

Lösungen: Algorithmen und Datenstrukturen - 02.08.2023

Ausgangsdatei: 40-Klausur vom 02.08.2023_korr.pdf

Die Lösungen umfassen jede Teilaufgabe der sechs Aufgaben. Der Programmcode nutzt außer den vorgegebenen Klassen und Sprachmitteln nur Math. Die auf dem Deckblatt genannten Bonuspunkte haben in der vorliegenden PDF keine eigene Aufgabenstellung.

Aufgabe 1 - Wissen und Multiple Choice

Die Nummerierung der Antwortmöglichkeiten folgt jeweils ihrer Reihenfolge in der Vorlage. Bei den Vergleichen „schneller“ und „gleich schnell“ ist die asymptotische Wachstumsklasse gemeint, nicht die gemessene Laufzeit einer bestimmten Implementierung.

a) Begriff Algorithmus

Anzukreuzen sind die Aussagen 1 und 5: Ein Algorithmus löst eine Problemklasse und liefert für jede zulässige Eingabe nach endlich vielen Schritten ein Ergebnis.

Nr.	Bewertung	Begründung
1	Richtig	Allgemeinheit: Die Beschreibung gilt für eine Problemklasse.
2	Falsch	Ein Algorithmus kann unabhängig von einer konkreten Programmiersprache beschrieben werden.
3	Falsch	Die Beschreibung muss präzise und ausführbar sein, nicht vage.
4	Nicht die gesuchte Eigenschaft	Eine Zerlegung vor jeder Ausführung ist keine definierende zusätzliche Voraussetzung; elementare ausführbare Schritte gehören bereits zur Beschreibung.
5	Richtig	Dies ist die Terminierung in der hier verwendeten einführenden Definition.

b) Komplexität von Sortialgorithmen

Anzukreuzen sind 1 und 4.

Nr.	Aussage in Kurzform	Bewertung und Grund
1	Quicksort und Bubblesort im Worst Case gleich schnell	Richtig: Beide haben $\Theta(n^2)$.
2	Insertionsort im Average Case schneller als Bubblesort	Falsch als asymptotischer Vergleich: Beide haben $\Theta(n^2)$.
3	Quicksort im Average Case schneller als Heapsort	Falsch als asymptotischer Vergleich: Beide haben $\Theta(n \log n)$.

Nr.	Aussage in Kurzform	Bewertung und Grund
4	Heapsort und Mergesort im Worst Case gleich schnell	Richtig: Beide haben $\Theta(n \log n)$.
5	Bottom-Up-Heapsort im Worst Case schneller als Heapsort	Falsch als asymptotischer Vergleich: Beide haben $\Theta(n \log n)$; Konstanten können verschieden sein.
6	Bubblesort im Average Case schneller als Mergesort	Falsch: $\Theta(n^2)$ wächst stärker als $\Theta(n \log n)$.

c) Komplexität von Suchverfahren

Die erkennbar beabsichtigten zwei Antworten sind 4 und 5: Binäre Suche benötigt bei sortierten Schlüsseln im schlechtesten Fall $\Theta(\log_2 n)$, lineare Suche im schlechtesten Fall $\Theta(n)$.

Nr.	Einordnung
1	Falsch: Lineare Suche auf ungeordneten Schlüsseln benötigt im üblichen Average Case $\Theta(n)$.
2	Falsch: Ohne Sortierung ist das normale binäre Suchverfahren nicht korrekt anwendbar.
3	Nicht als allgemeine enge Schranke richtig: Für Interpolationssuche genügt bloßes Sortiertsein nicht für eine logarithmische mittlere Laufzeit; unter passenden Gleichverteilungsannahmen gilt im Mittel $\Theta(\log \log n)$.
4	Beabsichtigte richtige Antwort: Halbieren des Suchbereichs ergibt $\Theta(\log n)$.
5	Beabsichtigte richtige Antwort: Auch in einer sortierten Liste kann lineare Suche alle n Elemente betrachten müssen.
6	Keine enge Schranke: Der Worst Case der Interpolationssuche ist $\Theta(n)$, nicht $\Theta(n^2)$.

