

Informatik kompakt

37 Hauptthemen auf 20 Seiten

Zum Erklären, Wiederholen und gemeinsamen Üben: Jeder Themenblock enthält die Grundidee, ein Beispiel und eine Verständnisfrage mit Antwort. Die Kennungen entsprechen der vollständigen Themenübersicht.

Die Kurzfassung bündelt die Hauptthemen. Für die ausführlichen Unterpunkte und weitere Beispiele dienen die Seitenverweise zur 90-Seiten-Langfassung.

A · Programmierung mit Java

A01	Java: Aufbau und Ausführung	2
A02	Zahlensysteme und Binärdarstellung	2
A03	Datentypen und Operatoren	3
A04	Kontrollstrukturen und Eingaben	3
A05	Zahlenalgorithmen und Extremwerte	4
A06	Stringalgorithmen	4
A07	Arrays und Messdaten	5
A08	Matrizen, Bilder und Muster	5
A09	Klassen und Konstruktoren	6
A10	Vererbung und Polymorphie	6
A11	Referenztypen und Methodenauflösung	7
A12	Speicherbilder und Ablaufanalyse	7
A13	UML und Assoziationen	8
A14	Radio, Lager und MediaPlayer	8
A15	Garten, Regal und Gebäudeflächen	9
A16	Putzplan und Reinigungskosten	9
A17	Veröffentlichungen und Themengebiete	10
A18	Polymorphe Anwendungsmodelle	10

B · Algorithmen und Datenstrukturen

B01	Algorithmus und Divide-and-Conquer	11
B02	Asymptotische Laufzeiten	11
B03	Schleifen und Rekursion	12
B04	Java: Referenzen und Vergleiche	12
B05	Sammlungen und verkettete Listen	13
B06	Prioritäten, Warteschlangen und Stapel	13
B07	Binärbäume und Suchbäume	14
B08	AVL-Bäume	14
B09	Binäre Heaps	15
B10	B-Bäume	15
B11	Lineare, binäre und Interpolationssuche	16
B12	Hash-Tabellen	16
B13	Sortierverfahren und Laufzeiten	17
B14	Cocktail-Sort und Gnome-Sort	17
B15	Graphen und Zusammenhang	18
B16	Topologische Sortierung	18
B17	Breitensuche	19
B18	Dijkstra und kürzeste Wege	19
B19	Spannbäume, Kruskal und Union-Find	20

Unterrichtsimpuls: Erst die Grundidee in eigenen Worten erklären lassen, dann das Beispiel gemeinsam prüfen und zuletzt die Frage ohne Blick auf die Antwort bearbeiten.

A01

Java: Programmaufbau, Syntax und Ausführung

Ein Java-Programm besteht aus Klassen, deren Methoden Anweisungen ausführen. Namen sind groß-/kleinschreibungsabhängig; reservierte Schlüsselwörter dürfen keine eigenen Bezeichner sein. Namenskonventionen verbessern die Lesbarkeit, bestimmen aber nicht allein die sprachliche Gültigkeit. Eine Methode besitzt einen Namen, Parameter und einen Rückgabotyp. Bei einem Ergebnis muss die return-Anweisung einen dazu passenden Wert liefern; void bezeichnet eine Methode ohne Rückgabewert.

Der Compiler übersetzt Quelltext in Bytecode und prüft dabei Syntax sowie statische Typregeln. Die Java Virtual Machine führt diesen Bytecode aus; häufig verwendete Teile können durch JIT-Kompilierung in Maschinencode übersetzt werden. Dadurch kann derselbe Bytecode auf unterschiedlichen Systemen mit passender JVM laufen. Hilfsklassen benötigen keine eigene main-Methode. Ein übersetzbares Programm kann trotzdem fachlich falsch rechnen.

Beispiel: Vom Ausdruck zur Ausgabe. Eine Methode mit Rückgabotyp int soll 2 und 3 addieren und gibt deshalb 5 zurück. Gibt sie stattdessen $2 * 3$ zurück, ist der Ausdruck ebenfalls typkorrekt, liefert aber das fachlich falsche Ergebnis 6.

Typischer Fehler: Erfolgreiches Übersetzen beweist noch nicht, dass die Lösung zur Aufgabenstellung passt.

Frage: Wofür ist die JVM nötig?

Antwort: Sie stellt die Ausführungsumgebung für den Bytecode bereit.

Vertiefung: Langfassung S. 5-6 | Themenübersicht A01

A02

Zahlensysteme und binäre Zahlendarstellungen

Ein Stellenwertsystem bewertet jede Ziffer mit einer Potenz seiner Basis. Binärzahlen verwenden die Basis 2, Hexadezimalzahlen die Basis 16. Eine Hexadezimalziffer entspricht genau vier Bits. Zum Umrechnen addiert man Stellengewichte oder gewinnt Ziffern durch wiederholte Division mit Rest. Führende Nullen ändern den Wert nicht, können aber eine festgelegte Bitbreite sichtbar machen.

Negative ganze Zahlen werden bei festgelegter Bitbreite häufig im Zweierkomplement gespeichert: Bits des positiven Betrags umkehren und 1 addieren. Bei acht Bit reicht der Wertebereich von -128 bis 127. Fließkommazahlen speichern Vorzeichen, Exponent und Signifikand. Viele Dezimalbrüche, etwa 0,1, besitzen keine endliche Binärdarstellung und werden deshalb gerundet. Mehr Speicher erhöht die Genauigkeit, macht aber nicht jede Dezimalzahl exakt darstellbar.

Beispiel: Darstellung von 13 und -13. Die Binärzahl 00001101 entspricht $8 + 4 + 1 = 13$ und hexadezimal 0D. Für -13 ergeben Bitumkehr und Addition von 1 die acht Bits 11110011. Die Rückprüfung lautet $243 - 256 = -13$.

Typischer Fehler: Ein Bitmuster lässt sich erst mit bekannter Bitbreite und Vorzeichenkonvention eindeutig deuten.

Frage: Wie viele Bits hat eine Hexadezimalziffer?

Antwort: Vier Bits.

Vertiefung: Langfassung S. 7-8 | Themenübersicht A02

A03

Datentypen, Operatoren und sichere Auswertung

Der Datentyp bestimmt, welche Werte und Operationen zulässig sind. `int` und `long` speichern ganze Zahlen, `float` und `double` Näherungen an reelle Zahlen; `boolean` beschreibt Wahrheitswerte. Bei ganzzahliger Division entfällt der Nachkommateil. Ein Cast muss vor der entscheidenden Operation stehen, wenn diese bereits mit höherer Genauigkeit oder größerem Wertebereich stattfinden soll. Ein nachträglicher Cast repariert weder verlorene Nachkommastellen noch einen vorherigen Überlauf.

Bei Ausdrücken sind Reihenfolge und Seiteneffekte getrennt zu verfolgen. Postinkrement liefert zunächst den alten Wert, Präinkrement den erhöhten Wert. Das Pluszeichen kann Zahlen addieren oder Strings verketteten. Die Operatoren `&&` und `||` werten den rechten Teil nur aus, wenn er für das Ergebnis benötigt wird. Das ermöglicht Schutzbedingungen vor potenziell ungültigen Operationen, beispielsweise vor einer Division durch 0.

Beispiel: Division und Schutzbedingung. Bei `int`-Werten ergibt `7 / 2` den Wert 3, während `(double) 7 / 2` den Wert 3,5 liefert. In `n != 0 && 12 / n > 2` wird für `n = 0` keine Division ausgeführt.

Typischer Fehler: Eine Umwandlung des fertigen Ergebnisses kann eine zuvor falsche Rechenart nicht rückgängig machen.

Frage: Was ergibt `(double) (7 / 2)`?

Antwort: 3.0, weil die Ganzzahldivision vor dem Cast erfolgt.

Vertiefung: Langfassung S. 9-11 | Themenübersicht A03

A04

Kontrollstrukturen, Konsoleneingabe und Ablaufspuren

Kontrollstrukturen legen fest, welche Anweisungen wann ausgeführt werden. `if/else` unterscheidet Fälle; `switch` eignet sich für eine Auswahl aus festen Alternativen. `for` und `while` prüfen vor einem Durchlauf, `do-while` prüft danach und läuft daher mindestens einmal. Initialisierung, Bedingung, Verarbeitung und Aktualisierung sollten in einer Ablaufspur getrennt erscheinen. Auch ein letzter, falscher Bedingungstest kann durch Inkrementoperatoren noch Werte verändern.

Für Konsoleneingaben liest Scanner beispielsweise Zahlen, Wörter oder ganze Zeilen. Ein vereinbarter Abschlusswert beendet eine Eingabefolge und wird nicht als Nutzwert mitgezählt. Summen und Zähler werden vor der Schleife initialisiert. Der Mittelwert ergibt sich aus Summe geteilt durch Anzahl; bei leerer Folge ist er nicht definiert. Werden bei einem Gleichstand sämtliche Spiele neu eingegeben, müssen auch alle bisherigen Punkte zurückgesetzt werden.

