

Lösungen zur Klausur vom 14.08.2025

Algorithmen und Datenstrukturen · SS 2025

Grundlage: 44-Klausur vom 14.08.2025_final.pdf. Ausgearbeitete Lernlösung mit Begründungen; keine offizielle Musterlösung. Alle Teilaufgaben sind enthalten. Für die Bonuspunkte auf dem Deckblatt ist keine zusätzliche Aufgabe angegeben.

Aufgabe 1 - Wissen und Multiple Choice

a) Listen ohne Ende-Zeiger verketteten

O(n), Antwort 2. Die Liste mit m Elementen wird an die Liste mit n Elementen angehängt. Deren letztes Element muss zunächst durch n Knoten hindurch gesucht werden. Danach reicht eine Zeigeränderung; die m anzuhängenden Knoten müssen nicht einzeln durchlaufen werden. Gemeint ist die engste passende asymptotische Schranke, obwohl $O(n+m)$ ebenfalls eine größere obere Schranke wäre.

b) Mögliche AVL-Eigenschaften

Antworten 1 und 2: Die Wurzel kann Balancefaktor -1 besitzen. Ein AVL-Baum aus genau drei Knoten kann an allen Knoten Balancefaktor 0 haben, nämlich eine Wurzel mit zwei Blättern.

Wenn jeder Knoten Balancefaktor 0 besitzt, sind beide Teilbäume überall gleich hoch; rekursiv ergibt sich ein perfekter Binärbaum mit $2^h - 1$ Knoten. Vier oder fünf Knoten sind somit unmöglich. In jedem nichtleeren endlichen Baum gibt es Blätter mit Balancefaktor 0, daher kann nicht jeder Knoten +1 besitzen. +2 ist ohnehin mit der AVL-Bedingung unvereinbar.

c) Datenstrukturen

Aussage	Bewertung	Begründung
In Suchbäumen sind alle Kindschlüssel kleiner als die Elternschlüssel	Falsch	Rechte Teilbäume enthalten größere Schlüssel.
Comparable kann Gleichheit mit compareTo prüfen	Richtig im Sinn der Sortiergleichheit	compareTo liefert 0 bei Gleichwertigkeit gemäß der festgelegten Ordnung.
Eine einfach verkettete Liste kann mehrere null-Nachfolger besitzen	Falsch	In einer gültigen, vom Anfang erreichbaren Kette besitzt nur das letzte Element einen null-Nachfolger.
Ende-Zeiger null bedeutet leere Liste	Richtig	Bei korrekt gepflegtem Ende-Zeiger gilt dies genau für die leere Liste.
Ein Binärbaum mit 20 Knoten hat maximal Höhe 3	Falsch	Eine Kette kann Höhe 20 in Knotenebenen bzw. 19 in Kanten haben.

Präzisierung zu Comparable: `compareTo(a) == 0` bedeutet nicht zwingend `equals(a)` oder Referenzidentität. Ein korrektes Comparable kann eine Ordnung definieren, die nicht mit `equals` konsistent ist. Falls „Gleichheit“ in der Aussage ausdrücklich `equals`-Gleichheit meinen sollte, wäre die pauschale Aussage falsch. Im üblichen Sortierkontext der Klausur ist die zweite Aussage als richtig zu lesen.

d) Gleiche Worst-Case-Komplexität

Richtig sind Aussagen 1, 4 und 6.

Paar	Bewertung	Worst-Case-Laufzeiten
Insertionsort / Selectionsort	Richtig	beide $\Theta(n^2)$
Heapsort / Bubblesort	Falsch	$\Theta(n \log n)$ / $\Theta(n^2)$
Binäre Suche / Interpolationssuche	Falsch	$\Theta(\log n)$ / $\Theta(n)$
Preorder / Inorder	Richtig	beide $\Theta(n)$
Bubblesort / Mergesort	Falsch	$\Theta(n^2)$ / $\Theta(n \log n)$
Mergesort / Bottom-Up-Mergesort	Richtig	beide $\Theta(n \log n)$

Aufgabe 2 - Komplexität und Rekursion

Druckfehler im Original: Im Kopf der äußeren for-Schleife von proz1 steht ein Komma statt des zweiten Semikolons. Gemeint ist `for (int i=n; i>=1; i/=2)`. Ohne diese Korrektur kompiliert der abgedruckte Java-Code nicht. Die folgende Lösung behandelt die ersichtlich beabsichtigte Schleife.

a) Aufrufzahlen

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

Methode	Aufrufe für $n = 4$	Aufrufe als Funktion von n	Zeitkomplexität
proz1	12	$n \cdot (\log_2(n) + 1)$	$O(n \log n)$
proz2	31	$2^{(n+1)} - 1$	$O(2^n)$

In proz1 nimmt i die Werte $n, n/2, \dots, 1$ an. Für jeden dieser $\log_2(n) + 1$ Werte läuft j genau n -mal. Für $n = 4$: $3 \cdot 4 = 12$.

