vorhandenen Kinder. Beim Min-Heap gilt die umgekehrte Ungleichung. Geschwister sind untereinander nicht geordnet; gleiche Schlüssel sind erlaubt, solange die jeweilige Eltern-Kind-Bedingung erfüllt bleibt.

B09.03 Heap und Suchbaum nicht verwechseln

Ein Suchbaum unterscheidet kleinere Schlüssel links von größeren rechts und beschränkt damit ganze Teilbäume. Ein Max-Heap verlangt dagegen auf beiden Seiten nur kleinere oder gleiche Nachkommen. Daher kann ein gültiger Max-Heap rechts einen kleineren Schlüssel besitzen und gerade deshalb kein gültiger strenger Suchbaum sein.

Beispiel: Ein vollständiger Max-Heap

Die Skizze zeigt einen Max-Heap mit sieben Knoten. Die Wurzel neun ist mindestens so groß wie vier und sieben. Vier übertrifft seine Kinder eins und drei, sieben seine Kinder zwei und sechs. Alle Positionen der drei Ebenen sind besetzt, also stimmt auch die Form. Zwischen den Geschwistern vier und sieben muss keine aufsteigende oder absteigende Reihenfolge eingehalten werden. Für einen Suchbaum wäre diese Anordnung dagegen ungeeignet, weil rechts von neun ausschließlich größere Schlüssel stehen müssten. Ein Heap garantiert nicht, dass seine Schlüssel beim üblichen Inorder-Durchlauf sortiert erscheinen. Er garantiert hier vielmehr, dass an der Wurzel ein global größter Schlüssel steht: Entlang jedes Pfads kann der Schlüssel nach unten niemals größer werden. Dieselbe Begründung gilt für einen Min-Heap mit umgekehrten Vergleichen und global kleinstem Wurzelschlüssel.



Typischer Fehler: Ein symmetrisch oder ordentlich gezeichnetes Bild genügt nicht als Heapnachweis. Sowohl die Belegung der Positionen als auch alle vorgeschriebenen Vergleiche müssen stimmen.

Verständnisfrage: Dürfen beim Max-Heap die beiden Kinder derselben Wurzel gleiche Schlüssel haben?

Antwort: Ja. Die Kinder werden jeweils mit dem Elternschlüssel verglichen, nicht untereinander streng geordnet. Beide dürfen gleich groß sein, solange sie die Wurzel nicht übertreffen.

B09 Binäre Heaps und Prüfung der Heapordnung

Heapordnung rekursiv prüfen und Kosten trennen

Du leitest aus lokalen Vergleichen die Ordnung eines ganzen Teilbaums ab und hältst zusätzliche Formprüfungen davon getrennt.

B09.04 Das Maximum direkt ablesen

Ist bereits bekannt, dass ein nichtleerer Baum ein gültiger Max-Heap ist, steht ein größter Schlüssel an der Wurzel. Seine Rückgabe benötigt nur einen Feldzugriff. Eine Suche über den rechten Rand wäre die passende Idee für einen Suchbaum, ist für einen Heap aber unnötig und im Allgemeinen falsch.

B09.05 Lokale und rekursive Prüfung kombinieren

Die lokale Bedingung vergleicht den aktuellen Schlüssel mit jedem vorhandenen Kind. Für die Ordnung des gesamten Teilbaums müssen außerdem beide Kindteilmäue dieselbe Prüfung bestehen. Erst das Zusammenspiel aus lokaler Bedingung und rekursiven Ergebnissen deckt eine Verletzung beliebig weit unter der Wurzel ab.

B09.06 Fehlende und gleiche Kinder richtig behandeln

Ein fehlendes Kind verletzt keine Schlüsselungleichung, weil es keinen zu vergleichenden Schlüssel besitzt. Der leere Teilbaum erfüllt die rekursive Ordnungeigenschaft ebenfalls. Vor jedem Kindzugriff ist deshalb dessen Existenz zu prüfen; bei gleichen Schlüsseln muss der nichtstrenge Vergleich weiterhin Erfolg liefern.

B09.07 Formprüfung und Assertions gesondert rechnen · Ergänzung

Eine rekursive Ordnungsmethode kann einen lückenhaften Baum akzeptieren, obwohl dieser kein vollständiger Heap ist. Für den vollständigen Begriff braucht man zusätzlich die Formeigenschaft. Steht vor dem Wurzelzugriff eine aktivierte Assertion mit vollständiger Heapprüfung, verursacht diese lineare Arbeit, auch wenn der anschließende Zugriff konstant bleibt.

Beispiel: Gute Schlüsselordnung, falsche Form

Betrachte die Kette neun mit linkem Kind acht und dessen linkem Kind sieben. Jeder Elternschlüssel ist größer als sein Kind, sodass die reine Max-Heap-Ordnungsprüfung true liefern darf. Trotzdem fehlt bereits das rechte Kind der Wurzel, obwohl eine tiefere Ebene belegt ist. Bei Feldnummerierung liegen die Knoten an den Positionen 0, 1 und 3; Position 2 ist frei. Damit ist die Belegung nicht linksbündig vollständig. Eine Methode mit dem Namen istHeap muss entweder beide Eigenschaften prüfen oder durch ihre Vorbedingungen deutlich machen, dass die korrekte Form bereits vorausgesetzt wird.

Typischer Fehler: Der Name einer Methode beweist nicht, welche Eigenschaft ihr Code tatsächlich prüft. Entscheidend sind Bedingungen, rekursive Schritte und ausdrücklich gegebene Vorbedingungen.

Verständnisfrage: Welche beiden Ursachen können eine rekursive Heapordnungsprüfung an einem nichtleeren Knoten scheitern lassen?

Antwort: Entweder verletzt ein vorhandenes Kind den lokalen Schlüsselvergleich, oder einer der Kindteilmäue enthält weiter unten eine Ordnungsverletzung.

B10 B-Bäume: Suche, Ausgabe, Höhe und Ordnung

Mehrere Schlüssel steuern die B-Baum-Suche

Du liest einen B-Baum-Knoten als sortierte Folge von Grenzen und leitest daraus Suche und absteigende Ausgabe ab.

B10.01 Schlüssel begrenzen Kindintervalle

Ein B-Baum-Knoten enthält mehrere aufsteigend sortierte Schlüssel. Zwischen ihnen und außerhalb der beiden äußeren Schlüssel entstehen Intervalle für die Kindteilbäume. Bei zwei Schlüssel a und b gibt es daher einen Bereich kleiner a , einen zwischen a und b und einen größer b .

B10.02 Schlüssel- und Kindfelder zuordnen

Enthält ein Knoten r Schlüssel, besitzt seine Darstellung $r + 1$ Kindpositionen. Bei einem Blatt sind diese Kindverweise null; bei inneren Knoten verweisen sie auf die passenden Teilbäume. Unterschiedliche Knoten dürfen verschieden viele Schlüssel speichern, solange die zusätzlich festgelegten Belegungsgrenzen eingehalten werden.

B10.03 Das passende Intervall auswählen

Bei der Suche werden die Knotenschlüssel geprüft, bis ein gleicher Schlüssel oder die passende Einfügegrenze gefunden ist. Bei Gleichheit ist die Suche erfolgreich; andernfalls wird das dazugehörige Kind gewählt. Die iterative Form ersetzt den aktuellen Knoten, die rekursive Form ruft dieselbe Suche für diesen Kindteilbaum auf.

B10.04 Absteigend über Grenzen und Kinder laufen

Für eine absteigende Ausgabe beginnt man beim äußersten rechten Kind und arbeitet die Schlüssel von rechts nach links ab. Nach jedem Schlüssel folgt das unmittelbar links davon liegende Kindintervall. So werden die Schlüssel aller Teilbäume an der richtigen Stelle zwischen den Schlüssel ihres Elternknotens ausgegeben.

Beispiel: Suche und Ausgabe am Zweischlüsselknoten

Die Wurzel enthält zwanzig und fünfzig. Ihre drei Blattkinder enthalten links fünf und zehn, in der Mitte dreißig und vierzig sowie rechts sechzig und siebenzig. Eine Suche nach vierzig wählt das mittlere Intervall und findet dort den zweiten Schlüssel. Für fünfundvierzig wird dasselbe Blatt erreicht; hinter vierzig folgt jedoch nur ein null-Kind, sodass die Suche erfolglos endet. Die absteigende Ausgabe beginnt rechts mit siebenzig und sechzig, dann folgen fünfzig, vierzig, dreißig, zwanzig, zehn und fünf. Die beiden Wurzelschlüssel werden also zwischen die Ausgaben ihrer Kindintervalle eingeschoben, nicht gesammelt vor oder nach allen Kindern ausgegeben.

```

      [20 | 50]
     /   |   \
  [5 10] [30 40] [60 70]
```

Typischer Fehler: Ein Knoten mit zwei Schlüssel hat drei Kindintervalle. Wer nur zwei Kindpositionen verarbeitet, übersieht einen möglichen Suchbereich und einen Teil der Ausgabe.

Verständnisfrage: Welches Kindintervall wählt eine Suche nach einem Schlüssel, der kleiner als jeder Schlüssel des aktuellen Knotens ist?

Antwort: Sie wählt das äußerste linke Kind. Ist dieser Verweis null, endet die Suche dort ohne Treffer; andernfalls wird im gewählten Teilbaum weitergesucht.

B10 B-Bäume: Suche, Ausgabe, Höhe und Ordnung

B-Baum-Höhe und vorgegebene Ordnung prüfen

Du nutzt die Eigenschaften eines gültigen B-Baums gezielt und unterscheidest eine Belegungsprüfung von einer vollständigen Strukturvalidierung.

B10.05 Für die Höhe genügt ein Pfad

In einem gültigen B-Baum liegen alle Blätter auf derselben Ebene. Deshalb darf eine iterative Höhenmethode wiederholt ein fest gewähltes vorhandenes Kind verfolgen und die besuchten Ebenen zählen. Für den leeren Baum wird hier 0 zurückgegeben; ohne die vorausgesetzte gemeinsame Blattebene wäre ein einzelner Pfad kein zuverlässiger Höhenmaßstab.

B10.06 Die Quellenkonvention für Ordnung m

Für ein positives m bedeutet Ordnung m hier: Die nichtleere Wurzel enthält zwischen einem und 2m Schlüsseln, jeder andere Knoten zwischen m und 2m. Diese Definition ist nicht mit allen Lehrbuchkonventionen identisch. Die Prüfung muss deshalb zwischen Wurzel und gewöhnlichem Knoten unterscheiden.

