

Lösungen zur Nachklausur vom 06.02.2026

Algorithmen und Datenstrukturen · SS 2025

Grundlage: 45-Klausur vom 06.02.2026_final_korr.pdf. Ausgearbeitete Lernlösung mit Begründungen; keine offizielle Musterlösung. Alle Teilaufgaben sind enthalten. Die Bonuspunkte auf dem Deckblatt sind keine zusätzliche Aufgabe. Die im Original vermerkte Korrektur zu Aufgabe 1d ist berücksichtigt.

Aufgabe 1 - Wissen und Multiple Choice

a) Listen mit Ende-Zeigern verketteten

O(1), Antwort 1. Der bisherige letzte Knoten der ersten Liste ist über ende direkt erreichbar. Sein Nachfolger wird auf den Anfang der anzuhängenden Liste gesetzt; der Ende-Zeiger wird angepasst. Auch die Sonderfälle leerer Listen brauchen nur konstant viele Zeigeroperationen. Die n bzw. m Knoten werden nicht kopiert.

b) Mögliche AVL-Bäume

Antworten 4 und 6 sind richtig: Ein perfekter Binärbaum der Höhe n in Knotenebenen besitzt $2^n - 1$ Knoten und erfüllt die AVL-Bedingung. Die Wurzel besitzt keinen Elternknoten.

Ein Binärbaum kann keine drei Kinder je Knoten besitzen. „Mindestens ein Elternknoten“ ist wegen der Wurzel falsch. Für Höhe n sind höchstens $2^n - 1$ Knoten möglich; 2^n und $2^n + 1$ überschreiten diese Obergrenze.

c) Schlüsselordnung in einem binären Suchbaum

Aussage in Originalreihenfolge	Bewertung	Begründung
Alle linken Schlüssel kleiner als alle Schlüssel auf dem Weg zur Wurzel	Falsch	Liegt der betrachtete Knoten rechts von einem Vorfahren, sind seine linken Schlüssel trotzdem größer als dieser Vorfahr.
Alle linken Schlüssel größer als alle Schlüssel auf einem beliebigen Weg zu einem Blatt	Falsch	Schon die Schlüssel im rechten Teilbaum sind größer als alle linken Schlüssel.
Minimum links kleiner als Minimum rechts	Richtig	Jeder linke Schlüssel ist kleiner als der Knotenschlüssel und jeder rechte größer.
Maximum links kleiner als Minimum rechts	Richtig	Es gilt $\max(\text{links}) < \text{Knoten} < \min(\text{rechts})$.
Maximum links größer als Minimum rechts	Falsch	Dies widerspricht genau der Suchbaumordnung.

Ein konkretes Gegenbeispiel für Aussage 1: Wurzel 10, deren rechtes Kind 20, mit linkem Kind 15 und rechtem Kind 25. Der betrachtete Knoten 20 erfüllt alle Voraussetzungen; der linke Schlüssel 15 ist aber nicht kleiner als die Wurzel 10. Für Aussage 2 genügt der Weg von 20 zu 25: 15 ist nicht größer als 25.

d) Worst-Case-Komplexität

Richtig sind Aussagen 1, 3, 4 und 6. Wie bereits im korrigierten Original ausdrücklich vermerkt, gibt es vier richtige Antworten, obwohl die Aufgabenzeile drei nennt.

Aussage	Bewertung	Begründung
Bubblesort gleich Quicksort im Worst Case	Richtig	Beide besitzen Worst-Case-Laufzeit $\Theta(n^2)$.
Bubblesort gleich Heapsort	Falsch	$\Theta(n^2)$ gegenüber $\Theta(n \log n)$.
Insertionsort gleich Selectionsort	Richtig	Beide $\Theta(n^2)$.
Preorder gleich Postorder	Richtig	Beide besuchen jeden Knoten einmal: $\Theta(n)$.
Binäre Suche: n ist der größte Schlüssel	Falsch	Die Schlüsselgröße ist nicht die Anzahl der Suchpositionen.
Binäre Suche: n ist die Anzahl der Schlüssel	Richtig	Das Suchintervall halbiert sich, daher $O(\log n)$.