Präzisionshinweis: O bezeichnet mathematisch nur eine obere Schranke. Somit ist auch $O(n^2)$ für eine tatsächlich lineare Worst-Case-Laufzeit eine wahre, aber ungenaue Angabe. Wörtlich gelesen ist daher auch Aussage 6 wahr, sodass die Vorgabe „zwei Antworten“ nicht sauber zu strenger O -Notation passt. Die obige Auswahl verwendet die in solchen Klausurvergleichen offenbar gemeinten engen Wachstumsklassen.

d) Hash-Verfahren

Anzukreuzen sind 1 und 3.

Nr.	Bewertung	Begründung
1	Richtig	Die Hash-Funktion bildet einen Schlüssel auf eine Hash-Adresse ab.
2	Falsch	Die Adresse wird aus dem Schlüssel berechnet, nicht der Schlüssel aus einer Adresse nachgeschlagen.

Nr.	Bewertung	Begründung
3	Richtig	Quadratisches Sondieren untersucht bei Kollisionen andere Tabellenplätze.
4	Falsch	Ein „Hash-Puffer“ ist keine hier definierte Kollisionsstrategie; Sondieren erzeugt auch keinen zusätzlichen Tabellenplatz.
5	Falsch	Eine Hash-Funktion führt keine binäre Suche aus.
6	Falsch	Eine Hash-Funktion führt keine Interpolationssuche aus.

Aufgabe 2 - Komplexität und Rekursion

a) Aufrufzahlen und Laufzeiten

Für `proz1` sei $n \geq 1$, für `proz2` $n = 2^k$ mit $k \geq 0$. Wichtig ist die vorgegebene Laufzeit $O(m)$ für `tuwas(m)`: Bei rekursiven Aufrufen wird das jeweils verkleinerte Argument verwendet.

Methode	Anzahl bei $n = 4$	Allgemeine Aufrufzahl	Zeitkomplexität
<code>proz1</code>	15	$(n + 1)(n - 1) = n^2 - 1$	$O(n^3)$
<code>proz2</code>	3	$n - 1$	$O(n \log n)$

Bei `proz1` gibt es $n + 1$ äußere und jeweils $n - 1$ innere Durchläufe. Jeder der $n^2 - 1$ Aufrufe bekommt denselben Parameter n , kostet also $O(n)$. Multiplikation ergibt $O(n^3)$.

Bei `proz2` gilt für die Anzahl $A(1) = 0$ und $A(n) = 1 + 2A(n/2)$, also $A(n) = n - 1$. Für $n = 4$ wird `tuwas` mit den Argumenten 4, 2 und 2 aufgerufen. Auf Rekursionsebene j gibt es 2^j Aufrufe mit Argument $n/2^j$. Ihr Gesamtaufwand ist $O(2^j \cdot n/2^j) = O(n)$. Es gibt $\log_2 n$ Ebenen mit `tuwas`-Aufrufen; insgesamt also $O(n \log n)$. Die Basisfälle verursachen zusammen nur $O(n)$ Zusatzaufwand.

b) Maximale Rekursionstiefe für $n = 4$

Ein längster Pfad ist `proz2(4) → proz2(2) → proz2(1)`. Das sind 3 gleichzeitig aktive Methodenaufrufe beziehungsweise 2 rekursive Abstiege. Die zwei Kinder eines Aufrufs werden nacheinander ausgeführt.

c) Maximale Rekursionstiefe allgemein

Nach $k = \log_2 n$ Halbierungen ist das Argument 1. Somit sind maximal $\log_2 n + 1$ Methodenaufrufe gleichzeitig aktiv; zählt man nur die Abstiege, ist die Tiefe $\log_2 n$. Der Rekursionsspeicher ist $O(\log n)$.