B10.07 Leere Bäume und leere Texte zulassen

Der leere Baum erfüllt in der Aufgabe die Belegungsbedingung und liefert bei einer Ausgabe den leeren Text. Beim Zusammensetzen nichtleerer Teilfolgen dürfen Leerzeichen nur als tatsächliche Trennzeichen erscheinen. Ein leerer Teilbaum darf weder zusätzliche Schlüssel noch störende Randabstände erzeugen.

B10.08 Nur die verlangte Bedingung nachweisen · Ergänzung

Eine Methode zur Ordnung m kontrolliert die Schlüsselanzahlen jedes Knotens und ruft sich für die Kinder erneut auf. Sind Sortierung und gemeinsame Blattebene vorausgesetzt, muss sie diese nicht erneut beweisen. Die Obergrenze sollte mit einer ausreichend breiten Zwischenrechnung gebildet werden, damit $2m$ nicht vorher als int überläuft.

Beispiel: Ordnung zwei an einem kleinen Baum

Für $m = 2$ darf die Wurzel ein bis vier Schlüssel und jeder andere Knoten zwei bis vier Schlüssel enthalten. Eine Wurzel mit zwei Schlüsseln und drei Blattkindern mit jeweils zwei Schlüsseln erfüllt diese Belegungsbedingung. Wird aus einem Blatt ein Schlüssel entfernt, bleibt dort nur einer: Das Blatt verletzt nun die Untergrenze, obwohl ein Wurzelknoten mit einem Schlüssel erlaubt wäre. Die Kennzeichnung `istWurzel` verhindert diese Verwechslung. Der folgende Ausdruck zeigt nur die lokale Anzahlprüfung; für die vollständige Methode müssen anschließend alle Kindknoten mit `istWurzel` gleich `false` geprüft werden.

```
int minimum = istWurzel ? 1 : m;
int anzahl = k.schluessel.length;
boolean lokal = minimum <= anzahl && anzahl <= 2L * m;
```

Typischer Fehler: Aus gültigen Schlüsselanzahlen allein folgen weder korrekt sortierte Schlüssel noch gleiche Blatttiefen. Solche Eigenschaften müssen vorausgesetzt oder separat geprüft werden.

Verständnisfrage: Warum verwendet die Obergrenze im Beispiel $2L$ statt nur 2 ?

Antwort: Das L sorgt dafür, dass die Multiplikation bereits als `long` berechnet wird. Eine erst nachträgliche Umwandlung könnte einen zuvor entstandenen `int`-Überlauf nicht rückgängig machen.

B11 Lineare, binäre und Interpolationssuche

Suchbereiche systematisch verkleinern

Du lernst, lineare und binäre Suche auszuführen und ihre Vergleiche nachvollziehbar zu zählen.

B11.01 Lineare Suche

Bei der linearen Suche vergleichst du den gesuchten Schlüssel nacheinander mit den Arrayelementen. Das funktioniert auch ohne Sortierung und benötigt im ungünstigsten Fall n Vergleiche bei n Elementen. In einem aufsteigend sortierten Array darfst du erfolglos abbrechen, sobald ein größerer Wert erreicht ist.

B11.02 Warum vorher sortieren?

Die binäre Suche benutzt die Reihenfolge der Schlüssel, um nach einem Vergleich eine ganze Hälfte auszuschließen. Bei aufsteigender Sortierung liegen kleinere Werte links und größere rechts. Ohne diese Voraussetzung könnte der gesuchte Schlüssel gerade in der verworfenen Hälfte stehen; eine schnelle Suche wäre dann möglicherweise falsch.

B11.03 Grenzen und Mittelpunkt

Die Grenzen l und r gehören hier beide zum Suchbereich. Berechne den abgerundeten Mittelpunkt als $l + (r-l)/2$, vergleiche dort und setze anschließend entweder $l = m+1$ oder $r = m-1$. Dadurch wird auch das bereits geprüfte mittlere Element aus dem verbleibenden Bereich entfernt.

B11.04 Erfolg und Misserfolg protokollieren

Notiere vor jedem Vergleich das Intervall, den Mittelindex und den dort gespeicherten Wert. Gleichheit beendet die Suche erfolgreich. Wenn nach einer Aktualisierung l größer als r ist, enthält das Intervall keine Position mehr; dann ist der Schlüssel nicht vorhanden und es erfolgt kein weiterer Elementvergleich.

B11.05 Maximale Vergleichszahl

Für n mindestens eins benötigt die übliche binäre Suche höchstens $\text{floor}(\log_2 n) + 1$ besuchte Arraypositionen, wobei floor das Abrunden bedeutet. Welche Schlüssel diese Zahl erreichen, hängt von der Rundung des Mittelindex ab. Bei acht Elementen und Abrundung liegt die letzte Position besonders tief: Ihre Suche kann vier Vergleiche benötigen.

Beispiel: Erfolglose Suche nach 13

Im Array $[2,5,6,9,10,12,15,16,18]$ beginnt die Suche mit $[0,8]$. Mitte 4 enthält 10, deshalb bleibt $[5,8]$; Mitte 6 enthält 15, deshalb bleibt $[5,5]$. Dort steht 12. Danach gilt $l=6$ und $r=5$: Nach drei Vergleichen ist die Suche beendet. Nach dem Schlüssel 15 würden dieselben ersten beiden Vergleiche dagegen bereits zum Erfolg führen.

Typischer Fehler: Verändere bei einem Misserfolg die Grenze hinter den geprüften Mittelpunkt; sonst kann die Suche hängen bleiben.

Verständnisfrage: Ist für die Laufzeit die Zahl 18 oder die Anzahl neun maßgeblich?

Antwort: Maßgeblich sind die neun Arrayelemente. Der größte gespeicherte Zahlenwert bestimmt nicht die Zahl der Suchpositionen.

B11 Lineare, binäre und Interpolationssuche

Interpolationssuche und faire Vergleiche

Du nutzt Schlüsselwerte zur Positionsschätzung und unterscheidest garantierte Laufzeiten von verteilungsabhängigen Erwartungen.

B11.06 Position aus dem Wertebereich schätzen

Die Interpolationssuche schätzt, wie weit der Suchwert k zwischen den Randwerten liegt. Bei verschiedenen Randwerten lautet die Position $m = l + \text{floor}((k - a[l]) \cdot (r - l) / (a[r] - a[l]))$. Nach dem Vergleich wird die passende Intervallgrenze verschoben; gleiche Randwerte und Suchwerte außerhalb des Wertebereichs müssen zuvor gesondert behandelt werden.

B11.07 Laufzeiten richtig vergleichen

Lineare Suche benötigt bei gleich wahrscheinlichen Fundpositionen durchschnittlich $\Theta(n)$ Vergleiche und im schlimmsten Fall ebenfalls $\Theta(n)$. Binäre Suche garantiert $O(\log n)$. Interpolationssuche erreicht bei geeigneter Gleichverteilung im Mittel $O(\log \log n)$, kann bei ungünstigen Verteilungen aber bis zu $O(n)$ Vergleiche benötigen.

B11.08 Den Bruch nicht zu früh kürzen

Eine mathematische Division kann einen Bruch wie $5/16$ ergeben. Berechnest du diesen in Java zunächst mit int-Werten, entsteht bereits 0; anschließendes Multiplizieren rettet den verlorenen Bruchteil nicht. Verwende geeignete Gleitkommaarithmetik oder eine sicher dimensionierte Zwischenrechnung und runde erst die gesamte Positionsverschiebung ab.

B11.09 Rundung bestimmt die Ablaufspur · Ergänzung

Für binäre Suche sind Abrunden und Aufrunden des Mittelindex grundsätzlich möglich, sofern die Grenzen dazu passend aktualisiert werden. Bei acht Elementen verändert dies den Schlüssel mit maximaler Suchlänge. Deshalb gibst du bei einer Rechnung zuerst deine Rundung an; eine andere Spur ist nicht automatisch ein Fehler.

B11.10 Sortiert bedeutet nicht gleichverteilt · Ergänzung

Eine sortierte Folge kann große Lücken und dicht gedrängte Werte enthalten. Dann sagt der relative Zahlenabstand wenig über den tatsächlichen Index aus und die Interpolationsschätzung kann wiederholt schlecht liegen. Die günstige mittlere Laufzeit verlangt daher zusätzliche Verteilungsannahmen und folgt nicht allein aus der Sortierung.

Beispiel: Zwei Schätzungen finden die 7

Für $[2, 5, 6, 7, 10, 12, 15, 16, 18]$ gilt anfangs $l=0$ und $r=8$. Die Verschiebung ist $\text{floor}((7-2) \cdot 8 / (18-2)) = \text{floor}(40/16) = 2$. An Index 2 steht 6, also wird $l=3$. Die nächste Rechnung ergibt $3 + \text{floor}((7-7) \cdot 5 / (18-7)) = 3$. Dort steht der gesuchte Wert. Es waren zwei Elementvergleiche erforderlich.

Typischer Fehler: Gute Interpolationsleistung im Mittel ist keine Worst-Case-Garantie; prüfe außerdem einen möglichen Nenner null.

Verständnisfrage: Was ergibt in Java der int-Ausdruck $(5 / 16) * 8$?

Antwort: Er ergibt 0. Die Division wird zuerst ganzzahlig ausgeführt; mathematisch wäre das ungerundete Ergebnis 2,5.

B12 Hash-Tabellen und Kollisionsauflösung

Hashadressen und Sondierfolgen

Du unterscheidest die Startadresse eines Schlüssels von seinem tatsächlichen Platz und rechnest Kollisionen vollständig nach.

B12.01 Hashwert und Speicherposition

Eine Hashfunktion ordnet einem Schlüssel eine Tabellenadresse zu, beispielsweise $h(k)=k \bmod 7$ für nichtnegative Schlüssel. Diese Adresse ist der erste Versuch, nicht unbedingt der endgültige Speicherplatz. Wenn dort bereits ein anderer Eintrag liegt, entscheidet die Kollisionsbehandlung, an welcher weiteren Adresse gesucht wird.

B12.02 Kollisionen im offenen Adressieren

Eine Kollision liegt vor, wenn der für das Einfügen betrachtete Platz schon belegt ist. Beim offenen Adressieren liegen alle Einträge in derselben Tabelle; eine festgelegte Sondierfolge liefert weitere Kandidaten. Suche und Einfügen müssen dieselbe Folge benutzen, damit ein verdrängter Eintrag später wiedergefunden wird.