Beispiel: Mittelwert mit Abschlusswert. Die Eingabe lautet 4, 7, 9, -1; -1 beendet die Folge. Nach drei Nutzwerten ist die Summe 20 und die Anzahl 3. Mit Gleitkommadivision erhält man den Mittelwert `20 / 3`, also ungefähr 6,67.

Typischer Fehler: Den Abschlusswert vor der Verarbeitung prüfen, damit Summe und Anzahl korrekt bleiben.

Frage: Wie oft läuft `do-while` mindestens?

Antwort: Einmal.

Vertiefung: Langfassung S. 12-13 | Themenübersicht A04

A05

Zahlenalgorithmen, Häufigkeiten und Auswahl von Extremwerten

Zahlenalgorithmen zerlegen eine Aufgabe in wiederholte einfache Schritte. Modulo erkennt Teilbarkeit, während ganzzahlige Division Ziffern oder Faktoren entfernt. Für die Primfaktorzerlegung wird jeder gefundene Faktor vollständig herausdividiert. Nach vollständiger Prüfung der nötigen Kandidaten ist ein verbleibender Rest größer als 1 prim. Dezimalziffern erhält man mit Rest modulo 10 und anschließender Division durch 10.

Häufigkeitsarrays zählen Werte eines begrenzten Bereichs; der Index steht dabei für den gezählten Wert. Eine Maximumsuche speichert stets das beste bisherige Ergebnis. Bei Gleichständen muss die verlangte Regel ausdrücklich umgesetzt werden, etwa der kleinste häufigste Wert. Für die k kleinsten Eingaben verwaltet man zusätzlich die Zahl belegter Plätze. Leere Eingaben, die Zahl 0 und negative Werte brauchen passende, ausdrücklich festgelegte Randfallbehandlungen.

Beispiel: Primfaktoren und Quersumme. Für 36 wird zweimal durch 2 geteilt: 36 wird zu 18 und dann zu 9. Zweimaliges Teilen durch 3 ergibt 1. Deshalb ist $36 = 2 * 2 * 3 * 3$. Die Quersumme lautet $3 + 6 = 9$.

Typischer Fehler: Freie Auswahlplätze sind keine eingegebenen Zahlen und dürfen nicht in Produkte eingehen.

Frage: Welche Primfaktoren hat 1?

Antwort: Keine; das leere Produkt hat den Wert 1.

Vertiefung: Langfassung S. 14-16 | Themenübersicht A05

A06

Stringalgorithmen: Filtern, Wiederholen, Suchen und Löschen

Stringalgorithmen verarbeiten Zeichen in ihrer ursprünglichen Reihenfolge. `length()` liefert die Zeichenanzahl, `charAt(i)` das Zeichen am gültigen Index i . Beim Filtern wird jedes erlaubte Zeichen in das Ergebnis übernommen. Soll nur der erste Treffer entfernt werden, merkt ein Zustandsflag, ob bereits gelöscht wurde. Stringinhalte vergleicht man mit `equals`; `==` prüft dagegen, ob dieselbe Referenz vorliegt.

Für die Teilwortsuche wird jede mögliche Startposition geprüft, ohne über das Wortende hinauszugreifen. Wiederholungen eines Grundworts lassen sich erkennen, indem passende Teiler der Gesamtlänge als Kandidaten untersucht werden. Bei wiederholtem Löschen können über einer Löschgrenze neue Treffer entstehen; deshalb ist eine erneute Suche nötig. Jede erfolgreiche Löschung eines nichtleeren Wortes verkürzt den Text und begründet so die Terminierung. Die Reihenfolge unterschiedlicher Löschwörter kann das Endergebnis verändern.

Beispiel: Ein Zeichen oder alle Treffer löschen. Im Wort `banane` steht `a` an den Indizes 1 und 3. Wird nur das erste `a` entfernt, entsteht `bnane`. Werden alle `a` entfernt, entsteht `bnne`. In beiden Fällen bleibt die Reihenfolge der übrigen Zeichen erhalten.

Typischer Fehler: Das leere Löschwort verkürzt den Text nicht und benötigt daher eine eigene Regel.

Frage: Welcher Vergleich prüft Stringinhalte?

Antwort: `equals`.

Vertiefung: Langfassung S. 17-19 | Themenübersicht A06

A07

Arrays: Grundlagen, Änderungen an Ort und Stelle und Messdaten

Ein Array speichert eine feste Anzahl von Elementen. Seine Indizes beginnen bei 0 und enden bei `length - 1`. Die Länge verändert sich nicht durch das Ändern eines Elements.

Mehrdimensionale Java-Arrays sind Arrays von Arrays; einzelne Zeilen können deshalb unterschiedlich lang sein. Eine leere Zeile enthält keinen Messwert und darf nicht mit einer vorhandenen Messung des Werts 0 verwechselt werden.

Bei Änderungen im vorhandenen Array ist die Verarbeitungsrichtung entscheidend. Zum Umkehren tauscht man spiegelbildliche Positionen und stoppt in der Mitte. Präfixsummen entstehen dagegen von links nach rechts, weil der Vorgänger bereits die benötigte Teilsumme enthält. Soll ein Messwert angehängt werden, legt man eine größere Zeile an, kopiert die Werte und ersetzt die Zeilenreferenz. Voraussetzungen wie gültige Indizes, vorhandene Zeilen und ausreichende Zahlenbereiche müssen vor der Verarbeitung stimmen.

Beispiel: Präfixsummen bilden. Aus `[2, 5, 1, 4]` wird zunächst `[2, 7, 1, 4]`, danach `[2, 7, 8, 4]` und schließlich `[2, 7, 8, 12]`. Jeder Eintrag enthält nun die Summe vom ursprünglichen Anfang bis zu dieser Position.

Typischer Fehler: Eine bereits überschriebene Zelle enthält möglicherweise nicht mehr ihren ursprünglichen Wert.

Frage: Was ist der letzte Index bei Länge 4?

Antwort: 3.

Vertiefung: Langfassung S. 20-22 | Themenübersicht A07

A08

Matrizen, Bildtransformationen und zweidimensionale Muster

Matrizen werden meist als rechteckige zweidimensionale Arrays dargestellt. Eine äußere Schleife durchläuft Zeilen, eine innere die Spalten. Das Skalarprodukt summiert komponentenweise Produkte. Bei der Matrixmultiplikation wird jede Ergebniszeile aus einer Zeile der linken und einer Spalte der rechten Matrix berechnet. Deshalb müssen die inneren Dimensionen übereinstimmen; die äußeren Dimensionen bestimmen die Form des Ergebnisses.

Beim Transponieren werden Zeilen- und Spaltenindex vertauscht. Eine symmetrische Matrix ist quadratisch und stimmt mit ihrer Transponierten überein. Auch Bildoperationen beruhen auf Indexabbildungen: Spiegeln kehrt eine Richtung um, eine Drehung vertauscht die Dimensionen und kehrt zusätzlich eine Richtung um. Gitter- und Schachbrettmuster entstehen aus der Parität von Zeilen, Spalten oder Blockkoordinaten. Bei Java-byte muss man die vorzeichenbehaftete Speicherung von einer vorzeichenlosen Graustufeninterpretation unterscheiden.

Beispiel: Eine Ergebniszeile berechnen. Die Zeile `[1, 2, 3]` und die Spalte `[4, 5, 6]` ergeben das Skalarprodukt $1 * 4 + 2 * 5 + 3 * 6 = 32$. Eine Matrix mit 2 Zeilen und 3 Spalten mal einer 3-mal-4-Matrix ergibt eine 2-mal-4-Matrix.

Typischer Fehler: Matrixmultiplikation ist kein bloßes Multiplizieren gleich positionierter Einträge.

Frage: Welche Form hat die Transponierte einer 2-mal-3-Matrix?

Antwort: 3 Zeilen und 2 Spalten.

Vertiefung: Langfassung S. 23-25 | Themenübersicht A08

A09

Klassen, Konstruktoren, Kennungen und Zugriffsrechte

Eine Klasse beschreibt gemeinsame Eigenschaften und Verhalten; ein Objekt besitzt seine eigenen Instanzfelder. `static`-Felder gehören zur Klasse und werden von ihren Objekten gemeinsam verwendet. Ein Konstruktor initialisiert neue Objekte und kann einen anderen Konstruktor aufrufen. Bei Vererbung wird die Initialisierung der Oberklasse in die Konstruktorfolge einbezogen. `this` bezeichnet das aktuelle Objekt und trennt gleichnamige Felder von Parametern.

Zugriffsrechte begrenzen die Sichtbarkeit: `private` schützt interne Zustände, `public` stellt zugängliche Operationen bereit, und ohne Modifizierer gilt Paketsichtbarkeit. `protected` erlaubt zusätzliche Zugriffe in Unterklassen, unterliegt außerhalb des Packages aber besonderen Regeln. Getter und kontrollierte Änderungsmethoden unterstützen die Kapselung. Eine fortlaufende Kennung benötigt einen gemeinsamen Zähler und ein getrenntes Feld pro Objekt. Mit `private final` und ohne Setter bleibt diese individuelle Kennung nach der Initialisierung unveränderlich.