Aufgabe 3 - Warteschlange

a) dequeue

```
public Person dequeue() {
    if (anfang == null) {
        return null;
    }
    Person ergebnis = anfang.daten;
    anfang = anfang.naechster;
    return ergebnis;
}
```

Die zuerst eingefügte Person steht am Anfang und wird dort entfernt. Bei genau einem Element wird anfang zu null. Aufwand: $O(1)$.

b) enqueue ohne Duplikate

```
public void enqueue(Person p) {
    assert(p != null);
    if (anfang == null) {
        anfang = new Link<Person>(p, null);
        return;
    }
    Link<Person> aktuell = anfang;
    while (true) {
        if (aktuell.daten.equals(p)) {
            return;
        }
        if (aktuell.naechster == null) {
            aktuell.naechster = new Link<Person>(p, null);
            return;
        }
        aktuell = aktuell.naechster;
    }
}
```

Alle vorhandenen Personen werden mit der vorgegebenen equals(Person)-Methode verglichen, also nach Vorname und Name statt nach Objektidentität. Wird eine gleiche Person gefunden, bleibt die Warteschlange unverändert. Andernfalls wird am letzten Knoten angehängt. Da kein Endzeiger existiert und ohnehin alle Elemente auf Gleichheit geprüft werden müssen, beträgt die Laufzeit $O(n)$.

Aufgabe 4 - Binärer Suchbaum und AVL-Eigenschaft

a) Maximum iterativ ermitteln

```
public int maximum() {
    if (wurzel == null) {
        return -1;
    }
    Knoten aktuell = wurzel;
    while (aktuell.rechtesKind != null) {
        aktuell = aktuell.rechtesKind;
    }
    return aktuell.schluesssel;
}
```

Im Suchbaum liegen größere Schlüssel rechts. Der größte Schlüssel steht daher am Ende des äußersten rechten Pfades. -1 ist wegen der vorausgesetzten positiven Schlüssel ein eindeutiges Leersignal. Die Laufzeit ist $O(h)$. Die Vorlage nennt die Methode an einer Stelle public und an einer anderen private; für die öffentliche Schnittstelle der angegebenen Klasse wird hier public verwendet.

b) Fälle im Rekursionsschritt

Für einen nichtleeren Baum werden zunächst die Ergebnisse beider Teilbäume bestimmt.

- Ist mindestens ein Teilbaum kein AVL-Baum, also sein Rückgabewert -1, ist der gesamte Baum kein AVL-Baum.
- Sind beide Teilbäume AVL-Bäume, unterscheiden sich ihre Höhen aber um mehr als 1, ist der gesamte Baum ebenfalls kein AVL-Baum.
- Sind beide Teilbäume AVL-Bäume und ist ihre Höhendifferenz höchstens 1, ist der gesamte Baum ein AVL-Baum; seine Höhe ist $1 + \max(\text{linke Höhe, rechte Höhe})$.

Der Basisfall ist der leere Teilbaum mit Höhe 0. Dadurch hat ein Blatt Höhe 1 und -1 bleibt als Fehlersignal frei. Die Suchbaumeigenschaft ist laut Aufgabe bereits gegeben.

c) Rekursive Implementierung

```
private int istAVL(Knoten knoten) {
    if (knoten == null) {
        return 0;
    }
    int links = istAVL(knoten.linkesKind);
    if (links == -1) {
        return -1;
    }
    int rechts = istAVL(knoten.rechtesKind);
    if (rechts == -1 || Math.abs(links - rechts) > 1) {
        return -1;
    }
    return 1 + Math.max(links, rechts);
}
```

Die Ergebnisse der Teilbäume werden jeweils nur einmal berechnet. Ein Fehler wird nach oben weitergegeben; andernfalls wird die korrekte Höhe zurückgegeben. Die vorhandene öffentliche Methode `return istAVL(wurzel) >= 0;` liefert damit genau den gewünschten Wahrheitswert. Aufwand: $O(n)$, Rekursionsspeicher $O(h)$.

Aufgabe 5 - Cocktail-Sort

a) Vergleiche im besten und schlechtesten Fall

Bester Fall: $O(n)$. Bei einem bereits sortierten Feld wird kein Element vertauscht. Nach einem Vorwärts- und einem Rückwärtsdurchlauf bleibt getauscht gleich false, und die while-Schleife endet. Für $n \geq 2$ werden dabei $(n - 1) + (n - 2) = 2n - 3$ Schlüsselvergleiche ausgeführt.