B12.03 Quadratisch sondieren

Eine mögliche quadratische Folge lautet $h_i(k)=(h(k)+i^2) \bmod M$ mit $i=0,1,2$ und so weiter. Jede Verschiebung wird von der ursprünglichen Hashadresse berechnet. Du addierst deshalb nicht einfach die nächste Quadratzahl zur zuletzt geprüften Position; das würde eine andere Folge erzeugen.

B12.04 Linear mit Schrittweite sondieren

Beim linearen Sondieren mit Schrittweite c gilt $h_i(k)=(h(k)+c \cdot i) \bmod M$. Der Modulo-Operator führt die Suche nach dem Tabellenende wieder an den Anfang zurück. Für $M=7$, Startadresse 5 und $c=3$ entstehen beispielsweise die Positionen 5, 1, 4, 0 und anschließend 3.

B12.05 Einfügereihenfolge beachten

Trage die Schlüssel genau in der angegebenen Reihenfolge ein und markiere nach jeder Einfügung den neuen belegten Platz. Jeder weitere Schlüssel sieht bereits den veränderten Tabellenzustand. Notiere auch erfolglose Versuche an belegten Positionen; nur die Endposition zeigt noch nicht, ob korrekt sondiert wurde.

Beispiel: Vollständiges lineares Einfügebeispiel

Verwende $M=7$, $h(k)=k \bmod 7$ und Schrittweite 3. Für die Folge 5,12,20,1,15 landet 5 an Platz 5. Die 12 prüft 5 und dann 1; die 20 belegt 6. Die 1 prüft 1 und dann 4. Die 15 prüft 1,4,0 und belegt 0. Von Index 0 bis 6 lautet die fertige Tabelle [15,12,leer,leer,1,5,20].

Typischer Fehler: Quadratische Verschiebungen beziehen sich hier auf die Startadresse; lineare Verschiebungen dürfen nicht mit ihnen vermischt werden.

Verständnisfrage: Wo würde die 8 anschließend in diesem Beispiel eingefügt?

Antwort: $h(8)=1$. Die Plätze 1,4,0 sind belegt; der nächste Kandidat 3 ist frei und nimmt die 8 auf.

B12 Hash-Tabellen und Kollisionsauflösung

Kollisionswahrscheinlichkeit und erreichbare Plätze

Du erklärst vorhandene Tabellenzustände und prüfst, ob eine Sondierfolge tatsächlich jeden freien Platz finden kann.

B12.06 Frühere Kollisionen erkennen

Liegt ein Schlüssel nicht an seiner ursprünglichen Hashadresse, musste er beim Einfügen verdrängt werden. Diese Schlussfolgerung setzt voraus, dass die Tabelle danach nicht durch Verschiebungen verändert wurde. Bei $h(k)=k \bmod 10$ beweist eine 16 an Position 7 daher, dass ihre Startposition 6 belegt war.

B12.07 Belegungsgrad als Wahrscheinlichkeit

Der Belegungsgrad α ist die Zahl belegter Plätze geteilt durch die Tabellenlänge. Sind die Startadressen gleichverteilt, entspricht α der Wahrscheinlichkeit, beim ersten Versuch einen belegten Platz zu treffen. Bei acht belegten von zehn Plätzen beträgt diese Wahrscheinlichkeit $8/10$, also 80 Prozent.

B12.08 Ein freier Platz und Suchkosten

Zyklisches lineares Sondieren mit Schrittweite eins besucht alle Tabellenplätze und findet deshalb auch den einzigen noch freien Platz. Bei guter Verteilung und kontrolliertem Füllgrad ist Hashsuche erwartet $O(1)$, im ungünstigsten Fall linear. Anders als binäre Suche oder Interpolationssuche benötigt sie keine sortierte Schlüsselreihenfolge.

B12.09 Konvention ausdrücklich festlegen · Ergänzung

Quadratisches Sondieren kann mit den Verschiebungen $0, +1, +4, +9$ oder mit $0, +1, -1, +4, -4$ definiert sein. Beide Regeln können verschiedene Endtabellen liefern; deshalb muss die verwendete Variante genannt werden. Der bloße Ausdruck Hash-Puffer aus einer Antwortoption beschreibt dagegen noch keine konkrete, ausführbare Kollisionsstrategie.

B12.10 Teilerfremde Schrittweiten · Ergänzung

Die Folge mit Schrittweite c erreicht alle M Plätze genau dann, wenn c und M teilerfremd sind. Bei $M=6$ und $c=2$ werden ab 0 nur 0,2,4 erreicht; freie ungerade Positionen bleiben unsichtbar. Dagegen erreicht Schrittweite 3 bei Tabellenlänge 7 sämtliche Positionen.

Beispiel: Einfügen bis zum zyklischen Anfang

In einer Zehnertabelle seien die Positionen 1,3,4,6,7,9 belegt; an Position 7 steht 16. Bei Schrittweite eins prüft der neue Schlüssel 26 die Plätze 6,7,8 und wird auf 8 gespeichert. Anschließend prüft der Schlüssel 8 die Plätze 8,9,0 und belegt 0. Nur 2 und 5 bleiben frei. Damit treffen acht der zehn gleich wahrscheinlichen Startadressen auf einen belegten Platz.

Typischer Fehler: Ein freier Platz irgendwo in der Tabelle garantiert bei ungeeigneter Schrittweite noch keinen erfolgreichen Fund.

Verständnisfrage: Wie viele verschiedene Plätze erreicht Schrittweite 2 in einer Tabelle der Länge 8?

Antwort: Vier Plätze. Ab Startadresse 1 sind das 1,3,5,7; anschließend wiederholt sich die Folge.

B13 Sortierv Verfahren und Vergleich ihrer Laufzeiten

Schnelle Sortierv Verfahren und ihre Garantien

Du verstehst die Grundideen von Merge-, Quick- und Heapsort und vergleichst ihre Laufzeiten unter passenden Voraussetzungen.

B13.01 Durchschnitt und schlimmster Fall

Die durchschnittliche Laufzeit beschreibt das Verhalten unter einem festgelegten Eingabemodell, während der Worst Case die ungünstigste Eingabe einer Größe betrachtet. Mergesort und Heapsort garantieren $O(n \log n)$. Quicksort erreicht diese Klasse im üblichen Durchschnittsmodell, kann jedoch auf $O(n^2)$ zurückfallen; gleiche Klassen bedeuten keine identischen Laufzeiten.

B13.02 Mergesort verbindet sortierte Hälften

Mergesort zerlegt die Folge, sortiert die Teilfolgen und verschmilzt sie durch Vergleich ihrer jeweils vordersten Elemente. Pro Ebene werden insgesamt linear viele Elemente bearbeitet, und es gibt logarithmisch viele Ebenen. Deshalb beträgt der Worst Case $\Theta(n \log n)$, sowohl bei rekursivem als auch bei Bottom-Up-Mergesort.

B13.03 Quicksort teilt an einem Pivot

Quicksort wählt ein Pivotelement und ordnet die übrigen Werte in passende Teilbereiche, die anschließend weiter sortiert werden. Ausgewogene Teilungen liefern insgesamt ungefähr $n \log n$ Arbeit. Wiederholt extrem ungleiche Teilungen können dagegen $n+(n-1)+\dots+1$ Arbeit verursachen; die Pivotwahl beeinflusst, welche Eingaben dies auslösen.

B13.04 Heapsort nutzt wiederholte Maxima

Heapsort baut für aufsteigende Ausgabe zunächst einen Max-Heap auf. Es tauscht das Maximum an das Ende des noch unsortierten Bereichs, verkleinert den Heap und stellt dessen Ordnung wieder her. Der Aufbau ist linear möglich, jede Entnahme kostet höchstens logarithmisch viel; iterative und rekursive Hilfsimplementierungen sind möglich.

B13.05 Bottom-Up-Heapsort einordnen

Bottom-Up-Heapsort ist eine Variante von Heapsort mit einer anders organisierten Wiederherstellung der Heapordnung. Sie kann weniger Schlüsselvergleiche benötigen, bleibt aber asymptotisch im Worst Case bei $O(n \log n)$. Gegenüber Quicksort besteht die wichtige Unterscheidung daher in der Worst-Case-Garantie, nicht in einer neuen logarithmischen Wachstumsklasse.

Beispiel: Ein vollständiger Merge

Die sortierten Folgen [2,7] und [3,5] werden verschmolzen. Vergleiche 2 mit 3 und übernahm 2; dann 7 mit 3 und übernahm 3; anschließend 7 mit 5 und übernahm 5. Die rechte Folge ist leer, also wird die verbliebene 7 angehängt. Das Ergebnis [2,3,5,7] erforderte drei Schlüsselvergleiche. Die letzte Übernahme benötigte keinen weiteren Vergleich zwischen zwei Schlüsseln.

Typischer Fehler: Eine rekursive Formulierung ist keine Voraussetzung für Mergesort und keine Garantie für eine bestimmte Laufzeit.

Verständnisfrage: Welches der genannten Verfahren besitzt einen quadratischen Worst Case?

Antwort: Quicksort besitzt einen solchen Worst Case. Mergesort und Heapsort bleiben bei einer asymptotischen Obergrenze von $O(n \log n)$.

B13 Sortierv Verfahren und Vergleich ihrer Laufzeiten

Einfache Sortierung und Kosten vieler Suchanfragen

Du unterscheidest Schlüsselvergleiche von Kontrolltests und erkennst, wann Vorsortieren zusätzlichen Aufwand rechtfertigt.

B13.06 Bubble- und Insertionsort

Bubblesort vertauscht benachbarte falsch geordnete Elemente, während Insertionsort jedes neue Element in ein bereits sortiertes Präfix einfügt. Beide benötigen im üblichen Durchschnitt und im Worst Case $\Theta(n^2)$ Arbeit. Ein Bubblesort mit Abbruch nach einem tauschfreien Durchlauf hat auf sortierten Daten einen linearen Best Case; ohne diese Optimierung nicht.

B13.07 Insertionsort auf sortierten Daten

Bei direktem Insertionsort wird ab dem zweiten Element jeweils der passende Platz im bisherigen Präfix gesucht. Ist das Array schon sortiert, genügt ein Schlüsselvergleich mit dem unmittelbaren Vorgänger und nichts wird verschoben. Für n Elemente entstehen deshalb genau $n-1$ solche Vergleiche und insgesamt lineare Arbeit.