Beispiel: Kennungen vergeben. Ein gemeinsamer Zähler startet bei 1. Das erste Objekt übernimmt 1 und erhöht den Zähler auf 2. Das zweite übernimmt 2; danach steht der Zähler auf 3. Die Objektkennungen bleiben 1 und 2, obwohl der gemeinsame Zähler weiterwächst.

Typischer Fehler: Ein `static`-Feld ist keine getrennte Speicherstelle für jedes einzelne Objekt.

Frage: Welches Schlüsselwort bezeichnet das aktuelle Objekt?

Antwort: `this`.

Vertiefung: Langfassung S. 26-27 | Themenübersicht A09

A10

Abstraktion, Interfaces, Vererbung und Polymorphie

Abstraktion beschreibt gemeinsame Fähigkeiten, ohne jede Ausführung bereits festzulegen. Eine abstrakte Klasse kann Felder, Konstruktoren sowie konkrete und abstrakte Methoden besitzen; direkt erzeugt werden nur konkrete Unterklassen. Interfaces beschreiben gemeinsame Verträge. Eine Klasse kann eine direkte Klassenoberklasse und mehrere Interfaces besitzen. Eine konkrete Klasse muss die erforderlichen abstrakten Operationen bereitstellen; passende geerbte Implementierungen können diese Pflicht bereits erfüllen.

Polymorphie erlaubt es, verschiedene Untertypen über denselben Oberklassen- oder Interface-Typ anzusprechen. Bei einer überschriebenen Instanzmethode entscheidet der tatsächliche Objekttyp, welche Implementierung ausgeführt wird. Dadurch kann beispielsweise dieselbe Schleife unterschiedliche Medien oder Pflanzen passend verarbeiten. `super.methode()` wählt gezielt die Oberklassenimplementierung; ein gewöhnlicher Methodenaufruf innerhalb dieser Implementierung kann jedoch wieder dynamisch gebunden werden. Öffentliche Interface-Operationen müssen durch entsprechend öffentliche Methoden erfüllt werden.

Beispiel: Gemeinsame Operation, unterschiedliche Ergebnisse. Ein Buch liefert über die gemeinsame Methode für die Breite 2 cm, eine CD 1 cm. Eine Schleife über beide Medien ruft jeweils die passende Implementierung auf und berechnet die Gesamtbreite $2 + 1 = 3$ cm.

Typischer Fehler: Eine abstrakte Klasse wird nicht direkt mit `new` instanziiert.

Frage: Was bestimmt die überschriebene Implementierung?

Antwort: Der tatsächliche Typ des Objekts.

Vertiefung: Langfassung S. 28-29 | Themenübersicht A10

A11

Referenztypen, Methodenauflösung und statische Typfehler

Der Compiler wählt anhand deklarierter Typen eine zugängliche, passende Methodensignatur. Danach bestimmt der tatsächliche Objekttyp die überschreibende Implementierung. Überladung benötigt unterschiedliche Parameterlisten; nur ein anderer Rückgabetyt reicht nicht. Eine allein im Untertyp vorhandene Überladung wird durch eine Oberklassenreferenz nicht automatisch verfügbar.

Felder folgen einer anderen Regel: Gleichnamige Felder in Ober- und Unterklasse sind getrennte Speicherstellen, und direkter Feldzugriff richtet sich nach dem statischen Referenztyp. Ein Unterklassenobjekt darf einer Oberklassenreferenz zugewiesen werden. Ein Cast in die Gegenrichtung muss zum Laufzeittyp passen; instanceof hilft bei der Prüfung. Ein Cast verändert die Sicht auf ein Objekt, erzeugt aber kein neues Objekt und ändert dessen tatsächlichen Typ nicht.

Beispiel: Feldzugriff und Methodenaufruf. Eine Referenz vom Typ A zeigt auf ein B-Objekt. A besitzt ein öffentliches `x = 1`, B ein verdeckendes öffentliches `x = 2`. Der Zugriff über diese A-Referenz liest 1. Eine von B überschriebene Instanzmethode wird dagegen in der B-Fassung ausgeführt.

Typischer Fehler: Dynamische Methodenbindung darf nicht auf direkte Feldzugriffe übertragen werden.

Frage: Erzeugt ein Referenzcast eine Objektkopie?

Antwort: Nein.

Vertiefung: Langfassung S. 30-31 | Themenübersicht A11

A12

Speicherabbilder, Referenzen und objektorientierte Ablaufanalyse

Ein Speicherabbild trennt gemeinsame Klassenfelder, Objekte und aktive Methodenaufrufe. Für jeden Aufruf werden Parameter und lokale Variablen in einem eigenen Rahmen erfasst. `this` zeigt auf das Empfängerobjekt; weitere Parameter können auf andere Objekte zeigen. Eine Referenz speichert einen Verweis, nicht das ganze Objekt. Mehrere Variablen können deshalb dasselbe Objekt erreichen und Änderungen daran gemeinsam beobachten. `null` bezeichnet einen fehlenden Verweis.

Für eine Ablaufanalyse verfolgt man Erzeugung, Konstruktoren, Feldänderungen und Aufrufe in der tatsächlichen Reihenfolge. Bei Vererbung gehören auch die geerbten Felder zum Objektzustand. Bereits zurückgekehrte Methoden besitzen keinen aktiven Aufrufrahmen mehr. Während die rechte Seite einer Zuweisung noch berechnet wird, ist die Zielvariable noch nicht aktualisiert. Java übergibt Parameter als Wertkopien: Bei primitiven Werten wird die Zahl kopiert, bei Objekten die Referenz.

Beispiel: Gemeinsames Objekt, getrennte Variablen. `a` und `b` verweisen auf dasselbe Objekt mit `wert = 4`. Eine Änderung über `a` auf 9 ist anschließend auch über `b` sichtbar. Wird nur `a` auf `null` gesetzt, zeigt `b` weiterhin auf das Objekt mit `wert = 9`.

Typischer Fehler: Eine Referenzzuweisung kopiert keinen vollständigen Objektzustand.

Frage: Was wird bei einem Objektparameter kopiert?

Antwort: Die Referenz, nicht das Objekt.

Vertiefung: Langfassung S. 32-33 | Themenübersicht A12

A13

UML und allgemeine Assoziationsverwaltung

Ein UML-Klassendiagramm beschreibt Typen, Attribute, Methoden und Beziehungen. Die Zeichen +, -, # und ~ stehen für public, private, protected und Paketsichtbarkeit; unterstrichene Attribute werden static. Vererbung wird mit extends umgesetzt, eine Interface-Realisierung mit implements. Multiplizitäten bestimmen, wie viele Partnerobjekte zulässig oder erforderlich sind.

Eine begrenzte Assoziation lässt sich durch ein Referenzarray und einen Belegungszähler verwalten. Der Konstruktor erfüllt Mindestzahlen; Einfügen und Entfernen erhalten die Grenzen. Bei lückenlosem Bestand rücken nach dem Löschen die folgenden Referenzen nach links. Identitätsvergleiche verhindern doppelte Zuordnung desselben Objekts. Eine defensive Arraykopie schützt die Sammlung vor fremdem Umsortieren, kopiert aber nicht die Fachobjekte selbst.

Beispiel: Ein Objekt entfernen. Aus [a, b, c, null] mit anzahl = 3 wird nach Entfernen von b die Folge [a, c, null, null] mit anzahl = 2. c bleibt dasselbe Objekt; nur seine Arrayposition ändert sich. Der frei gewordene letzte Platz wird geleert.

Typischer Fehler: Kapazität ist nicht Belegung: Ein Array mit zehn Plätzen erzeugt keine zehn Fachobjekte.

Frage: Reicht eine Obergrenzenprüfung für eine Beziehung mit Mindestzahl?

Antwort: Nein. Auch Konstruktion und Entfernen müssen die Mindestzahl sichern.

Vertiefung: Langfassung S. 34-35 | Themenübersicht A13

A14

UML-Modelle: InternetRadio, Lager und MediaPlayer

InternetRadio, Lager und MediaPlayer verbinden gemeinsame Objekttypen mit unterschiedlichen Verwaltungsregeln. Das Radio nutzt eine Geräteoberklasse, ein IPGerät-Interface und eine begrenzte Favoritenliste. Im Lager liefern konkrete Behälter wie Karton und Dose ihre Volumen über die gemeinsame abstrakte Oberklasse. So kann eine Schleife verschiedene Behälter einheitlich auswerten.

Der MediaPlayer verwaltet Audio- und Videoobjekte über den gemeinsamen Typ Medium. Eine Aufnahme benötigt sowohl einen freien Arrayplatz als auch genügend Speicherbudget. Entfernen gibt das Budget zurück und liefert die entfernte Referenz. Favoriten werden lückenlos gehalten; beim MediaPlayer sind dagegen freie Zwischenplätze vorgesehen. Bei gleichen Mediennamen entscheidet der kleinste passende Index. Der gespeicherte Medienverbrauch muss für eine korrekte Rückbuchung unverändert bleiben.