Schlechtester Fall: $O(n^2)$. Die aktiven Grenzen rücken pro äußerem Durchlauf von beiden Seiten um jeweils eine Position nach innen. Es können Vergleiche in der Größenordnung $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$ nötig sein.

b) Vollständige Ablaufspur

Die senkrechten Striche begrenzen den aktiven Bereich von start bis ende. Außerhalb liegende Elemente sind bereits endgültig einsortiert. „Vor“ beziehungsweise „Rück“ bezeichnet jeweils eine einzelne Iteration der zugehörigen for-Schleife; auch Iterationen ohne Tausch sind aufgeführt. Nach jeder Änderung von start oder ende steht eine eigene Zeile.

Schritt	start	ende	Feld
Initialisierung	0	5	65 34 23 19 54 43
1. Vor, i=0	0	5	34 65 23 19 54 43
1. Vor, i=1	0	5	34 23 65 19 54 43
1. Vor, i=2	0	5	34 23 19 65 54 43
1. Vor, i=3	0	5	34 23 19 54 65 43
1. Vor, i=4	0	5	34 23 19 54 43 65
1. ende--	0	4	34 23 19 54 43 65
1. Rück, i=3	0	4	34 23 19 43 54 65
1. Rück, i=2	0	4	34 23 19 43 54 65
1. Rück, i=1	0	4	34 19 23 43 54 65
1. Rück, i=0	0	4	19 34 23 43 54 65
1. start++	1	4	19 34 23 43 54 65
2. Vor, i=1	1	4	19 23 34 43 54 65
2. Vor, i=2	1	4	19 23 34 43 54 65
2. Vor, i=3	1	4	19 23 34 43 54 65
2. ende--	1	3	19 23 34 43 54 65
2. Rück, i=2	1	3	19 23 34 43 54 65
2. Rück, i=1	1	3	19 23 34 43 54 65
2. start++	2	3	19 23 34 43 54 65
3. Vor, i=2	2	3	19 23 34 43 54 65
3. ende--	2	2	19 23 34 43 54 65
3. start++	3	2	19 23 34 43 54 65

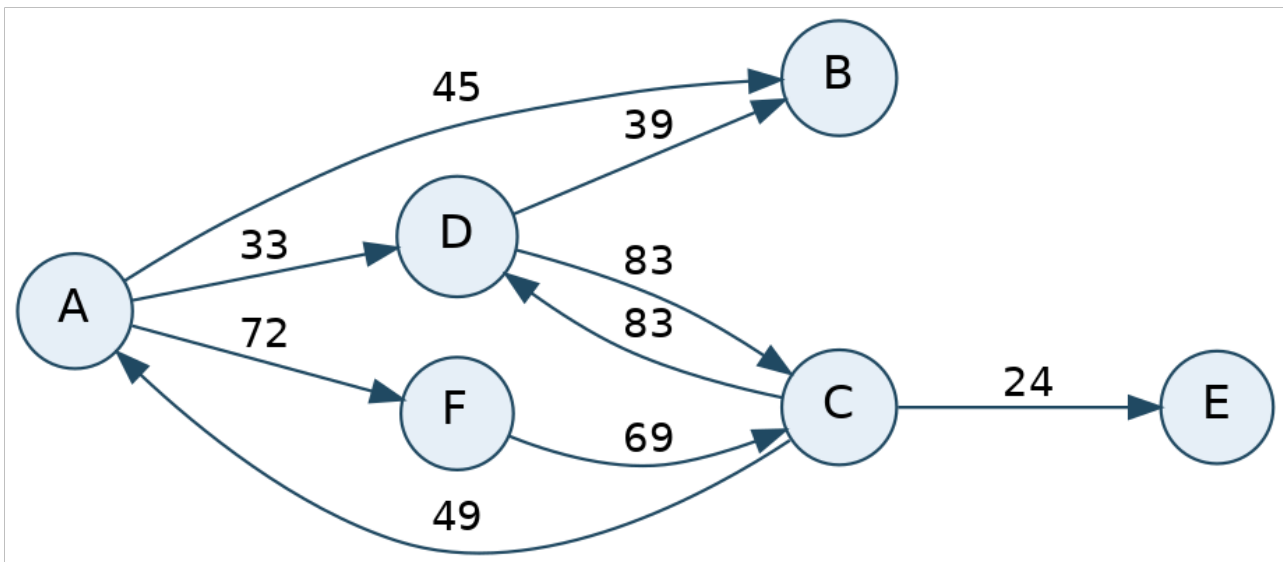
Im dritten äußeren Durchlauf hat die Rückwärtsschleife keine Iteration: Ihr Startwert wäre $i = \text{ende} - 1 = 1$, aber start ist bereits 2. Danach gilt $\text{start} > \text{ende}$ und getauscht ist false. Das Ergebnis lautet 19, 23, 34, 43, 54, 65. Insgesamt erfolgen 15 Schlüsselvergleiche und 9 Vertauschungen.