B13.08 Selectionsort zählt unabhängig von der Ordnung

Selectionsort sucht im unsortierten Rest das kleinste Element und bringt es an dessen Anfang. Die Restlängen sinken jeweils um eins; die Vergleichszahl ist $(n-1)+(n-2)+\dots+1 = n(n-1)/2$. Daher bleibt auch eine bereits sortierte Eingabe quadratisch, obwohl möglicherweise keine wirksamen Vertauschungen nötig sind.

B13.09 Vergleiche und Suchvorbereitung

Ein Schlüsselvergleich entscheidet über zwei Datenwerte; Schleifengrenzen und Indexprüfungen sind andere Operationen. Für viele Suchen kann sich einmaliges Sortieren lohnen: $O(n \log n)$ Sortierung plus n binäre Suchen bleibt $O(n \log n)$. Dagegen benötigen n lineare Suchen im Worst Case zusammen $O(n^2)$.

B13.10 Mergesort ohne rekursive Aufrufe · Ergänzung

Bottom-Up-Mergesort beginnt mit einzelnen Elementen, die bereits sortierte Läufe der Länge eins sind. Es vereinigt benachbarte Läufe zu Längen zwei, vier, acht und so weiter, bis die ganze Folge ein Lauf ist. Diese Schleifenorganisation realisiert dasselbe Verschmelzprinzip ohne rekursive Aufrufe und bleibt $\Theta(n \log n)$.

Beispiel: Vier sortierte Schlüssel

Betrachte $[1,2,3,4]$. Insertionsort prüft die Paare $1/2$, $2/3$ und $3/4$: drei Schlüsselvergleiche. Optimiertes Bubblesort prüft in seinem ersten Durchlauf ebenfalls drei Nachbarpaare und stoppt ohne Tausch. Selectionsort untersucht dagegen zuerst drei, dann zwei, dann einen weiteren Kandidaten: insgesamt sechs Vergleiche. Eine günstige Eingabe hilft also nicht jedem Verfahren in gleichem Maß.

Typischer Fehler: Behaupte den linearen Bubblesort-Best-Case nur dann, wenn das beschriebene Verfahren einen vorzeitigen Abbruch besitzt.

Verständnisfrage: Wie viele Schlüsselvergleiche braucht direkter Insertionsort für acht bereits sortierte Elemente?

Antwort: Sieben. Ab Index eins wird jedes Element genau einmal mit seinem Vorgänger verglichen; es findet keine Verschiebung statt.

B14 Cocktail-Sort und Gnome-Sort schrittweise ausführen

Cocktail-Sort und Gnome-Sort verstehen

Du verfolgst Nachbartausche und hältst Datenzustand, Suchrichtung und aktive Grenzen getrennt fest.

B14.01 Cocktail-Sort läuft in zwei Richtungen

Cocktail-Sort vergleicht benachbarte Elemente zunächst von links nach rechts und anschließend von rechts nach links. Im Vorwärtslauf wandert ein größtes Element an den rechten Rand, im Rückwärtslauf ein kleinstes an den linken. Durch diese abwechselnden Richtungen können weit falsch liegende kleine Werte schneller nach links gelangen.

B14.02 Grenzen und Tauschkennzeichen

Nach einem vollständigen Vorwärtslauf kann der rechte fertige Rand aus dem aktiven Bereich ausgeschlossen werden; nach dem Rückwärtslauf entsprechend der linke. Ein Tauschkennzeichen merkt, ob sich während eines Durchlaufs etwas verändert hat. Bleibt es unverändert falsch, ist der geprüfte Bereich geordnet und die optimierte Variante darf beenden.

B14.03 Eine lückenlose Ablaufspur

Schreibe nach jeder geforderten Schleifeniteration den Feldzustand auf, auch wenn kein Tausch stattgefunden hat. Führe außerdem start, ende und die Laufrichtung mit, damit die nächste Vergleichsposition eindeutig ist. Eine Schleife ohne Durchlauf verändert keine Daten, kann aber von einer im Code folgenden Grenzänderung begleitet sein.

B14.04 Gnome-Sort geht bei Fehlern zurück

Gnome-Sort prüft ein Element zusammen mit seinem Vorgänger. Ist das Paar richtig geordnet, geht der Index nach rechts; andernfalls werden beide Werte getauscht und der Index geht nach links. Am Feldanfang muss wieder nach rechts gewechselt werden, da dort kein Vorgänger existiert.

Beispiel: Cocktail-Sort für [3,2,1]

Verwende zunächst start=0 und ende=2. Der erste Vorwärtsvergleich tauscht 3 und 2: [2,3,1]. Der nächste tauscht 3 und 1: [2,1,3]. Die 3 steht nun endgültig rechts, deshalb sinkt ende auf 1. Der Rückwärtsvergleich tauscht 2 und 1: [1,2,3]. Danach steigt start auf 1; der aktive Bereich enthält höchstens ein Element und die Sortierung ist abgeschlossen. Insgesamt waren es drei Schlüsselvergleiche und drei Nachbartausche. Für eine schriftliche Spur genügt es nicht, nur die abschließende sortierte Folge aufzuschreiben: Die beiden Zwischenfelder zeigen, dass tatsächlich in der geforderten Reihenfolge verglichen wurde.

Typischer Fehler: Vermische nicht die Grenzen verschiedener Cocktail-Sort-Varianten; die konkrete Schleifenbedingung bestimmt, welches Paar noch geprüft wird.

Verständnisfrage: Warum darf die rechts liegende 3 nach dem ersten Vorwärtslauf ausgeschlossen werden?

Antwort: Sie ist das Maximum des zuvor aktiven Bereichs. Bei jedem Nachbarvergleich wanderte das größere Element nach rechts; am Ende steht deshalb das Maximum ganz rechts.

B14 Cocktail-Sort und Gnome-Sort schrittweise ausführen

Sortierspuren, Tauschoperationen und Inversionen

Du zählst nur die verlangten Vergleiche und begründest, warum Nachbartauschverfahren im schlimmsten Fall quadratisch werden.

B14.05 Indexzustand bei Gnome-Sort

Für eine vollständige Spur notierst du nach jeder while-Iteration den aktuellen Index und das gesamte Array. Ein Rückwärtsschritt kann einen bereits betrachteten Bereich erneut erreichen; das ist beabsichtigt. Vor dem Zugriff auf den Vorgänger muss der Feldanfang gesondert behandelt werden, und bei Index n endet die Schleife.

B14.06 Tauschen braucht einen Zwischenspeicher

Beim Vertauschen zweier Arraywerte muss einer zunächst in einer Hilfsvariablen gesichert werden. Danach kann der zweite Wert den ersten Platz übernehmen und der gesicherte Wert den zweiten. Zwei direkte Zuweisungen ohne Sicherung würden einen ursprünglichen Wert überschreiben, sodass am Ende beide Plätze denselben Inhalt hätten.

B14.07 Best Case und Worst Case

Gnome-Sort benötigt auf bereits sortierten Daten $n-1$ Schlüsselvergleiche; optimiertes Cocktail-Sort stoppt ebenfalls nach einem tauschfreien Durchlauf. Auf stark umgekehrten Folgen sind dagegen quadratisch viele Nachbartausche und Vergleiche nötig. Die genaue Vergleichszahl hängt von der Schleifenvariante ab, die asymptotische Worst-Case-Klasse bleibt für beide $\Theta(n^2)$.

B14.08 Inversionen zählen Fehlordnungen · Ergänzung

Eine Inversion ist ein Positionspaar $i < j$ mit $a[i] > a[j]$. Bei paarweise verschiedenen, rückwärts sortierten Werten gibt es $n(n-1)/2$ solche Paare. Jeder Tausch eines benachbarten falsch geordneten Paares beseitigt genau eine Inversion; daher erzwingt diese Eingabe bereits quadratisch viele erforderliche Tausche.

Beispiel: Gnome-Sort für [3,1,2]

Beginne mit Index 1. Der Vergleich $3 > 1$ bewirkt den Tausch zu $[1,3,2]$ und den Rückschritt auf 0. Der Feldanfang setzt den Index ohne Schlüsselvergleich wieder auf 1. Nun ist $1 \leq 3$, also geht es auf 2. Dort werden 3 und 2 getauscht: $[1,2,3]$, Index 1. Die letzten Vergleiche $1 \leq 2$ und $2 \leq 3$ führen zum Ende bei Index 3. Damit wurden fünf Schlüsselvergleiche, zwei Tausche und ein reiner Feldanfangsschritt ausgeführt. Die Ausgangsfolge hatte genau zwei Inversionen, nämlich $3/1$ und $3/2$.

```
int temp = a[i - 1];  
a[i - 1] = a[i];  
a[i] = temp;
```

Typischer Fehler: Ein Feldanfangstest oder ein Schleifenvergleich ist nicht automatisch ein zusätzlicher Vergleich zweier Schlüssel.

Verständnisfrage: Wie viele Inversionen besitzt $[4,3,2,1]$?

Antwort: Sechs: drei mit der 4, zwei mit der 3 und eine mit der 2. Deshalb sind sechs korrigierende Nachbartausche notwendig.

B15 Graphdarstellung, Veränderungen und Zusammenhang

Graphen als Zeichnung, Matrix und Liste lesen

Du übersetzt dieselben Verbindungen zwischen Darstellungen und beachtest Richtung, Gewichte sowie fehlende Kanten.

B15.01 Grundelemente eines Graphen

Ein Graph besteht aus Knoten und Kanten, die Beziehungen zwischen Knoten darstellen. Eine gerichtete Kante hat einen Ausgangs- und einen Zielknoten; eine ungerichtete Verbindung kann in beide Richtungen benutzt werden. Gewichte beschreiben beispielsweise Kosten oder Längen, während ein isolierter Knoten überhaupt keine Verbindung besitzt.

B15.02 Zeilen starten, Spalten enden

In einer Adjazenzmatrix steht der Eintrag in Zeile u und Spalte v für die Kante von u nach v . Ein leeres Feld oder ein vereinbartes Symbol kennzeichnet eine fehlende Kante. Verwendest du dafür 0, musst du klarstellen, dass nicht gleichzeitig echte Kanten mit Gewicht null gemeint sind.

B15.03 Symmetrie richtig deuten

Bei einem ungerichteten gewichteten Graphen ist die Matrix symmetrisch, weil beide Richtungen dasselbe Gewicht besitzen. Fehlt ein Gegenstück oder unterscheiden sich die Gewichte, müssen die Pfeile getrennt behandelt werden. Bei entsprechender Modellkonvention lassen sich symmetrische Gegenkanten gleichen Gewichts als eine ungerichtete Kante darstellen.