Beispiel: Speicherbudget erhalten. Sind noch 100 Einheiten frei, senkt ein Medium mit Verbrauch 40 das Budget auf 60. Nach seinem Entfernen stehen wieder 100 Einheiten zur Verfügung. Ein weiteres Medium mit Verbrauch 70 passt während der Speicherung nicht, auch wenn noch ein Arrayplatz frei ist.

Typischer Fehler: Ein freier Platz allein beweist nicht, dass die Aufnahme zulässig ist.

Frage: Warum können Karton und Dose gemeinsam summiert werden?

Antwort: Beide bieten über ihre Oberklasse die passende Volumenoperation an.

Vertiefung: Langfassung S. 36-37 | Themenübersicht A14

A15

UML-Modelle: Garten, Regal und Gebäudeflächen

Garten, Regal und Gebäude prüfen Mengen sowie Flächen-, Breiten- oder Gewichtsbudgets. Im Garten liefern Pflanzen ihren Flächenbedarf über ein Interface. `testeFlaeche` vergleicht die Summe mit Länge mal Breite des Gartens. Diese Bestandsprüfung bedeutet nicht automatisch, dass schon jede Aufnahme eine Überschreitung verhindern muss; maßgeblich ist der Methodenvertrag.

Ein Regal begrenzt seine Brettzahl, jedes Brett wiederum seine Medienzahl. Bücher und CDs stellen Breite und Gewicht über einen gemeinsamen Typ bereit. Ein Medium passt nur, wenn beide Summen die Brettgrenzen einhalten; gesucht wird gegebenenfalls das erste geeignete Brett. Ein Gebäude nimmt Räume nur bei ausreichender Kapazität und Gesamtfläche auf. Die gesamte Raumfläche und die reine Wohnfläche sind getrennte Summen, weil Garagen keine Wohnräume sind.

Beispiel: Eine Grenze genau erreichen. Ein Garten mit 6 mal 5 Flächeneinheiten bietet 30. Ein Pflanzenbedarf von 30 besteht die Flächenprüfung, ein Bedarf von 31 nicht. Davon unabhängig muss die vorgeschriebene Anzahl der Pflanzen eingehalten werden. Die Flächensumme beschreibt keine konkrete Anordnung der Pflanzen.

Typischer Fehler: Die Bedingung kleiner oder gleich darf nicht versehentlich durch echt kleiner ersetzt werden.

Frage: Genügt freie Brettbreite, um ein Buch aufzunehmen?

Antwort: Nein. Auch Tragkraft und freie Medienplätze müssen ausreichen.

Vertiefung: Langfassung S. 38-39 | Themenübersicht A15

A16

UML-Modell: Putzplan, Fenster und Reinigungskosten

Ein Gebäude besitzt im Modell genau einen Putzplan und eine begrenzte, nichtleere Fenstersammlung. Der Putzplan verwaltet Reinigungseinträge. Fenster- und Bodenreinigung spezialisieren den abstrakten Reinigungseintrag und liefern ihre Kosten über dieselbe Operation. Fensterkosten entstehen aus Flächen und Kostensatz; ein Bodenreinigungseintrag kann einen festen Preis liefern.

Die Vollständigkeitsprüfung lautet: Für jedes Gebäudefenster muss mindestens eine passende Fensterreinigung existieren. Deshalb betrachtet eine äußere Schleife die Fenster und eine innere Suche die geeigneten Reinigungseinträge. `instanceof` erkennt den passenden Untertyp, ein Cast ermöglicht dessen besondere Zugriffe. Gesucht wird das konkrete Fensterobjekt mit `==`, nicht bloß ein anderes Fenster gleicher Größe. Erst wenn alle Fenster einen Treffer besitzen, darf die Gesamtprüfung erfolgreich sein.

Beispiel: Fehlende Zuordnung erkennen. Das Gebäude hat `f1`, `f2` und `f3`. `R1` enthält `f1` und `f2`; `R2` enthält nur `f2`. Die Prüfung liefert `false`, weil `f3` fehlt. Ergänzt `R2` auch `f3`, liefert sie `true`. Dass `f2` mehrfach vorkommt, verletzt die Forderung mindestens einmal nicht.

Typischer Fehler: Die umgekehrte Prüfung, ob jede Reinigung irgendein Gebäudefenster enthält, übersieht möglicherweise unversorgte Fenster.

Frage: Ersetzt eine Bodenreinigung die fehlende Zuordnung von `f3`?

Antwort: Nein. Sie kann Kosten beitragen, enthält aber keine passende Fensterzuordnung.

Vertiefung: Langfassung S. 40-41 | Themenübersicht A16

A17

UML-Modell: Veröffentlichungen und Themengebiete

Das Veröffentlichungsmodell trennt Rollen und Beziehungen. Editor erbt von Reviewer, dieser von der abstrakten Klasse UserAcct. Ein Editor ist deshalb zugleich Reviewer und UserAcct. Author und Paper sind dagegen assoziiert. Mindestzahlen müssen schon bei der Konstruktion erfüllt werden. Ein gemeinsamer static-Zähler vergibt Kennungen; ein privates final-Instanzfeld bewahrt die jeweilige Kennung dauerhaft.

Bei der bidirektionalen Editor-Paper-Beziehung speichert der Editor seine Papers und jedes Paper höchstens einen Herausgeber. Beide Richtungen müssen nach einer Änderung zusammenpassen. Die eigene Editor-Sammlung und die geerbte Reviewer-Sammlung sind unterschiedliche Beziehungen. SubjectArea kann andere SubjectArea-Objekte als Unterthemen enthalten. Diese Selbstassoziation ist keine Vererbung und garantiert ohne weitere Bedingungen weder Baumstruktur noch Kreisfreiheit.

Beispiel: Den Herausgeber wechseln. Paper p gehört zu e1 und verweist auf e1. Beim Entfernen wird p aus dessen Sammlung gelöscht und der Rückverweis geleert. Erst eine erfolgreiche Aufnahme bei e2 setzt beide neuen Verweise. Fehlt e2 die Kapazität, darf p nicht fälschlich als zugeordnet erscheinen.

Typischer Fehler: Nur den Rückverweis zu ändern lässt die alte Sammlung inkonsistent zurück.

Frage: Verhindert eine Obergrenze von fünf Unterthemen bereits Themenzyklen?

Antwort: Nein. Sie begrenzt nur die Anzahl zugeordneter Unterthemen.

Vertiefung: Langfassung S. 42-43 | Themenübersicht A17

A18

Polymorphe Anwendungsmodelle und Text-/Zahlengabungen

Polymorphe Modelle trennen eine gemeinsame Darstellung von typabhängigen Angaben. toString kann beispielsweise Beruf, Gehalt oder Tierlaut über überschreibbare Methoden erfragen. Diese Aufrufe richten sich nach dem tatsächlichen Objekt, auch wenn die gemeinsame Ausgabemethode aus der Oberklasse stammt. Deshalb können unterschiedliche Personen oder Tiere in einem gemeinsamen Array stehen.

Bei EBus, Pkw und Lkw werden Verbrauch, Treibstoff und weitere Eigenschaften passend bereitgestellt. Für einen positiven Verbrauch v in Litern pro 100 Kilometer gilt 100/v Kilometer pro Liter. Sonderausgaben müssen den Aufgabenvertrag erfüllen: Eine Gleitkommadivision durch +0.0 liefert bei positivem endlichem Zähler Infinity. Im Wurfmodell werden begrenzte Dauer, dynamisch berechnete mittlere Geschwindigkeit und Weite in Ausführungsreihenfolge verfolgt. Ein lokaler Parameter bleibt vom gleichnamigen Objektfeld getrennt.

Beispiel: Eine geerbte Ausgabe verstehen. Person p verweist auf einen Manager. Ruft die geerbte Methode toString intern getGehalt auf, gilt die für dieses Manager-Objekt wirksame Implementierung. Eine eigene Kopie der gesamten Textausgabe ist deshalb nicht erforderlich; die gemeinsame Struktur bleibt wiederverwendbar.

Typischer Fehler: Die Klasse, in der der Methodentext steht, legt nicht alle späteren virtuellen Aufrufe fest.

Frage: Warum ist Infinity nicht automatisch die gewünschte EBus-Ausgabe?

Antwort: Die Aufgabe kann ausdrücklich eine besondere Rückgabe wie 0.0 verlangen.

Vertiefung: Langfassung S. 44-45 | Themenübersicht A18

B01

Algorithmusbegriff und Divide-and-Conquer

Ein Algorithmus beschreibt eine allgemeine Lösung für eine ganze Klasse zulässiger Eingaben. Seine Beschreibung besteht aus endlich vielen eindeutigen und ausführbaren Anweisungen. Nach der hier verwendeten Klausurdefinition muss die Ausführung für jede zulässige Eingabe nach endlich vielen Schritten mit einem Ergebnis enden. Eine endliche Beschreibung allein garantiert das nicht: Auch eine Endlosschleife lässt sich kurz aufschreiben.