Aufgabe 2 - Komplexität und Rekursion

a) Aufrufzahlen

Für $n \geq 1$ und $n = 2^k$, im üblichen Analysemodell ohne Maschinenüberlauf:

Methode	Aufrufe für $n = 4$	Aufrufe als Funktion von n	Zeitkomplexität
proz1	15	$(n + 1)(\log_2(n) + 1)$	$O(n \log n)$
proz2	4	n	$O(n)$

In proz1 läuft i von n bis einschließlich 0, also $n + 1$ Mal. Für jeden Wert von i läuft j durch 1, 2, 4, ..., n, also $\log_2(n) + 1$ Mal. Für $n = 4$ ergeben sich $5 \cdot 3 = 15$ Aufrufe.

proz2 erzeugt nur einen rekursiven Folgeaufruf und ruft tuwas bei jedem positiven Parameter einmal auf. $C(0) = 0$ und $C(n) = C(n-1) + 1$ liefern $C(n) = n$. Für $n = 0$ führen beide Prozeduren keinen tuwas-Aufruf aus; die logarithmische Formel gilt nur für $n \geq 1$.

b) tuwas(n) mit Kosten $O(n)$

$O(n^2 \log n)$. Die $(n+1)(\log_2(n)+1)$ Aufrufe erhalten alle denselben Wert n. Damit wird die Zahl der Aufrufe mit $O(n)$ multipliziert.

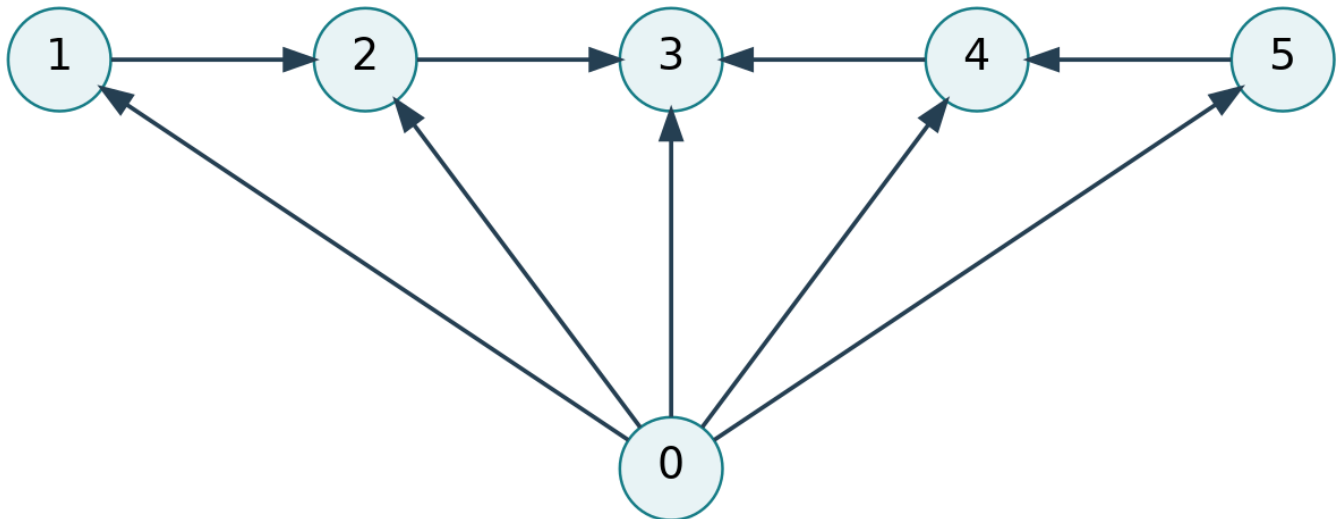
c) Maximale Rekursionstiefe

5 aktive Aufrufe: proz2(4), proz2(3), proz2(2), proz2(1), proz2(0). Bei einer Tiefendefinition über rekursive Übergänge statt aktive Aufrufe ist der Wert 4.

Aufgabe 3 - Graphen $A(n)$

Die obere Kette hat $2n - 1$ Knoten. Beide Hälften zeigen zum Knoten n : $1 \rightarrow 2 \rightarrow \dots \rightarrow n$ und $(2n-1) \rightarrow \dots \rightarrow (n+1) \rightarrow n$. Zusätzlich hat 0 zu jedem oberen Knoten einen Pfeil.

a) Graph $A(3)$



Oben steht $1 \rightarrow 2 \rightarrow 3 \leftarrow 4 \leftarrow 5$. Von 0 führen fünf weitere Pfeile nach 1, 2, 3, 4 und 5. Damit enthält $A(3)$ sechs Knoten und neun Kanten.

b) Adjazenzmatrix für $n = 3$

Zeilen und Spalten stehen in der Reihenfolge 0, 1, 2, 3, 4, 5. Eine 1 kennzeichnet einen Pfeil von der Zeile zur Spalte.