B15.04 Adjazenzlisten übersetzen

Die Adjazenzliste eines Knotens nennt seine unmittelbar erreichbaren Nachbarn und gegebenenfalls die Kantengewichte. Übertrage jeden Listeneintrag in genau die entsprechende Matrixzelle und zeichne daraus die Verbindung. Kontrolliere zusätzlich Knoten mit leeren Listen, damit isolierte Knoten oder Knoten ohne ausgehende Kanten nicht versehentlich verschwinden.

Beispiel: Drei Knoten, drei verschiedene Pfeile

Es gibt die Pfeile $A \rightarrow B$ mit Gewicht 2, $B \rightarrow A$ mit Gewicht 5 und $B \rightarrow C$ mit Gewicht 1. Die Liste von A enthält $B:2$, die Liste von B enthält $A:5$ und $C:1$; C besitzt keine ausgehenden Nachbarn. In der Matrix stehen deshalb 2 an Position A/B , 5 an B/A und 1 an B/C . Alle anderen Positionen sind leer. Die Zeichnung benötigt zwischen A und B zwei beschriftete Pfeile: Eine einzelne ungerichtete Kante könnte die beiden unterschiedlichen Kosten nicht korrekt wiedergeben. C bleibt als Knoten sichtbar, obwohl seine eigene Liste leer ist.

Typischer Fehler: Eine symmetrische Nachbarschaft allein genügt bei Gewichten nicht: Auch die Werte der Gegenkanten müssen übereinstimmen.

Verständnisfrage: Ist C in diesem Beispiel isoliert, weil seine Nachbarliste leer ist?

Antwort: Nein. C hat die eingehende Kante $B \rightarrow C$. Isoliert wäre C erst, wenn weder eingehende noch ausgehende Kanten existierten.

B15 Graphdarstellung, Veränderungen und Zusammenhang

Graphen verändern und ihren Zusammenhang prüfen

Du zählst regelhaft erzeugte Kanten und unterscheidest Speicherbedarf, schwachen Zusammenhang und starken Zusammenhang.

B15.05 Kanten aus einer Regel zählen

Verbindet jeder Knoten i die Nachfolger $i+1$ und $i+2$, gibt es bei $n \geq 2$ Knoten $n-1$ Kanten der ersten Art und $n-2$ der zweiten. Zusammen sind das $2n-3$ Kanten. Zähle jede Regelgruppe getrennt und prüfe die Randknoten, bei denen nicht mehr alle Nachfolger existieren.

B15.06 Matrix oder Listen?

Eine Adjazenzmatrix speichert für jedes Knotenpaar einen Eintrag und benötigt deshalb $\Theta(V^2)$ Platz bei V Knoten. Adjazenzlisten benötigen $\Theta(V+E)$ Platz für die Knoten und tatsächlich vorhandenen Kanten. Bei nur linear vielen Kanten wachsen Listen damit deutlich sparsamer als eine vollständige Matrix.

B15.07 Knoten und ihre Kanten entfernen

Wenn du einen Knoten löschst, verschwinden zugleich alle seine ein- und ausgehenden Kanten. Die verbleibenden Nachbarn werden dadurch nicht automatisch miteinander verbunden. Um die neuen Zusammenhangskomponenten zu bestimmen, untersuchst du danach, welche übrigen Knoten noch über Wege verbunden sind; zusätzliche Kanten können solche Komponenten wieder vereinigen.

B15.08 Pfeilrichtungen vorübergehend ignorieren

Der zugrunde liegende ungerichtete Graph behält die Knoten und Verbindungen, ignoriert aber die Pfeilrichtungen. Zwei entgegengesetzte Kanten zwischen demselben Paar ergeben für den Zusammenhang eine einzige Verbindung. Gleiche Gegenkantengewichte können ebenfalls zusammengefasst werden; unterschiedliche Gewichte darfst du für gewichtete Folgeaufgaben nicht ohne weitere Festlegung vereinheitlichen.

B15.09 Zwei Arten von Zusammenhang · Ergänzung

Schwach zusammenhängend bedeutet bei gerichteten Graphen verbunden nach Ignorieren der Richtungen; stark zusammenhängend verlangt gerichtete Wege in beide Richtungen. Ein Pfeil $A \rightarrow B$ verbindet beide daher schwach, aber nicht stark. Pfeile zwischen Listenkästchen einer Speicherzeichnung stellen zudem nur die Listenverkettung dar und erzeugen keine zusätzlichen Graphkanten.

Beispiel: Eine Lücke trennt zwei Bereiche

Im Nachfolgergraphen für die Knoten 1 bis 8 gibt es $7+6=13$ Kanten. Entferne 4 und 5 samt ihren Kanten. Übrig bleiben die Gruppen $\{1,2,3\}$ und $\{6,7,8\}$. Die nächste mögliche Verbindung müsste von 3 nach 6 springen, doch die Regel erlaubt nur Sprungweite eins oder zwei. Somit entstehen zwei schwache Komponenten. Stark zusammenhängend ist dagegen nur jeder einzelne verbliebene Knoten: Jeder Pfeil führt zu einer größeren Zahl.

Typischer Fehler: Das Entfernen eines Knotens verkürzt keine früheren Wege zu neuen direkten Kanten zwischen seinen Nachbarn.

Verständnisfrage: Wie viele starke Komponenten bleiben in diesem Beispiel erhalten?

Antwort: Sechs, eine pro verbliebenem Knoten. Wegen der streng wachsenden Nummern kann kein gerichteter Weg zu seinem Ausgang zurückführen.

B16 Topologische Sortierung und gerichtete Kreisfreiheit

Abhängigkeiten topologisch anordnen

Du findest Reihenfolgen, die alle gerichteten Abhängigkeiten respektieren, und erkennst, warum mehrere Antworten richtig sein können.

B16.01 Jeder Pfeil zeigt nach vorn

Eine topologische Reihenfolge führt jeden Knoten genau einmal auf und setzt bei jeder Kante $u \rightarrow v$ den Knoten u vor v . Die beiden müssen dabei nicht unmittelbar nebeneinander stehen. Du überprüfst eine vorgeschlagene Lösung deshalb anhand aller Kanten, nicht durch einen bloßen Vergleich der Knotennamen.

B16.02 Kreisfreiheit erlaubt eine Reihenfolge

Ein endlicher gerichteter Graph besitzt genau dann eine topologische Sortierung, wenn er keinen gerichteten Kreis enthält. Zwischen unabhängigen Knoten kann die Reihenfolge frei sein, sodass mehrere gültige Ergebnisse entstehen. Ein praktisches Vorgehen wählt wiederholt einen verbleibenden Knoten ohne eingehende Kante und entfernt anschließend dessen ausgehende Kanten.

Beispiel: Fünf Arbeitsschritte vollständig anordnen

Die Abhängigkeiten seien $A \rightarrow C$, $B \rightarrow C$, $B \rightarrow D$, $C \rightarrow E$ und $D \rightarrow E$. Anfangs haben A und B keine Voraussetzungen; C wartet auf zwei Vorgänger, D auf einen und E ebenfalls auf zwei. Wähle zuerst A. Dadurch fällt für C eine Voraussetzung weg, doch B fehlt noch. Nach der anschließenden Wahl von B werden C und D gleichzeitig verfügbar. Wähle C: E wartet danach weiterhin auf D. Wähle D, und zuletzt kann E ausgeführt werden. Die erhaltene Reihenfolge lautet A,B,C,D,E. Prüfe nun jede ursprüngliche Kante: A und B stehen beide vor C, B steht vor D, und sowohl C als auch D stehen vor E. Damit ist die Reihenfolge vollständig bestätigt. Auch B,D,A,C,E ist zulässig: D benötigt nur B und darf daher schon vor A kommen. Dagegen ist A,C,B,D,E falsch, weil C dort vor seinem erforderlichen Vorgänger B steht. Entscheidend sind also die dargestellten Abhängigkeiten, nicht eine alphabetische Ordnung. Beim Entfernen während der Berechnung werden nur bereits erfüllte Voraussetzungen aus der Arbeitskopie gestrichen; die ursprüngliche Aufgabenbeschreibung bleibt für die abschließende Kontrolle maßgeblich.

Typischer Fehler: Topologische Sortierung ordnet keine Zahlenwerte nach Größe und muss keine eindeutige Reihenfolge liefern.

Verständnisfrage: Dürfen C und D in der ersten gefundenen Reihenfolge vertauscht werden?

Antwort: Ja. Beide brauchen dort nur bereits eingeordnete Vorgänger und müssen beide vor E bleiben; zwischen C und D existiert keine Abhängigkeit.

B16 Topologische Sortierung und gerichtete Kreisfreiheit

Kreise beweisen und gezielt erzeugen

Du begründest die Unmöglichkeit einer topologischen Sortierung durch konkrete Pfeile und unterscheidest hinreichende Kriterien.

B16.03 Ein Kreis widerspricht jeder Reihenfolge

Bei einem Kreis $A \rightarrow B \rightarrow C \rightarrow A$ müsste A vor B, B vor C und C wiederum vor A stehen. Diese Anforderungen können in keiner linearen Reihenfolge gleichzeitig erfüllt sein. Ein einziger vollständig gerichteter Kreis genügt daher, um die topologische Sortierbarkeit des gesamten Graphen zu widerlegen.

B16.04 Streng steigende Nummern helfen

Wenn jede Kante von einer kleineren zu einer größeren Knotennummer führt, steigen die Nummern entlang jedes gerichteten Weges streng an. Ein solcher Weg kann nie zu seiner Ausgangsnummer zurückkehren; der Graph ist kreisfrei. Das Kriterium ist hinreichend, aber nicht notwendig: Auch anders nummerierte Graphen können kreisfrei sein.

B16.05 Eine Rückkante schließt einen Weg

Existiert bereits ein gerichteter Weg von u nach v, erzeugt die zusätzliche Kante $v \rightarrow u$ einen Kreis. Suche deshalb zuerst einen tatsächlich vorhandenen Weg und ergänze dann die Gegenrichtung zwischen seinen Endpunkten. Besonders einfach ist eine einzelne vorhandene Kante, deren Rückkante unmittelbar einen Kreis aus zwei Pfeilen erzeugt.