Divide-and-Conquer zerlegt ein Problem in kleinere gleichartige Teilprobleme, löst diese und setzt die Teilergebnisse zusammen. Basisfälle werden unmittelbar behandelt. Die Verkleinerung erklärt die Terminierung; die korrekte Zusammensetzung erklärt zusammen mit korrekten Teillösungen das Gesamtergebnis. Ein Algorithmus ist unabhängig von seiner Programmiersprache und kann korrekt sein, obwohl es eine schnellere Lösung gibt.

Beispiel: Mergesort an vier Zahlen. Die Folge [4, 1, 3, 2] wird in [4, 1] und [3, 2] zerlegt. Nach weiterer Teilung werden [1, 4] und [2, 3] gebildet. Beim Zusammenführen wird jeweils der kleinere vordere Wert gewählt: [1, 2, 3, 4].

Typischer Fehler: Nur kleinere Teilprobleme zu erzeugen reicht ohne passende Basisfälle und korrekte Zusammenführung nicht aus.

Frage: Warum endet die Zerlegung?

Antwort: Jeder Teilbereich wird kleiner, bis er höchstens ein Element enthält.

Vertiefung: Langfassung S. 46-47 | Themenübersicht B01

B02

Asymptotische Laufzeiten und Aufwand von Operationen

Die Laufzeit wird als Funktion der Eingabegröße n beschrieben, etwa der Anzahl gespeicherter Elemente. Typische Wachstumsklassen sind konstant, logarithmisch, linear, linearithmisch, quadratisch und exponentiell. Konstante Faktoren und kleinere Summanden ändern die asymptotische Klasse nicht. O bezeichnet eine obere Schranke; Θ bezeichnet eine passende obere und untere Schranke und damit das genaue Wachstum bis auf konstante Faktoren.

Best Case, Average Case und Worst Case beziehen sich auf unterschiedliche Eingaben derselben Größe. Eine mittlere Laufzeit braucht Annahmen über deren Verteilung. Bei zusammengesetzten Verfahren werden aufeinanderfolgende Kosten addiert; Wiederholungen multiplizieren den Aufwand des Rumpfs. Eine scheinbar einzelne Operation kann selbst linear teuer sein, beispielsweise die Suche nach dem Ende einer Liste ohne Endzeiger.

Beispiel: Einmal sortieren, mehrfach suchen. Für n Elemente und q Anfragen kostet Mergesort mit anschließender binärer Suche $O(n \log n + q \log n)$. q lineare Suchen benötigen im schlechtesten Fall $\Theta(qn)$. Die Vorbereitung lohnt sich daher besonders bei vielen Anfragen.

Typischer Fehler: Aus $O(n^2)$ folgt nicht, dass ein Verfahren tatsächlich quadratisch wächst; auch lineares Wachstum erfüllt diese Obergrenze.

Frage: Welche Klasse hat $3n^2 + 7n + 5$?

Antwort: $\Theta(n^2)$, weil für große n der quadratische Summand das Wachstum bestimmt.

Vertiefung: Langfassung S. 48-49 | Themenübersicht B02

B03

Schleifenanalyse, Rekursion und Rekursionsspeicher

Bei einer Schleifenanalyse zählen zunächst die tatsächlichen Durchläufe und die Kosten des Rumpfs. Zwei ineinander verschachtelte Schleifen mit jeweils n Durchläufen erzeugen n^2 Wiederholungen des inneren Rumpfs. Läuft die innere Schleife beim äußeren Index i genau i -mal, entsteht die Summe $1 + 2 + \dots + n$. Wiederholtes Halbieren oder Verdoppeln führt dagegen typischerweise zu logarithmisch vielen Schritten.

Für Rekursionen trennt man Gesamtarbeit und maximale Tiefe. Ein Rekursionsbaum zeigt sämtliche Aufrufe; ein längster Wurzel-Blatt-Pfad zeigt die gleichzeitig aktiven Aufrufe. Nacheinander ausgeführte Geschwisteraufrufe erhöhen die Tiefe nicht. Bei zwei halbierten Teilproblemen und linearer Zusatzarbeit gilt $T(n) = 2T(n/2) + \Theta(n)$, also $\Theta(n \log n)$. Pro Ebene fällt insgesamt lineare Arbeit an.

Beispiel: Halbierungsrekursion für $n = 8$. Jeder Aufruf teilt sich bis zum Basisfall $n = 1$ in zwei Hälften. Die Ebenen enthalten 1, 2, 4 und 8 Aufrufe, insgesamt 15. Gleichzeitig aktiv sind höchstens vier Aufrufe einschließlich des Erstaufrufs; bei konstantem Speicher je Aufruf ist der Stapelplatz logarithmisch.

Typischer Fehler: Die Gesamtzahl aller Aufrufe ist nicht die Rekursionstiefe; zusätzlich reservierte Daten benötigen eigenen Speicher.

Frage: Wie groß ist $1 + \dots + n$?

Antwort: $n(n + 1)/2$, daher $\Theta(n^2)$.

Vertiefung: Langfassung S. 50-52 | Themenübersicht B03

B04

Java-Implementierung, Referenzen und Vergleiche

Knotenklassen speichern Daten und Referenzen auf weitere Knoten. Ein leeres Teilstück wird häufig durch null dargestellt. Öffentliche Methoden bilden die nutzbare Schnittstelle; private Hilfsmethoden übernehmen interne Arbeit, etwa eine rekursive Suche ab einem Knoten. Generische Klassen wie `Knoten<T>` erlauben unterschiedliche Datentypen bei gemeinsamer Struktur. Konstruktoren müssen einen gültigen Anfangszustand herstellen.

Bei Objekten prüft `==` die Identität, `equals` die jeweils definierte inhaltliche Gleichheit. `compareTo` liefert eine negative Zahl, 0 oder eine positive Zahl gemäß einer Ordnung; Gleichwertigkeit in dieser Ordnung muss nicht mit `equals` übereinstimmen.

Kurzschlussauswertung kann einen Zugriff absichern. Assertions dokumentieren Vorbedingungen, sind aber nicht immer aktiviert. Rückgabesignale für erfolglose Suchen müssen deshalb klar vereinbart sein.

Beispiel: Gleicher Inhalt, verschiedene Objekte. Zwei separat mit `new String("Ada")` erzeugte Strings sind verschiedene Objekte: `==` ergibt false, `equals` ergibt true. Bei `knoten != null && knoten.wert == 7` wird das rechte Teilstück nur ausgewertet, wenn die Referenz gültig ist; so wird dieser null-Zugriff vermieden.

Typischer Fehler: Ein `int`-Produkt kann bereits vor der Zuweisung an `long` überlaufen; mindestens einen Faktor vorher nach `long` umwandeln.

Frage: Was bedeutet `compareTo`-Ergebnis 0?

Antwort: Gleichwertigkeit nach der verwendeten Sortierordnung.

Vertiefung: Langfassung S. 53-54 | Themenübersicht B04

B05

List, Set, Map und allgemeine Operationen verketteter Listen

Eine List speichert eine positionsbezogene Folge mit möglichen Duplikaten. Ein Set enthält keine doppelten Elemente gemäß seiner Gleichheitsregel; eine Map ordnet jedem Schlüssel höchstens einen Wert zu. ArrayList bietet konstanten Indexzugriff und amortisiert konstantes Anhängen. Bei verketteten Listen folgt der Indexzugriff dagegen den Knoten und benötigt im schlechtesten Fall lineare Zeit.

In einer einfach verketteten Liste zeigt jeder Knoten auf seinen Nachfolger; der letzte verweist auf null. Einfügen und Entfernen ändern Referenzen. Ist der Vorgänger bereits bekannt, kostet das Ausketten seines Nachfolgers konstant viel Zeit. Das Finden dieses Vorgängers kann jedoch linear sein. Beim Verketteten, Halbieren oder Umkehren müssen Kopf, gegebenenfalls Ende und Randfälle leerer Listen erhalten bleiben. Neue Knoten können weiterhin dieselben Datenobjekte referenzieren: Das ist eine flache Kopie.

Beispiel: Einen inneren Knoten entfernen. $A \rightarrow B \rightarrow C$ wird durch $A.nachfolger = B.nachfolger$ zu $A \rightarrow C$. Soll B zurückgegeben werden, wird seine Referenz vorher gesichert. Beim Entfernen des Kopfes muss stattdessen der Kopfzeiger geändert werden.

Typischer Fehler: Konstantes Umhängen von Referenzen macht eine vorherige lineare Suche nicht kostenlos.

Frage: Wann ist Anhängen konstant teuer?

Antwort: Wenn ein gültiger Endzeiger vorhanden ist und keine zusätzliche Suche erforderlich ist.