Von / nach	0	1	2	3	4	5
0	0	1	1	1	1	1
1	0	0	1	0	0	0
2	0	0	0	1	0	0
3	0	0	0	0	0	0
4	0	0	0	1	0	0
5	0	0	0	0	1	0

c) Topologische Sortierbarkeit

Ja. Beispielsweise 0, 1, 2, 5, 4, 3. Zuerst kommt 0; jede obere Teilkette wird in Pfeilrichtung abgearbeitet, und der gemeinsame Endknoten 3 kommt zuletzt. Es gibt keinen gerichteten Zyklus.

d) Kantenanzahl in $A(12)$

45 Kanten. Oben gibt es $(n-1) + (n-1) = 2n-2$ Kanten. Von 0 kommen $2n-1$ weitere hinzu. Insgesamt sind es $4n-3$, also $4 \cdot 12 - 3 = 45$.

e) Zusätzliche Kante in A(1000)

Ja, zum Beispiel $1 \rightarrow 0$. Weil $0 \rightarrow 1$ bereits existiert, entsteht der gerichtete Zyklus $0 \rightarrow 1 \rightarrow 0$. Ein Graph mit einem gerichteten Zyklus ist nicht topologisch sortierbar.

f) Knoten 60 aus A(60) entfernen

1 schwache Zusammenhangskomponente. Zwar wird die obere Kette am mittleren Knoten 60 getrennt, doch Knoten 0 bleibt erhalten und ist weiterhin mit jedem der übrigen Knoten verbunden. Unter Ignorieren der Pfeilrichtungen bleibt der gesamte Graph zusammenhängend.

g) Knoten 0, 10 und 20 aus A(120) entfernen

3 schwache Zusammenhangskomponenten: $\{1, \dots, 9\}$, $\{11, \dots, 19\}$ und $\{21, \dots, 239\}$. Ohne Knoten 0 bleibt nur die obere Kette. Das Entfernen ihrer Knoten 10 und 20 teilt sie in genau drei nichtleere Abschnitte; beim mittleren Knoten 120 ändert sich nur die Pfeilrichtung, nicht der ungerichtete Zusammenhang.

Begriffshinweis zu f und g: Bei gerichteten Graphen muss zwischen schwachem und starkem Zusammenhang unterschieden werden. Die obigen Antworten benutzen den hier naheliegenden schwachen Zusammenhang. Da alle $A(n)$ kreisfrei sind, ist jeder verbleibende Knoten eine eigene starke Komponente: 119 in f und 237 in g.

Aufgabe 4 - Liste in umgekehrter Reihenfolge kopieren

a) Am Anfang einfügen

```
private void einfuegen(T daten) {
    anfang = new Link<T>(daten, anfang);
}
```

Der neue Knoten enthält daten und zeigt auf den bisherigen Anfang. Anschließend wird er selbst zum Anfang. Bei leerer Liste ist sein Nachfolger automatisch null. Laufzeit $O(1)$.

b) Invertierte Kopie erzeugen

```
public Liste<T> invertieren() {
    Liste<T> kopie = new Liste<T>();
    for (Link<T> p = anfang; p != null; p = p.naechster) {
        kopie.einfuegen(p.daten);
    }
    return kopie;
}
```

Die Ausgangsliste wird von vorn nach hinten gelesen. Jeder neue Wert wird vorne in die Kopie eingefügt und steht damit vor allen zuvor gelesenen Werten. Aus $[a,b,c]$ wird $[c,b,a]$. Die Ausgangsliste bleibt unverändert; es entstehen neue Listenknoten. Die Datenobjekte des generischen Typs T werden als Referenzen übernommen, nicht tief kopiert. Das ist die mit dieser Schnittstelle mögliche Listenkopie. Die private Methode darf innerhalb derselben Klasse auch auf einem anderen Liste-Objekt aufgerufen werden. Laufzeit und Speicher für die Kopie jeweils $O(n)$.