Beispiel: Eine sichere Änderung und zwei schädliche Änderungen

Betrachte die Knoten 1 bis 6 mit allen Kanten $i \rightarrow i+1$ und $i \rightarrow i+2$, soweit das Ziel noch vorhanden ist. Die Reihenfolge 1,2,3,4,5,6 ist topologisch, weil jeder Pfeil nach rechts zeigt. Ergänzt man $1 \rightarrow 6$, bleibt diese Reihenfolge gültig: Die neue Abhängigkeit wird ebenfalls respektiert. Ergänzt man stattdessen $6 \rightarrow 2$, entsteht der konkrete Kreis $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 2$. Alle fünf Pfeile dieses Kreises sind vorhanden; die topologische Sortierung ist damit unmöglich. Noch kürzer wäre die Ergänzung $2 \rightarrow 1$, weil bereits $1 \rightarrow 2$ existiert. Eine Begründung allein mit dem Wort Rückkante wäre weniger aussagekräftig: Durch das Aufschreiben des gesamten Kreises lässt sich sofort kontrollieren, dass jede benötigte Verbindung wirklich im Graphen liegt. Umgekehrt beweist ein Pfeil von einer größeren zu einer kleineren Nummer für sich allein noch keinen Kreis. Erst der vorhandene Rückweg macht ihn im betrachteten Graphen problematisch.

Typischer Fehler: Eine ungerichtete geschlossene Linie ist noch kein gerichteter Kreis; alle Pfeile müssen dem Umlauf folgen.

Verständnisfrage: Ist der Graph mit ausschließlich der Kante $5 \rightarrow 2$ topologisch sortierbar?

Antwort: Ja. Die Reihenfolge 5,2 respektiert die einzige Kante. Fallende Knotennummern allein verhindern eine topologische Ordnung nicht.

B17 Breitensuche mit Warteschlange

Breitensuche mit einer FIFO-Warteschlange

Du besuchst Knoten nach ihrer Nähe zum Start und bewahrst die Reihenfolge bereits wartender Knoten.

B17.01 Vom Start aus schichtweise suchen

Breitensuche beginnt an einem vorgegebenen Startknoten und entdeckt erst dessen direkte Nachbarn, danach deren noch unbekannte Nachbarn. Für die hier betrachtete ungerichtete Variante werden Pfeilrichtungen gegebenenfalls zunächst ignoriert. Nicht erreichbare Knoten gehören zu anderen Komponenten und werden von diesem einzelnen Start aus nicht besucht.

B17.02 Hinten einreihen, vorne entnehmen

Die Warteschlange arbeitet nach dem FIFO-Prinzip: Wer zuerst eingereiht wurde, wird zuerst entnommen. Nach dem Entnehmen eines Knotens fügst du dessen bisher unentdeckte Nachbarn hinten an. Bereits wartende Knoten behalten ihre Plätze; dadurch werden früh entdeckte Entfernungsebenen vor späteren Ebenen bearbeitet.

B17.03 Alphabetische Regeln lokal anwenden

Wenn mehrere neue Nachbarn desselben Knotens verfügbar sind, kann die Aufgabe alphabetisches Einreihen verlangen. Sortiere dafür nur diese neu einzureihenden Nachbarn. Du sortierst nicht nachträglich die gesamte Warteschlange, denn das würde ältere Wartepositionen verändern und könnte die verlangte Breitensuchreihenfolge verletzen.

Beispiel: Warum D vor C besucht wird

Der ungerichtete Graph besitzt die Kanten A-B, A-D, B-C und C-D. Starte in A und markiere Knoten beim Einreihen. Anfangs lautet die Warteschlange [A]. Nach dem Entnehmen von A kommen dessen neue Nachbarn B und D alphabetisch hinein: [B,D]. Entnimm B. Sein Nachbar A ist bereits bekannt, C ist neu und wird hinten ergänzt: [D,C]. Nun wird D vor C bearbeitet, obwohl C alphabetisch früher steht. D entdeckt nichts Neues, denn A ist bekannt und C wurde schon eingereiht; es bleibt [C]. Nach dem Bearbeiten von C ist die Schlange leer. Die Besuchsfolge lautet somit A,B,D,C. A liegt auf Ebene null, B und D auf Ebene eins und C auf Ebene zwei. Diese Ebenen erklären zugleich den Sinn der Warteschlange: Die nächste Ebene wartet so lange, bis die früher entdeckte Ebene abgearbeitet ist. Würde man nach jedem Schritt die ganze Warteschlange alphabetisch sortieren, würde nach B fälschlich C vor D drankommen.

Typischer Fehler: Eine Warteschlange ist kein Stapel: Das zuletzt angehängte Element wird nicht als nächstes entnommen.

Verständnisfrage: Welche beiden kürzesten Wege führen hier von A nach C?

Antwort: A-B-C und A-D-C. Beide bestehen aus zwei Kanten; die alphabetische Regel beeinflusst die Besuchsfolge, nicht diese minimale Entfernung.

B17 Breitensuche mit Warteschlange

Warteschlangenspuren und Entfernungsebenen

Du dokumentierst den vollständigen Suchzustand und verhinderst, dass gemeinsame Nachbarn mehrfach in die Warteschlange gelangen.

B17.04 Den Messzeitpunkt festhalten

Eine Warteschlangentabelle ist nur eindeutig, wenn feststeht, wann ihr Inhalt notiert wird. Üblich ist die Angabe nach Entnahme des aktuellen Knotens und Einreihen aller neuen Nachbarn. Eine zusätzliche Initialzeile zeigt nur den Startknoten; die letzte Zeile enthält die leere Warteschlange und bestätigt den Abschluss.

B17.05 Besuchsfolge und Ebenen unterscheiden

Die Besuchsfolge listet die nacheinander entnommenen Knoten auf, während eine Entfernungsebene alle Knoten mit gleicher minimaler Kantenzahl vom Start zusammenfasst. In ungewichteten Graphen liefert Breitensuche genau diese kürzesten Entfernungen. Innerhalb einer Ebene kann eine andere Nachbarschaftsreihenfolge die Besuchsfolge ändern, ohne die Entfernung zu verändern.

B17.06 Schon beim Einreihen markieren · Ergänzung

Ein Knoten wird als entdeckt markiert, sobald er erstmals in die Warteschlange kommt. Wartest du damit bis zur Entnahme, könnten mehrere bereits bearbeitete Nachbarn denselben Knoten zusätzlich einreihen. Die frühe Markierung sichert, dass jeder Knoten höchstens einmal eingereiht wird, und verhindert unnötige Wiederholungen bei Kreisen.

Beispiel: Ein gemeinsamer Nachbar ohne Doppeleintrag

Verwende die ungerichteten Kanten A-B, A-C, B-D, C-D und C-E; beginne in A und ordne neue Nachbarn alphabetisch. Nach A stehen B und C in der Warteschlange. B entdeckt D und markiert es sofort. Wenn danach C bearbeitet wird, ist D bereits bekannt und wird nicht erneut eingereiht; nur E kommt hinzu. Die vollständige Spur steht unten. Die Entnahmefolge ist A,B,C,D,E. Die Ebenen lauten {A}, {B,C} und {D,E}, also Entfernungen 0,1,1,2,2. Ein kürzester Weg zu E ist A-C-E. Zu D existieren A-B-D und A-C-D; entdeckt wurde D zuerst über B. Der gemeinsame Nachbar verursacht somit weder einen zusätzlichen Besuch noch eine falsche Entfernung. Nach der letzten Zeile sind alle vom Start erreichbaren Knoten verarbeitet, und kein weiterer Warteschlangeneintrag bleibt offen.

Zeitpunkt	Warteschlange
Start	[A]
nach A	[B,C]
nach B	[C,D]
nach C	[D,E]
nach D	[E]
nach E	[]

Typischer Fehler: Entdeckt bedeutet bereits eingereiht; fertig bearbeitet bedeutet, dass auch alle Nachbarn untersucht wurden.

Verständnisfrage: Warum wird D bei der Bearbeitung von C nicht noch einmal angehängt?

Antwort: D wurde bereits beim Bearbeiten von B entdeckt und markiert. Die Markierung gilt auch, solange D selbst noch in der Warteschlange wartet.

B18 Dijkstra und Rekonstruktion kürzester Wege

Dijkstra vorbereiten und sicher auswählen

Du prüfst die Gewichtsvoraussetzung und verstehst vorläufige Distanzen, Vorgänger und endgültig ausgewählte Knoten.

B18.01 Nichtnegative Gewichte voraussetzen

Dijkstra bestimmt kürzeste Wege in einem gewichteten Graphen, dessen Kantengewichte mindestens null sind. Diese Voraussetzung sichert, dass ein längerer Weg durch weiteres Fortsetzen nicht plötzlich kürzer wird. Negative Kanten können die endgültige Festlegung eines Knotens falsch machen; ein negativer Kreis ist für dieses Problem nicht erforderlich.

B18.02 Distanzen und Vorgänger initialisieren

Der Startknoten erhält Distanz 0, alle anderen zunächst ∞ als Zeichen eines noch unbekannten Weges. Vorgänger bleiben zunächst unbelegt. Eine endliche vorläufige Distanz beschreibt immer die Länge eines bereits gefundenen Weges; ∞ bedeutet während der Rechnung noch nicht, dass der Knoten tatsächlich unerreichbar ist.

B18.03 Das kleinste vorläufige Etikett wählen

Wähle unter den noch nicht endgültig markierten Knoten einen mit kleinster vorläufiger Distanz. Bei nichtnegativen Gewichten ist dieser Wert nun endgültig und der Knoten kann markiert werden. Bei Gleichstand sind mehrere korrekte Wahlreihenfolgen möglich; haben alle übrigen Knoten Distanz ∞ , sind sie vom Start nicht erreichbar.

Beispiel: Warum die Nichtnegativität entscheidend ist

Die gerichteten Kanten seien $A \rightarrow B$ mit Gewicht 1, $A \rightarrow C$ mit Gewicht 2 und $C \rightarrow B$ mit Gewicht -3. Vom Start A werden zunächst die vorläufigen Werte $B=1$ und $C=2$ gefunden. Die übliche Dijkstra-Auswahl würde B mit Wert 1 zuerst endgültig markieren. Tatsächlich besitzt aber der Weg $A \rightarrow C \rightarrow B$ die Länge $2 + (-3) = -1$ und ist damit kürzer. Der später bearbeitete Knoten C würde also einen bereits endgültig festgelegten Wert widerlegen. Der Graph enthält keinen Kreis: Das einzelne negative Gewicht genügt, um die Begründung des Algorithmus zu zerstören. Ersetzt man die -3 durch +3, kostet der Umweg dagegen $2 + 3 = 5$; $B=1$ bleibt korrekt. Eine Kante mit Gewicht null ist ebenfalls erlaubt, weil sie einen Weg nicht verkürzt. Beachte außerdem die Bedeutung der Markierung: Bei Breitensuche markierst du schon beim Einreihen, bei Dijkstra bedeutet endgültiges Markieren die Auswahl einer kleinsten vorläufigen Distanz. Eine neu entdeckte endliche Distanz allein reicht dafür nicht aus.