Vertiefung: Langfassung S. 55-57 | Themenübersicht B05

B06

Prioritätslisten, Warteschlangen und Stapel

Eine Warteschlange arbeitet nach FIFO: zuerst hinein, zuerst hinaus. Mit Kopf- und Endzeiger sind Einreihen hinten und Entnehmen vorn konstant teuer. Ein Stapel arbeitet nach LIFO: Der zuletzt abgelegte Eintrag wird zuerst entnommen. Verkettet reichen dafür push und pop am Kopf. Eine verlangte Duplikatsuche kann beide Strukturen jedoch um linearen Aufwand ergänzen.

Eine Prioritätsliste entnimmt die Aktion mit der höchsten Priorität. In einer absteigend sortierten Liste liegt sie am Kopf; das Einfügen sucht dafür im schlechtesten Fall linear nach der Position. Eine unsortierte Liste erlaubt konstantes Einfügen am Kopf, benötigt aber lineare Maximumsuche. Für gleiche Prioritäten muss eine Reihenfolge vereinbart werden. Bei negativen Prioritäten wird das erste tatsächliche Element als bisheriges Maximum verwendet.

Beispiel: Drei verschiedene Entnahmeregeln. In der Reihenfolge A mit Priorität 2, B mit 7 und C mit 4 werden drei Aufträge aufgenommen. Die FIFO-Warteschlange entnimmt A, der LIFO-Stapel C und die Prioritätsliste B. Alle drei Ergebnisse sind korrekt, weil die Strukturen unterschiedliche Auswahlregeln besitzen.

Typischer Fehler: Eine Initialisierung des Maximums mit 0 übersieht Listen, deren Prioritäten sämtlich negativ sind.

Frage: Wie bleibt das erste Maximum erhalten?

Antwort: Nur bei einer echt größeren Priorität den bisherigen Kandidaten ersetzen.

Vertiefung: Langfassung S. 58-59 | Themenübersicht B06

B07

Binärbäume, Suchbäume und Traversierungen

Ein Binärbaum besitzt pro Knoten höchstens ein linkes und ein rechtes Kind. Blätter haben keine Kinder. Wir zählen die Höhe in Ebenen: Der leere Baum hat Höhe 0, ein Einzelknoten Höhe 1. Bei h Ebenen sind höchstens $2^h - 1$ Knoten möglich. In einem perfekten Baum sind alle Ebenen vollständig besetzt; ein vollständiger Baum darf eine nur linksbündig gefüllte letzte Ebene besitzen.

Ein Suchbaum verlangt eine globale Ordnung: Hier liegen alle kleineren Schlüssel links und alle größeren rechts; Duplikate werden ausgeschlossen. Dadurch folgt die Suche nur einem Pfad und kostet $O(h)$, bei Entartung $O(n)$. Das Minimum liegt ganz links. Inorder besucht links, Wurzel, rechts und liefert sortierte Schlüssel; Preorder besucht die Wurzel zuerst, Postorder zuletzt. Vollständige Traversierungen besuchen jeden Knoten einmal.

Beispiel: Drei Traversierungen desselben Baums. Die Wurzel 4 besitzt links 2 und rechts 6. Preorder ergibt 4, 2, 6; Inorder 2, 4, 6; Postorder 2, 6, 4. Umgekehrte Inorder liefert 6, 4, 2 und damit absteigende Reihenfolge.

Typischer Fehler: Lokale Eltern-Kind-Vergleiche allein beweisen keine globale Suchbaumordnung.

Frage: Wie hoch ist eine Kette mit n Knoten?

Antwort: n Ebenen; Richtungswechsel zwischen linken und rechten Kindern ändern daran nichts.

Vertiefung: Langfassung S. 60-62 | Themenübersicht B07

B08

AVL-Bäume und Balancebedingungen

Ein AVL-Baum ist ein Suchbaum mit einer Balancebedingung: An jedem Knoten unterscheiden sich die Höhen seiner beiden Teilbäume um höchstens 1. Wir verwenden den Balancefaktor Höhe links minus Höhe rechts. Zulässig sind also -1, 0 und 1. Andere Vorzeichenkonventionen sind bei durchgehender Verwendung möglich. Die Bedingung gilt für jeden Knoten, nicht nur für die Wurzel.

Die Balance begrenzt die Höhe auf $O(\log n)$; deshalb benötigt eine Suche höchstens logarithmisch viele Schritte. Einfügen kann die Balance verletzen, die durch passende Rotationen wiederhergestellt wird. Bei einer Prüfung lassen sich Höhe und Gültigkeit in einem rekursiven Durchlauf berechnen. Ein ungültiger Teilbaum wird sofort weitergemeldet. So wird jeder Knoten einmal bearbeitet, statt seine Höhe immer wieder neu zu bestimmen.

Beispiel: Höhen von unten berechnen. Die Wurzel 8 hat einen linken Teilbaum der Höhe 3 und einen rechten der Höhe 2. Ihr Balancefaktor ist 1 und damit zulässig. Hat dort jedoch ein Knoten Teilbaumhöhen 2 und 0, ist der gesamte Baum trotzdem kein AVL-Baum.

Typischer Fehler: AVL-Balance verlangt keine identischen Teilbaumhöhen und keinen perfekten Baum.

Frage: Wie teuer ist die gemeinsame Höhen- und Balanceprüfung?

Antwort: $\Theta(n)$, wenn jeder Knoten genau einmal verarbeitet wird.

Vertiefung: Langfassung S. 63-64 | Themenübersicht B08

B09

Binäre Heaps und Prüfung der Heapordnung

Ein binärer Heap kombiniert zwei unabhängige Eigenschaften. Seine Form ist vollständig: Alle oberen Ebenen sind besetzt, die letzte wird von links gefüllt. In einem Max-Heap ist jeder Elternschlüssel mindestens so groß wie seine Kinder; im Min-Heap höchstens so groß. Gleiche Schlüssel sind erlaubt. Geschwister werden nicht untereinander geordnet; die globale Suchbaumordnung gilt hier ebenfalls nicht.

Durch die Eltern-Kind-Bedingung liegt im Max-Heap ein größter Schlüssel an der Wurzel. Ist die Gültigkeit bereits bekannt, kostet sein Ablesen $O(1)$. Eine vollständige Prüfung der Schlüsselordnung besucht dagegen alle Knoten und kostet $\Theta(n)$. Bei einer verzeigten Darstellung muss die Form gesondert überprüft oder ausdrücklich vorausgesetzt werden. Fehlende Kinder und leere Teilbäume sind zulässige lokale Fälle.

Beispiel: Gültige Heapordnung ohne Suchbaumordnung. Die Wurzel 9 mit linkem Kind 4 und rechtem Kind 7 bildet einen Max-Heap. Beide Kinder sind kleiner als 9, und die Form ist vollständig. Ein Suchbaum ist dies nicht, denn der rechte Teilbaum enthält mit 7 einen kleineren Schlüssel als die Wurzel.

Typischer Fehler: Eine bestandene Schlüsselprüfung beweist bei beliebiger Baumform noch keinen gültigen Heap.

Frage: Muss das linke Kind kleiner als das rechte sein?

Antwort: Nein, die Heapordnung vergleicht Eltern mit Kindern.

Vertiefung: Langfassung S. 65-66 | Themenübersicht B09

B10

B-Bäume: Suche, Ausgabe, Höhe und Ordnung

Ein B-Baum speichert mehrere sortierte Schlüssel pro Knoten. Bei k Schlüsseln hat ein innerer Knoten $k + 1$ Kinder für die dazwischenliegenden Werteintervalle. Alle Blätter haben dieselbe Tiefe; deshalb lässt sich die Höhe eines gültigen B-Baums entlang eines Wurzel-Blatt-Pfads bestimmen. Die Suche findet einen passenden Schlüssel oder folgt dem zugehörigen Kindintervall.

Die Klausuren verwenden folgende Ordnungskonvention: Bei Ordnung m besitzt die Wurzel eines nichtleeren Baums 1 bis $2m$ Schlüssel, jeder andere Knoten m bis $2m$. Andere Lehrbücher können Ordnung anders definieren. Eine rekursive Belegungsprüfung kontrolliert diese Grenzen in allen Knoten; sie beweist allein weder die Schlüsselordnung noch gleiche Blatttiefen. Die rekursive absteigende Ausgabe beginnt beim rechten Kind; danach folgen abwechselnd Schlüssel und Kinder nach links.

Beispiel: Suche im passenden Intervall. Ein Knoten enthält $[20, 50]$. Sein linkes Kind umfasst Werte unter 20, das mittlere zwischen 20 und 50, das rechte über 50. Die Suche nach 35 folgt deshalb dem mittleren Kind. Bei 50 endet sie bereits im Knoten.

Typischer Fehler: Die Bezeichnung Ordnung darf nicht ohne Prüfung zwischen verschiedenen Definitionen übertragen werden.

Frage: Wie viele Kinder hat ein innerer Knoten mit drei Schlüsseln?