Aufgabe 6 - Graphen

a) Gerichtet oder ungerichtet; gewichtet oder ungewichtet

Der Graph ist gerichtet, weil seine Adjazenzmatrix nicht symmetrisch ist: Beispielsweise existiert $A \rightarrow B$, aber nicht $B \rightarrow A$. Er ist gewichtet; die Zahlen 45, 33, 72 usw. geben die Kantengewichte an. Leere Felder bedeuten fehlende Kanten.

b) Vollständige Zeichnung



Alle neun gerichteten Kanten sind mit ihrem Gewicht eingezeichnet. Die korrigierte rote 83 der Vorlage bei $D \rightarrow C$ ist berücksichtigt.

c) Topologische Sortierung

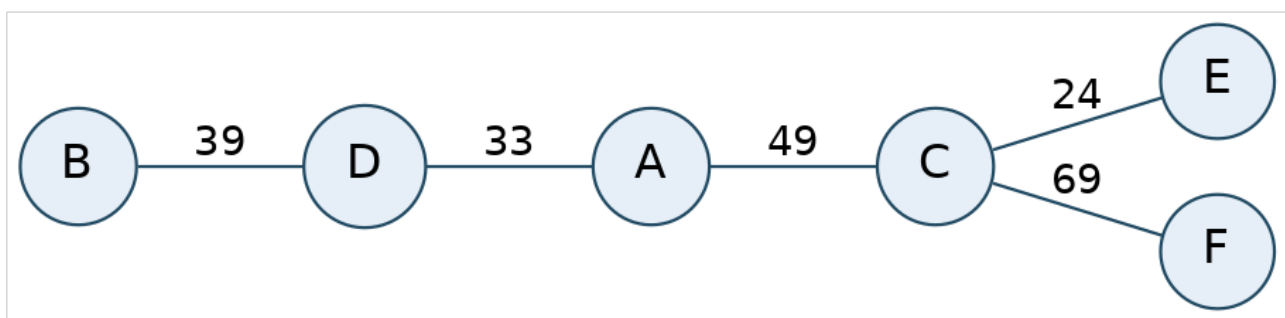
Es gibt keine topologische Sortierung, weil der gerichtete Zyklus $C \rightarrow D \rightarrow C$ existiert. Eine topologische Reihenfolge müsste zugleich C vor D und D vor C anordnen, was unmöglich ist.

d) Minimaler Spannbaum mit Kruskal

Für den zugrunde liegenden ungerichteten Graphen werden die Pfeilrichtungen entfernt; die zwei Gegenkanten zwischen C und D entsprechen derselben ungerichteten Verbindung mit Gewicht 83.

Reihenfolge	Kante	Gewicht	Entscheidung
1	C-E	24	Aufnehmen
2	A-D	33	Aufnehmen
3	B-D	39	Aufnehmen
4	A-B	45	Verwerfen: würde den Kreis A-D-B-A schließen
5	A-C	49	Aufnehmen: verbindet zwei Komponenten
6	C-F	69	Aufnehmen: bindet F an

Nun sind alle sechs Knoten durch fünf Kanten verbunden. Die schwereren Kanten A-F (72) und C-D (83) werden nicht benötigt.



Das Gesamtgewicht ist $24 + 33 + 39 + 49 + 69 = 214$. Kruskal wählt jeweils die leichteste noch zulässige Kante zwischen verschiedenen Komponenten und liefert dadurch einen minimalen Spannbaum.