Typischer Fehler: Dijkstra braucht nichtnegative Gewichte; lediglich das Fehlen negativer Kreise genügt nicht als Voraussetzung.

Verständnisfrage: Kann ein nach der Initialisierung mit ∞ beschrifteter Knoten später einen endlichen Wert erhalten?

Antwort: Ja. Sobald über bearbeitete Knoten ein Weg zu ihm gefunden wird, wird ∞ durch dessen Länge ersetzt.

B18 Dijkstra und Rekonstruktion kürzester Wege

Kanten entspannen und kürzeste Wege rekonstruieren

Du führst eine Dijkstra-Rechnung vollständig durch und kontrollierst das Ergebnis mithilfe seiner Vorgängerkette.

B18.04 Entspannen heißt einen besseren Weg prüfen

Für eine ausgehende Kante $u \rightarrow v$ berechnest du den Kandidaten $\text{dist}[u] + \text{gewicht}(u, v)$. Ist er kleiner als $\text{dist}[v]$, ersetzt du die bisherige Distanz und setzt u als Vorgänger von v . So bleibt die gespeicherte Wegbeschreibung passend zum Zahlenwert; ohne Vorgängeränderung wäre die spätere Rekonstruktion möglicherweise falsch.

B18.05 Jeden Zustand eindeutig protokollieren

Führe für jeden Knoten seine Distanz und seinen Vorgänger und notiere außerdem den neu endgültig markierten Knoten. Eine Tabellenzeile kann den Zustand nach dem Entspannen seiner ausgehenden Kanten zeigen. Halte diesen Zeitpunkt durchgängig ein, damit vorläufige Verbesserungen und die Reihenfolge der endgültigen Auswahl nachvollziehbar bleiben.

B18.06 Der Vorgängerkette rückwärts folgen

Beginne beim Ziel und folge den gespeicherten Vorgängern bis zum Start. Die erhaltene Folge läuft rückwärts und muss anschließend umgekehrt werden. Addiere die Gewichte der so gefundenen Kanten als Kontrolle; ihre Summe muss dem endgültigen Distanzwert entsprechen, während ein unerreichbares Ziel keinen solchen Weg besitzt.

Beispiel: Vollständige Rechnung ab A

Es gelten $A \rightarrow B:4$, $A \rightarrow C:1$, $C \rightarrow B:2$, $C \rightarrow D:5$ und $B \rightarrow D:1$. Anfangs sind nur $A=0$ und die anderen Distanzen ∞ bekannt. Nach A werden $B=4$ über A und $C=1$ über A eingetragen. C wird als nächster Knoten gewählt: Der Weg über C verbessert B auf $1+2=3$ und erreicht D zunächst mit $1+5=6$. Danach wird B gewählt und verbessert D auf $3+1=4$. Zuletzt wird D endgültig markiert. Die Markierungsfolge lautet A,C,B,D. Rückwärts folgt man $D \leftarrow B \leftarrow C \leftarrow A$; vorwärts erhält man $A \rightarrow C \rightarrow B \rightarrow D$ mit Gewicht $1+2+1=4$. Die direkte Alternative über $A \rightarrow B \rightarrow D$ kostet $4+1=5$, die Alternative $A \rightarrow C \rightarrow D$ kostet $1+5=6$. Damit bestätigt auch ein Vergleich aller hier möglichen Wege das Tabellenergebnis. In der Spur bedeutet ein Strich, dass noch kein Vorgänger festgelegt ist.

nach Wahl	A	B	C	D
Start	0/-	∞ /-	∞ /-	∞ /-
A	0/-	4/A	1/A	∞ /-
C	0/-	3/C	1/A	6/C
B	0/-	3/C	1/A	4/B
D	0/-	3/C	1/A	4/B

Typischer Fehler: Aktualisiere Distanz und Vorgänger gemeinsam; ein neuer Zahlenwert mit einem alten Vorgänger beschreibt möglicherweise keinen gültigen kürzesten Weg.

Verständnisfrage: Welcher Vorgänger steht nach dem vollständigen Lauf bei B?

Antwort: C. Der zuerst eingetragene Vorgänger A wird ersetzt, weil der Weg über C die Distanz von 4 auf 3 verbessert.

B19 Minimale Spann bäume, Kruskal und Union-Find

Mit Kruskal einen minimalen Spannbaum finden

Du verbindest alle Knoten möglichst kostengünstig und erkennst, wann eine weitere Kante lediglich einen Kreis erzeugen würde.

B19.01 Das Ziel eines minimalen Spannbaums

Ein Spannbaum eines zusammenhängenden ungerichteten Graphen enthält alle Knoten und verbindet sie ohne Kreis. Unter allen solchen Bäumen sucht ein minimaler Spannbaum die kleinste Summe der Kantengewichte. Er muss nicht den kürzesten Weg zwischen einem bestimmten Knotenpaar enthalten; das ist eine andere Optimierungsaufgabe.

B19.02 Kanten nach ihrem Gewicht bearbeiten

Kruskal betrachtet die Kanten in aufsteigender Gewichtsreihenfolge. Eine Kante wird aufgenommen, wenn ihre Endknoten bislang zu verschiedenen Komponenten der ausgewählten Kanten gehören. Sie verbindet diese Komponenten zu einer größeren; anfangs ist jeder Knoten eine eigene Komponente und es wurde noch keine Kante ausgewählt.

B19.03 Kreise verwerfen und rechtzeitig beenden

Liegen beide Endknoten bereits in derselben Komponente, existiert zwischen ihnen schon ein Weg aus aufgenommenen Kanten. Die zusätzliche Kante würde einen Kreis schließen und wird verworfen. Sobald bei V Knoten $V-1$ Kanten aufgenommen und alle Knoten verbunden sind, ist der Spannbaum vollständig; weitere Prüfungen sind unnötig.

Beispiel: Ein Kreis wird erkannt, bevor alles verbunden ist

Der ungerichtete Graph hat die Kanten A-B:1, B-C:2, A-C:3, C-D:4 und B-D:6. Anfangs sind die Komponenten $\{A\}$, $\{B\}$, $\{C\}$ und $\{D\}$. Kruskal nimmt A-B mit Gewicht 1 auf und vereinigt A mit B. Danach verbindet B-C mit Gewicht 2 diese Komponente mit C. Nun folgt A-C mit Gewicht 3. A und C sind jedoch bereits über A-B-C verbunden; diese Kante wird verworfen. Die nächste Kante C-D mit Gewicht 4 verbindet den bisher alleinstehenden Knoten D. Die drei ausgewählten Kanten bilden bei vier Knoten einen vollständigen Spannbaum: A-B, B-C und C-D. Das Gesamtgewicht ist $1+2+4=7$. B-D mit Gewicht 6 muss nicht mehr geprüft werden. An diesem Beispiel lässt sich der Unterschied zur bloßen Auswahl der billigsten drei Kanten sehen: Die Kanten mit Gewichten 1,2,3 würden nur ein Dreieck zwischen A,B,C erzeugen und D überhaupt nicht anbinden. Erst die zusätzliche Komponentenprüfung sorgt dafür, dass ein Baum über alle Knoten entsteht.

Typischer Fehler: Die billigsten $V-1$ Kanten sind nicht automatisch ein Spannbaum; Zusammenhang und Kreisfreiheit müssen gleichzeitig erfüllt sein.

Verständnisfrage: Warum ist nach der Auswahl von C-D keine weitere Kante erforderlich?

Antwort: Alle vier Knoten liegen jetzt in einer Komponente, und es wurden drei kreisfreie Kanten aufgenommen. Das erfüllt genau die Eigenschaften eines Spannbaums.

B19 Minimale Spannbäume, Kruskal und Union-Find

Union-Find und die vollständige Kruskal-Spur

Du hältst ausgewählte Graphkanten und interne Elternzeiger auseinander und erklärst unterschiedliche korrekte Endstrukturen.

B19.04 Reihenfolge und Ergebnis getrennt angeben

Eine vollständige Kruskal-Lösung nennt die bearbeiteten Kanten mit Aufnahme oder Verwerfung, danach die tatsächlichen Baumkanten und deren Gewichtssumme. Eine verworfene Kante gehört zwar zur Ablaufspur, aber nicht zum Spannbaum. Bei gleichen Gewichten kann eine andere Bearbeitungsreihenfolge einen anderen ebenfalls minimalen Baum ergeben.

B19.05 Komponenten durch Repräsentanten verwalten

Union-Find speichert für jeden Knoten einen Elternzeiger; durch wiederholtes Folgen dieser Zeiger erreicht find den Repräsentanten seiner Komponente. Anfangs zeigt jeder Knoten auf sich selbst. Unterschiedliche Repräsentanten bedeuten verschiedene Komponenten; union verbindet ihre Repräsentanten, wenn Kruskal eine entsprechende Graphkante aufnimmt.

B19.06 Die Endstruktur kontrollieren

Prüfe am Ende, ob alle durch den Spannbaum verbundenen Knoten bei find denselben Repräsentanten besitzen. Die Elternzeiger von Union-Find sind interne Verwaltungsverbindungen und müssen nicht die ausgewählten Graphkanten nachbilden. Deshalb zeichnest du den Spannbaum und die Union-Find-Struktur getrennt und beschriftest ihre jeweilige Bedeutung.

B19.07 Größe, Gleichstand und Pfadkompression · [Ergänzung](#)

Bei Vereinigung nach Größe wird die kleinere Komponente unter die größere gehängt; bei Gleichstand brauchst du eine festgelegte Wahlregel. Pfadkompression lässt bei find besuchte Knoten direkt auf den Repräsentanten zeigen. Dadurch können verschiedene Elternzeiger entstehen, obwohl die vertretene Komponentenaufteilung und das Kruskal-Ergebnis unverändert bleiben.

Beispiel: Elternzeiger zu einer konkreten Kantenfolge

Bearbeite B-D:1, A-C:3, C-E:4 und A-B:5. Vereinige nach Größe und wähle bei Gleichstand den alphabetisch kleineren Repräsentanten. Nach B-D zeigt D auf B; nach A-C zeigt C auf A. Durch C-E kommt E zu A, sodass diese Komponente drei Knoten enthält. A-B hängt dann B mit seiner Zweierkomponente unter A. Ohne abschließende Pfadkompression lauten die Elternzeiger $A \rightarrow A$, $B \rightarrow A$, $C \rightarrow A$, $D \rightarrow B$, $E \rightarrow A$. Alle fünf Knoten gehören zur Komponente A; die aufgenommenen Graphkanten wiegen zusammen $1+3+4+5=13$. Ein nachfolgendes find(D) mit Pfadkompression ändert $D \rightarrow B$ in $D \rightarrow A$. Keine Graphkante des Spannbaums wird dadurch verändert.