Aufgabe 5 - Binärer Suchbaum

Die Konvention lautet $\text{kinder}[0]$ = linkes Kind, $\text{kinder}[1]$ = rechtes Kind.

a) Kleinsten Schlüssel iterativ bestimmen

```
public int kleinsterSchluessel() {
    assert(wurzel != null);
    Knoten p = wurzel;
    while (p.kinder[0] != null) {
        p = p.kinder[0];
    }
    return p.schluessel;
}
```

Wenn ein linkes Kind existiert, enthält der linke Teilbaum kleinere Schlüssel. Durch wiederholtes Linksgehen erreicht man den kleinsten Schlüssel des gesamten Baums. Laufzeit $O(h)$, Zusatzspeicher $O(1)$.

b) Inneren Knoten erkennen

```
private boolean istInnererKnoten(Knoten knoten) {
    assert(knoten != null);
    return knoten.kinder[0] != null
        || knoten.kinder[1] != null;
}
```

Ein innerer Knoten besitzt mindestens ein Kind. Deshalb ist ein logisches Oder erforderlich, kein Und; auch ein Knoten mit nur einem Kind zählt.

c) Innere Knoten rekursiv zählen

```
private int anzahlInnereKnoten(Knoten knoten) {
    if (knoten == null) return 0;
    int selbst = istInnererKnoten(knoten) ? 1 : 0;
    return selbst
        + anzahlInnereKnoten(knoten.kinder[0])
        + anzahlInnereKnoten(knoten.kinder[1]);
}
```

Für jeden existierenden Knoten wird sein Beitrag 0 oder 1 mit den Beiträgen beider Teilbäume addiert. Null-Teilbäume enthalten keine inneren Knoten. Jeder Knoten wird einmal besucht: Laufzeit $O(n)$, Rekursionsspeicher $O(h)$.

Aufgabe 6 - Hash-Tabelle

Die Teilaufgaben b bis d werden nacheinander auf derselben Tabelle ausgeführt. Lineares Sondieren verwendet die zyklische Folge $h_i(x) = (x \bmod 10 + i) \bmod 10$.

a) Einträge mit zwingender Kollision beim Einfügen

Nur der Eintrag 16 an Position 7 wird angekreuzt. Sein Hashwert ist $16 \bmod 10 = 6$. Um nach Position 7 verschoben zu werden, muss Position 6 beim Einfügen belegt gewesen sein. Alle anderen vorhandenen Werte stehen genau an ihrer Hashposition: $31 \rightarrow 1$, $43 \rightarrow 3$, $4 \rightarrow 4$, $6 \rightarrow 6$ und $9 \rightarrow 9$. Hier wird die übliche Entstehung der angegebenen Tabelle durch Einfügen ohne spätere Lös- oder Verschiebeoperationen zugrunde gelegt.

b) Zahl 26 einfügen

Position 8. $h(26) = 6$; Position 6 ist durch 6, Position 7 durch 16 belegt. Die erste freie Position ist 8.

c) Zusätzlich Zahl 8 einfügen

Position 0. $h(8) = 8$ ist nun durch 26 belegt, Position 9 durch 9. Nach dem zyklischen Weitergehen ist Position $(9+1) \bmod 10 = 0$ frei.

d) Kollisionswahrscheinlichkeit

8/10 = 0,8 = 80 %. Nach b und c sind acht Plätze belegt; nur Positionen 2 und 5 bleiben leer. Eine gleichverteilt gewählte Zahl aus 0 bis 9 besitzt genau ihre eigene Zahl als Hashwert. Acht der zehn möglichen Hashpositionen sind daher belegt und erzeugen beim ersten Zugriff eine Kollision. Ob der gleiche Schlüssel bereits vorkommt, ändert nichts daran, dass sein Startplatz belegt ist.

Index	0	1	2	3	4	5	6	7	8	9
Nach b und c	8	31	leer	43	4	leer	6	16	26	9

e) Ein einziger freier Platz bei Schrittweite +1

Ja, bei zyklischem linearem Sondieren und einem tatsächlich neu einzufügenden Eintrag. Für Tabellengröße M werden ausgehend von h die Plätze h , $(h+1) \bmod M$, ..., $(h+M-1) \bmod M$ geprüft. Diese M Indizes sind sämtlich verschieden: Würden zwei zusammenfallen, wäre ihre Differenz zwischen 1 und $M-1$ durch M teilbar, was unmöglich ist. Die Folge besucht somit jeden Platz genau einmal, ehe sie zum Start zurückkehrt. Der einzige freie Platz wird unabhängig vom Anfangs-hash nach spätestens M Prüfungen gefunden und belegt.