Antwort: Vier Kinder, eines für jedes Werteintervall.

Vertiefung: Langfassung S. 67-68 | Themenübersicht B10

B11**Lineare, binäre und Interpolationssuche**

Lineare Suche prüft die Elemente nacheinander und benötigt im schlechtesten Fall n Schlüsselvergleiche. Binäre Suche setzt ein sortiertes Feld voraus. Sie vergleicht mit dem mittleren Element und verwirft anschließend die Hälfte, in der der Suchwert nicht liegen kann. Deshalb benötigt sie höchstens logarithmisch viele Vergleiche. Inklusive Grenzen, Abrundung des Mittelindex und ein leeres Suchintervall müssen eindeutig festgelegt sein.

Interpolationssuche schätzt die nächste Position aus dem Suchwert und den beiden Randwerten. Bei geeigneter annähernder Gleichverteilung ist ihre erwartete Suchzeit $O(\log \log n)$, im schlechtesten Fall jedoch $O(n)$. Sortierung allein genügt für die günstige Erwartung nicht. Gleiche Randwerte erfordern eine Sonderbehandlung, damit keine Division durch 0 entsteht.

Beispiel: Drei Vergleiche. Im sortierten Feld $[1, 3, 5, 7, 9, 11, 13, 15, 17]$ sucht die binäre Suche mit abgerundetem Mittelindex nach 13: zuerst 9, dann 13. Für 11 prüft sie dagegen 9, 13 und 11, also drei Werte.

Typischer Fehler: Eine vorzeitige Ganzzahldivision kann die Positionsschätzung der Interpolationssuche verfälschen.

Frage: Darf das Feld unsortiert sein?

Antwort: Bei linearer Suche ja; bei binärer und Interpolationssuche nein.

Vertiefung: Langfassung S. 69-70 | Themenübersicht B11

B12**Hash-Tabellen und Kollisionsauflösung**

Eine Hash-Funktion berechnet aus einem Schlüssel eine Tabellenadresse. Treffen verschiedene Schlüssel dieselbe Adresse, entsteht eine Kollision. Beim offenen Adressieren sucht eine festgelegte Sondierfolge einen anderen Platz. Die berechnete Hash-Adresse und die tatsächliche Speicherposition können daher verschieden sein. Einfüge- und Suchverfahren müssen dieselbe Folge verwenden; die Reihenfolge der eingefügten Schlüssel beeinflusst die Belegung.

Lineares Sondieren mit Schrittweite d erreicht alle m Plätze genau dann, wenn d und m teilerfremd sind. Quadratische Sondierung berechnet Abstände von der ursprünglichen Hash-Adresse; Vorzeichen und Quadratfolge müssen zur Aufgabenstellung passen. Bei geeigneter Hashverteilung und kontrolliertem Belegungsgrad sind konstante erwartete Suchkosten möglich. Im schlechtesten Fall wächst der Aufwand linear; eine Sondierfolge kann auch scheitern, obwohl noch unerreichbare Plätze frei sind.

Beispiel: Kollisionen mit Schrittweite 3. Für $m = 7$ und $h(k) = k \bmod 7$ werden 1, 8, 15 eingefügt. Schlüssel 1 landet auf Platz 1. Schlüssel 8 prüft 1 und 4 und bleibt auf 4. Schlüssel 15 prüft 1, 4, 0 und bleibt auf 0.

Typischer Fehler: Quadratische Abstände werden nicht wiederholt zur zuletzt besuchten Adresse addiert.

Frage: Wie hoch ist der Belegungsgrad?

Antwort: Drei von sieben Plätzen: $3/7$.

Vertiefung: Langfassung S. 71-72 | Themenübersicht B12

B13**Sortierverfahren und Vergleich ihrer Laufzeiten**

Mergesort zerlegt die Folge und vereinigt sortierte Teilfolgen. Pro Ebene entsteht lineare Arbeit; die logarithmische Ebenenzahl ergibt $O(n \log n)$ im schlechtesten Fall. Bottom-Up-Mergesort arbeitet iterativ mit immer längeren Läufen. Quicksort partitioniert um einen Pivot: Bei geeigneter zufälliger Pivotwahl ist die erwartete Laufzeit $O(n \log n)$, bei ständig extrem ungleichen Teilungen kann sie $O(n^2)$ betragen.

Heapsort baut einen Heap auf und entnimmt wiederholt das Maximum; seine schlechteste Laufzeit ist $O(n \log n)$. Bottom-Up-Heapsort verändert den Ablauf und mögliche Konstanten, nicht diese Größenordnung. Bubblesort, Insertionsort und Selectionsort brauchen im Durchschnitt und schlechtesten Fall quadratische Arbeit. Auf bereits sortierten Feldern benötigt Insertionsort jedoch nur $n - 1$ Schlüsselvergleiche. Schlüsselvergleiche sind von Schleifenprüfungen getrennt zu zählen.

Beispiel: Zwei sortierte Folgen vereinigen. Beim Mischen von $[1, 4]$ und $[2, 3]$ werden 1 mit 2, dann 4 mit 2 und schließlich 4 mit 3 verglichen. Danach wird die verbleibende 4 angehängt: $[1, 2, 3, 4]$, insgesamt drei Schlüsselvergleiche.

Typischer Fehler: Eine günstige durchschnittliche Laufzeit ist keine Garantie für jede Eingabe.

Frage: Kann Mergesort ohne Rekursion arbeiten?

Antwort: Ja, als Bottom-Up-Mergesort.

Vertiefung: Langfassung S. 73-74 | Themenübersicht B13

B14**Cocktail-Sort und Gnome-Sort schrittweise ausführen**

Cocktail-Sort vergleicht benachbarte Elemente abwechselnd vorwärts und rückwärts. Bei aufsteigender Sortierung wandert im Vorwärtslauf ein Maximum nach rechts, im Rückwärtslauf ein Minimum nach links. Der noch aktive Bereich wird dadurch kleiner. Ein Tauschkennzeichen erlaubt den Abbruch, wenn ein vollständiger maßgeblicher Durchlauf keine Änderung erzeugt. Für Ablaufspuren werden Feld, Laufrichtung und Grenzen nach den vorgeschriebenen Schritten festgehalten.

Gnome-Sort geht bei einem geordneten Nachbarpaar einen Schritt vorwärts. Bei falscher Reihenfolge tauscht es die Elemente und geht zurück; am Feldanfang muss es wieder vorwärtsgehen. Beide Verfahren benötigen im schlechtesten Fall $O(n^2)$ Arbeit. Der Grund sind viele Inversionen, also falsch geordnete Paare: Jeder strikte Nachbartausch beseitigt genau eine Inversion. Auf geordneten Folgen laufen passende Standardvarianten linear.

Beispiel: Gnome-Sort nachvollziehen. Aus $[3, 1, 2]$ entsteht durch Tausch von 3 und 1 zunächst $[1, 3, 2]$. Nach erneutem Vorwärtsgehen werden 3 und 2 getauscht: $[1, 2, 3]$. Es sind zwei Tausche nötig; die restlichen Prüfungen bestätigen die Ordnung.

Typischer Fehler: Eine Schleifeniteration ohne Tausch darf in einer verlangten vollständigen Spur nicht fehlen.

Frage: Warum sind zwei Tausche notwendig?

Antwort: Anfangs sind genau die Paare $(3, 1)$ und $(3, 2)$ invertiert.

Vertiefung: Langfassung S. 75-76 | Themenübersicht B14

B15**Graphdarstellung, Veränderungen und Zusammenhang**

Ein Graph besteht aus Knoten und Kanten, die gerichtet oder ungerichtet und zusätzlich gewichtet sein können. In der Adjazenzmatrix bezeichnet die Zeile den Ausgangs-, die Spalte den Zielknoten. Eine Adjazenzzliste speichert die Nachbarn eines Knotens. V bezeichnet die Knotenanzahl, E die Kantenanzahl. Die Matrix benötigt $O(V^2)$, die Listendarstellung $O(V + E)$ Speicher. Bei wenigen Kanten spart die Listendarstellung daher häufig Platz.

Beim Entfernen eines Knotens verschwinden sämtliche ein- und ausgehenden Kanten. Für gerichtete Graphen bedeutet schwacher Zusammenhang Zusammenhang nach Weglassen der Richtungen; starker Zusammenhang verlangt gerichtete Wege zwischen allen Knotenpaaren in beide Richtungen. Eine symmetrische Matrix kann auch gegenläufige gerichtete Kanten darstellen: Die Interpretation gehört zum Modell. Unterschiedliche Kantengewichte dürfen beim Vereinfachen nicht willkürlich verschwinden.

Beispiel: Listen und Komponenten. Der ungerichtete Graph mit Kanten A-B und B-C hat die Listen A: B; B: A, C; C: B. Er enthält drei Knoten, zwei Kanten und vier Nachbareinträge. Eine Matrix besitzt neun Zellen. Nach dem Löschen von B bleiben A und C als zwei isolierte Komponenten übrig.