Typischer Fehler: Zeichne einen Union-Find-Elternzeiger nicht ungeprüft als zusätzliche Kante in den ursprünglichen Graphen ein.

Verständnisfrage: Kann eine andere Gleichstandsregel einen anderen Repräsentanten erzeugen?

Antwort: Ja. Die Wurzel kann anders heißen, während dieselben Knoten weiterhin genau eine gemeinsame Komponente bilden.

Java-Kurzreferenz für den Unterricht

Diese Seite bündelt häufig verwechselte Begriffe. Lies bei Unsicherheit den genannten Themenbereich und überprüfe die Regel an einem kleinen Beispiel.

Unterscheidung	Merksatz	Thema
Wert und Referenz	Primitive Variablen enthalten Werte; Referenzvariablen verweisen auf Objekte oder enthalten null.	A03, A12
== und equals	== prüft bei Referenzen Identität. equals kann eine vereinbarte inhaltliche Gleichheit ausdrücken.	A06, B04
Überladen und Überschreiben	Überladen: verschiedene Parameterlisten. Überschreiben: neue Implementierung einer geerbten Instanzmethode.	A10, A11
Statischer und dynamischer Typ	Der deklarierte Typ bestimmt zulässige Zugriffe; der tatsächliche Objekttyp bestimmt überschreibende Instanzmethoden.	A11
Instanzfeld und static	Ein Instanzfeld gehört zum Objekt. Ein static-Feld gehört zur Klasse und wird von deren Instanzen gemeinsam genutzt.	A09
Arraylänge und Belegung	length gibt die feste Zahl der Plätze an. Ein Belegungszähler zählt die tatsächlich gespeicherten Elemente.	A07, A13
null und leere Sammlung	null bezeichnet keinen vorhandenen Array- oder Listenträger. Ein existierendes Array kann trotzdem Länge 0 haben.	A07, B05
Flache und tiefe Kopie	Eine neue Arrayhülle oder neue Listenknoten können weiterhin dieselben Datenobjekte referenzieren.	A13, B05

Vier kurze Prüfungen vor der Abgabe

1. Sind alle gelesenen lokalen Variablen initialisiert? 2. Bleiben sämtliche Indizes innerhalb der gültigen Grenzen? 3. Wird ein Ergebnis mit dem richtigen Typ berechnet, bevor es zugewiesen wird? 4. Sind sowohl der Normalfall als auch die erlaubten Randfälle bedacht?

Sichtbarkeit

private: Zugriff im umschließenden Klassenkontext. Ohne Modifizierer: Zugriff im Paket. protected: Zugriff im Paket und unter zusätzlichen Bedingungen in Unterklassen anderer Pakete. public: öffentlicher Member; auch die Zugänglichkeit des verwendeten Typs muss passen. Die Einzelheiten erläutert A09.

Laufzeiten mit Voraussetzungen lesen

n bezeichnet die Zahl der Elemente, h die Baumhöhe. Die Tabelle nennt übliche enge Wachstumsklassen und die dafür nötigen Annahmen; eine O-Obergrenze darf mathematisch auch größer sein.

Verfahren / Operation	Aufwand	Voraussetzung / Hinweis
Lineare Suche	Worst Case $\Theta(n)$	Elemente der Reihe nach prüfen.
Binäre Suche	Worst Case $\Theta(\log n)$	Sortiertes Feld mit direktem Indexzugriff.
Interpolationssuche	Erwartet $\Theta(\log \log n)$; Worst $\Theta(n)$	Für den günstigen Erwartungswert ist eine geeignete Schlüsselverteilung nötig.
Suchbaum: Suche	$O(h)$	Entartet $h = \Theta(n)$; bei AVL $h = \Theta(\log n)$.
Liste: Zugriff auf Index i	Worst Case $\Theta(n)$	Durchlaufen der Verknüpfungen.
Einfügen am Listenkopf	$\Theta(1)$	Ohne vorherige Duplikatsuche oder andere lineare Zusatzprüfung.
Merge- und Heapsort	Worst Case $\Theta(n \log n)$	Unterschiedliche Arbeitsweise, gleiche Wachstumsklasse.
Quicksort	Mittlerer Fall $\Theta(n \log n)$; Worst $\Theta(n^2)$	Der mittlere Fall benötigt ein geeignetes Eingabe- oder Randomisierungsmodell.
Insertion, Bubble, Cocktail, Gnome	Worst Case $\Theta(n^2)$	Linearer Best Case bei geeigneter Ausführung; Bubble/Cocktail mit Abbruchprüfung.
Selectionsort	$\Theta(n^2)$ Vergleiche	Auch ein bereits sortiertes Feld wird nach Minima durchsucht.

Graphverfahren gezielt unterscheiden

Breitensuche ordnet erreichbare Knoten nach der Zahl der Kanten vom Start. Dijkstra bestimmt kürzeste gewichtete Wege bei nichtnegativen Kantengewichten. Eine topologische Sortierung ordnet einen gerichteten azyklischen Graphen kantenverträglich. Kruskal bestimmt einen minimalen Spannbaum eines zusammenhängenden ungerichteten gewichteten Graphen.

Frage vor jeder Laufzeitaussage: Welche Eingabegröße zählt, welcher Fall wird betrachtet, welches Kostenmodell gilt und welche Vorbedingungen sind erfüllt? Zusätzliche Prüfungen können den Aufwand einer ansonsten konstanten Operation verändern.

Aufgaben und zugehörige Lösungen

Die folgenden Dateien sind die Grundlage der Themenauswahl. Die Aufgaben liegen im Hauptordner informatik-Klausuren; die Lösungen stehen in output/pdf/. Jede ID bezeichnet ein Aufgabenblatt und die zugehörige Lösungsdatei.

09 | 19.02.2018 | Einführung in die Programmierung

Aufgaben: 09_EidP_Klausur_19.02.2018.pdf

Lösung: 09_EidP_Klausur_19.02.2018_Loesungen.pdf

10 | 18.07.2018 | Einführung in die Programmierung

Aufgaben: 10_EidP_Klausur_18.07.2018.pdf

Lösung: 10_EidP_Klausur_18.07.2018_Loesungen.pdf

11 | 19.02.2019 | Einführung in die Programmierung

Aufgaben: 11_EidP_Klausur_19.02.2019.pdf

Lösung: 11_EidP_Klausur_19.02.2019_Loesungen.pdf

12 | 22.07.2019 | Einführung in die Programmierung

Aufgaben: 12_EidP_Klausur_22.07.2019.pdf

Lösung: 12_EidP_Klausur_22.07.2019_Loesungen.pdf

15 | 10.03.2021 | Einführung in die Programmierung

Aufgaben: 15_EidP_Klausur_A_10_03_2021.pdf

Lösung: 15_EidP_Klausur_A_10_03_2021_Loesungen.pdf

16 | 27.08.2021 | Einführung in die Programmierung

Aufgaben: 16_EidP_Klausur_27.08.2021.pdf

Lösung: 16_EidP_Klausur_27.08.2021_Loesungen.pdf

17 | 21.02.2022 | Einführung in die Programmierung

Aufgaben: 17_EidP_Klausur_A_21.02.2022.pdf

Lösung: 17_EidP_Klausur_A_21.02.2022_Loesungen.pdf

20 | 28.07.2023 | Einführung in die Programmierung

Aufgaben: 20_EidP_Klausur_28.07.2023 (MI und INF).pdf

Lösung: 20_EidP_Klausur_28.07.2023 (MI und INF)_Loesungen.pdf

Aufgaben und zugehörige Lösungen

Fortsetzung des Quellenverzeichnisses: Algorithmen und Datenstrukturen. Die Aufgabenverweise verwenden dieselben IDs wie die ursprünglichen Dateinamen und die bereits erstellte Themenübersicht.

38 | 11.08.2022 | Algorithmen und Datenstrukturen

Aufgaben: 38-Klausur vom 11.08.2022_korr.pdf

Lösung: 38-Klausur vom 11.08.2022_korr_Loesungen.pdf

39 | 08.02.2023 | Algorithmen und Datenstrukturen

Aufgaben: 39-Klausur vom 08.02.2023_final.pdf

Lösung: 39-Klausur vom 08.02.2023_final_Loesungen.pdf

40 | 02.08.2023 | Algorithmen und Datenstrukturen

Aufgaben: 40-Klausur vom 02.08.2023_korr.pdf

Lösung: 40-Klausur vom 02.08.2023_korr_Loesungen.pdf

41 | 09.02.2024 | Algorithmen und Datenstrukturen

Aufgaben: 41-Klausur vom 09.02.2024_final.pdf

Lösung: 41-Klausur vom 09.02.2024_final_Loesungen.pdf

42 | 20.08.2024 | Algorithmen und Datenstrukturen

Aufgaben: 42-Klausur vom 20.08.2024_final.pdf

Lösung: 42-Klausur vom 20.08.2024_final_Loesungen.pdf

43 | 31.01.2025 | Algorithmen und Datenstrukturen

Aufgaben: 43-Klausur vom 31.01.2025_final.pdf

Lösung: 43-Klausur vom 31.01.2025_final_Loesungen.pdf

44 | 14.08.2025 | Algorithmen und Datenstrukturen

Aufgaben: 44-Klausur vom 14.08.2025_final.pdf

Lösung: 44-Klausur vom 14.08.2025_final_Loesungen.pdf

45 | 06.02.2026 | Algorithmen und Datenstrukturen

Aufgaben: 45-Klausur vom 06.02.2026_final_korr.pdf

Lösung: 45-Klausur vom 06.02.2026_final_korr_Loesungen.pdf

So findest du die passende Übung

Wähle im Inhaltsverzeichnis den Themenbereich, beachte seine Klausur-IDs und öffne die Themenübersicht für die genaue Teilaufgabe. Bearbeite die Originalaufgabe zunächst selbstständig. Vergleiche anschließend mit der Lösung und begründe insbesondere Abweichungen, Randfälle und verwendete Konventionen.

Begleitdatei: Themenuebersicht_aller_Klausuren_und_Loesungen.pdf (ebenfalls in output/pdf/).