Typischer Fehler: Verkettungspfeile innerhalb einer Adjazenzzliste sind keine zusätzlichen Graphkanten.

Frage: Ist eine Kante mit Gewicht 0 möglich?

Antwort: Ja; fehlende Kanten müssen dann anders markiert werden.

Vertiefung: Langfassung S. 77-78 | Themenübersicht B15

B16**Topologische Sortierung und gerichtete Kreisfreiheit**

Eine topologische Sortierung ordnet die Knoten eines gerichteten Graphen so an, dass jede Kante von einem früheren zu einem späteren Knoten zeigt. Sie beschreibt beispielsweise zulässige Reihenfolgen von Arbeitsschritten mit Abhängigkeiten. Eine solche Reihenfolge existiert genau dann, wenn der Graph keinen gerichteten Kreis enthält. Mehrere gültige Reihenfolgen sind möglich, sobald zwischen manchen Knoten keine zwingende Abhängigkeit besteht.

Ein Konstruktionsverfahren entfernt wiederholt einen Knoten ohne eingehende Kante und anschließend seine ausgehenden Kanten. Bleiben Knoten übrig, aber keiner besitzt Eingangsgrad 0, liegt ein Kreis vor. Mit Adjazenzzlisten und gepflegten Eingangsgraden benötigt das Verfahren $O(V + E)$ Zeit. Streng wachsende Knotennummern entlang jeder Kante beweisen Kreisfreiheit: Ein Rückweg zur kleineren Ausgangsnummer wäre unmöglich.

Beispiel: Zwei erlaubte Reihenfolgen. Für A nach C und B nach C sind A, B, C sowie B, A, C gültig. Beide setzen C hinter seine Voraussetzungen. Ergänzt man C nach A, entsteht der Kreis A nach C nach A. Dann müsste A zugleich vor und hinter C stehen.

Typischer Fehler: Eine numerische oder alphabetische Sortierung ist nur gültig, wenn sie alle Kantenrichtungen respektiert.

Frage: Muss der Graph zusammenhängend sein?

Antwort: Nein; auch getrennte kreisfreie Komponenten lassen sich gemeinsam anordnen.

Vertiefung: Langfassung S. 79-80 | Themenübersicht B16

B17 Breitensuche mit Warteschlange

Breitensuche erkundet einen Graphen schichtweise vom Startknoten aus. Eine FIFO-Warteschlange sorgt dafür, dass früher entdeckte Knoten zuerst bearbeitet werden. Der Start wird markiert und eingereiht. Danach wird jeweils der vorderste Knoten entnommen; seine unbesuchten Nachbarn werden markiert und hinten angefügt. Die Markierung erfolgt bereits beim Einreihen, damit gemeinsame Nachbarn nicht mehrfach in der Warteschlange landen.

Bei vorgegebener alphabetischer Nachbarreihenfolge wird jeder Schritt entsprechend ausgeführt. Die Entfernungsebenen zählen die kleinste Zahl von Kanten vom Start. Das liefert kürzeste Wege in ungewichteten Graphen, aber nicht automatisch minimale Gewichtssummen in gewichteten Graphen. Mit Adjazenzlisten beträgt der Aufwand $O(V + E)$. Unerreichbare Knoten werden von diesem Start aus nicht besucht; für alle Komponenten muss man gegebenenfalls erneut starten.

Beispiel: Warteschlange protokollieren. Im ungerichteten Graphen A-B, A-C, B-D, C-D beginnt die Schlange mit [A]. Nach Bearbeitung von A steht [B, C], nach B [C, D], nach C [D] und nach D []. Die Besuchsfolge lautet A, B, C, D. Die Entfernungen sind 0, 1, 1 und 2.

Typischer Fehler: Erst beim Entnehmen zu markieren kann denselben Knoten mehrfach einreihen.

Frage: Warum erscheint D nur einmal?

Antwort: B markiert D bereits vor der Bearbeitung von C.

Vertiefung: Langfassung S. 81-82 | Themenübersicht B17

B18 Dijkstra und Rekonstruktion kürzester Wege

Dijkstra bestimmt kürzeste Wege von einem Startknoten in einem Graphen mit nichtnegativen Kantengewichten. Anfangs gilt: Startdistanz 0, sonst unendlich; Vorgänger fehlen. Wiederholt wird der noch nicht festgelegte Knoten mit kleinster vorläufiger Distanz ausgewählt. Für jede ausgehende Kante wird geprüft, ob der Weg über diesen Knoten kürzer ist. Falls ja, werden Distanz und Vorgänger aktualisiert.

Die Nichtnegativität begründet, warum eine endgültig ausgewählte Distanz später nicht mehr kleiner werden kann. Die Vorgängerkette rekonstruiert den Weg rückwärts; anschließend wird die Reihenfolge umgekehrt. Nicht erreichbare Knoten behalten unendliche Distanz. Eine einfache Umsetzung benötigt $O(V^2)$ Zeit, eine Umsetzung mit Adjazenzlisten und binärem Heap $O((V + E) \log V)$. Gleichstände können verschiedene gleich kurze Wege ergeben.

Beispiel: Distanz und Vorgänger. Gerichtete Kanten seien A nach B mit 4, A nach C mit 1, C nach B mit 2 und B nach D mit 1. Von A aus erhält C 1. Über C verbessert sich B auf 3; über B erhält D den Wert 4. Der kürzeste Weg ist A, C, B, D.

Typischer Fehler: Mit negativen Kantengewichten ist Dijkstras übliche Festlegungsregel nicht zuverlässig.

Frage: Welche Vorgänger hat D?

Antwort: Direkt B; rückwärts folgt B, C, A.

Vertiefung: Langfassung S. 83-84 | Themenübersicht B18

B19**Minimale Spannbäume, Kruskal und Union-Find**

Ein Spannbaum verbindet alle Knoten eines zusammenhängenden ungerichteten Graphen ohne Kreis und besitzt bei V Knoten genau $V - 1$ Kanten. Ein minimaler Spannbaum minimiert die Summe der Kantengewichte. Kruskal bearbeitet Kanten nach aufsteigendem Gewicht und übernimmt eine Kante genau dann, wenn sie bisher getrennte Komponenten verbindet. Eine Verbindung innerhalb derselben Komponente würde einen Kreis erzeugen und wird verworfen.

Union-Find verwaltet diese Komponenten: find liefert einen Repräsentanten, union vereinigt zwei Mengen. Vereinigung nach Größe oder Rang und Pfadkompression beschleunigen die Verwaltung. Verschiedene Gleichstandsregeln können unterschiedliche Elternzeiger erzeugen, obwohl die Komponenten identisch sind. Mit effizientem Union-Find dominiert gewöhnlich das Kantensortieren mit $O(E \log E)$. Ist der Graph nicht zusammenhängend, entsteht statt eines Spannbauums ein minimaler Spannwald.

Beispiel: Kruskal mit einer verworfenen Kante. Gegeben sind A-B: 1, B-C: 2, A-C: 3, C-D: 4 und B-D: 5. Kruskal übernimmt A-B und B-C, verwirft A-C wegen des Kreises und übernimmt C-D. Der Spannbaum besitzt drei Kanten und Gesamtgewicht $1 + 2 + 4 = 7$.

Typischer Fehler: Der interne Union-Find-Baum ist nicht die Menge der ausgewählten Graphkanten.

Frage: Warum reichen drei Kanten?

Antwort: Vier verbundene Knoten benötigen im Baum genau vier minus eins Kanten.

Vertiefung: Langfassung S. 85-86 | Themenübersicht B19

Das passende Verfahren wählen

Aufgabe	Passendes Werkzeug
Objekte geordnet abarbeiten	FIFO: Warteschlange; LIFO: Stapel
In sortiertem Array suchen	Binäre Suche: $O(\log n)$ im Worst Case
Kürzeste Wege nach Kantenzahl	Breitensuche: ungewichtete / gleich gewichtete Kanten
Kleinste Summe von Weggewichten	Dijkstra: nur nichtnegative Kantengewichte
Alle Knoten günstig verbinden	Kruskal: minimaler Spannbaum, ungerichteter Graph
Abhängigkeiten in Reihenfolge bringen	Topologische Sortierung: gerichteter Graph ohne Kreis

Materialgrundlage und Verwendung

Grundlage sind die 16 Aufgabenblätter und ihre deutschen Lösungen: Einführung in die Programmierung, Kennungen 09, 10, 11, 12, 15, 16, 17 und 20; Algorithmen und Datenstrukturen, Kennungen 38-45. Die Langfassung mit 90 Seiten und die Themenübersicht enthalten ausführliche Erklärungen und die Zuordnung zu den Klausuraufgaben.

Beispiele in dieser Kurzfassung dienen der Erklärung und sind teilweise vereinfacht. Bei der Bearbeitung einer Klausur gelten deren genaue Vorbedingungen, Schnittstellen und Zählkonventionen. n bezeichnet die Anzahl der Elemente.