In proz2 steht `tuwas` außerhalb der `if`-Anweisung. Deshalb wird es auch bei $n = 0$ einmal ausgeführt. $C(0) = 1$ und $C(n) = 2C(n-1) + 1$ ergeben $C(n) = 2^{(n+1)} - 1$. Für $n = 4$ hat der Rekursionsbaum $1 + 2 + 4 + 8 + 16 = 31$ Knoten. Für $n = 0$ hat proz1 null, proz2 einen `tuwas`-Aufruf.

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

$O(n^2 \log n)$. Es gibt $n(\log_2(n) + 1)$ Aufrufe, und jeder erhält denselben Parameter n . Die zusätzlichen Kosten $O(n)$ pro Aufruf ergeben $O(n^2(\log n + 1)) = O(n^2 \log n)$.

c) Maximale Rekursionstiefe

5 gleichzeitig aktive Aufrufe, entsprechend den Parameterwerten 4, 3, 2, 1, 0. Nach der Konvention „Anzahl der Übergänge unter dem Startaufruf“ sind es 4. Die insgesamt 31 Aufrufe sind nicht gleichzeitig aktiv.

Aufgabe 3 - Gerichteter Nachfolgergraph

Die Kantenregel lautet $j - i = 1$ oder $j - i = 2$. Es gibt somit nur Pfeile zu größeren Knotennummern.

a) Adjazenzmatrix für $N = 5$

Zeile = Start, Spalte = Ziel, 1 = Kante vorhanden, 0 = keine Kante.

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

Die beiden Diagonalen direkt oberhalb der Hauptdiagonalen enthalten die Einsen.

b) Topologische Sortierbarkeit für $N = 5$

Ja. Eine topologische Reihenfolge ist 1, 2, 3, 4, 5. Jede Kante zeigt in dieser Reihenfolge nach vorn.

c) Topologische Sortierbarkeit für beliebiges N

Ja. Die Reihenfolge 1, 2, ..., N ist immer topologisch. Entlang jeder gerichteten Kante wächst die Knotennummer streng; ein gerichteter Kreis ist daher unmöglich.

d) Kante ergänzen für $N = 10000$

Ja, zum Beispiel $2 \rightarrow 1$. Die Kante $1 \rightarrow 2$ existiert bereits. Die ergänzte Gegenkante erzeugt den gerichteten Zyklus $1 \rightarrow 2 \rightarrow 1$. Damit ist der neue Graph nicht mehr topologisch sortierbar.

e) Speicherbedarf

Adjazenzlisten benötigen asymptotisch weniger Speicher: Es gibt N Knoten und $(N-1) + (N-2) = 2N-3$ Kanten, also $O(N)$ Speicher für Listen gegenüber $O(N^2)$ für die Matrix.

f) Entfernen von 59, 60 und 61 bei $N = 120$

2 schwache Zusammenhangskomponenten: $\{1, \dots, 58\}$ und $\{62, \dots, 120\}$. Innerhalb jeder Gruppe verbinden die Nachfolgerkanten alle Knoten; zwischen 58 und 62 beträgt der Abstand 4 und kann durch eine Kante mit Sprungweite höchstens 2 nicht überbrückt werden.

Bei gerichteten Graphen ist „Zusammenhangskomponente“ ohne Zusatz mehrdeutig. Hier ist der übliche Zusammenhang unter Ignorieren der Pfeilrichtungen gemeint. Würde nach starken Zusammenhangskomponenten gefragt, wäre wegen der Kreisfreiheit jeder der 117 verbliebenen Knoten eine eigene starke Komponente.

Aufgabe 4 - Jedes zweite Listenelement entfernen

a) Nachfolger entfernen

```
private void entfernen(Link<T> zeiger) {  
    if (zeiger != null && zeiger.naechster != null) {  
        zeiger.naechster = zeiger.naechster.naechster;  
    }  
}
```

Der Zeiger auf den direkten Nachfolger wird durch den Zeiger auf dessen Nachfolger ersetzt. Dadurch wird genau ein Knoten übersprungen. Ohne Nachfolger gibt es nichts zu entfernen. Der zusätzliche null-Test macht auch den Aufruf mit null wirkungslos. Laufzeit $O(1)$.

b) Liste halbieren

```
public void halbieren() {  
    Link<T> p = anfang;  
    while (p != null) {  
        entfernen(p);  
        p = p.naechster;  
    }  
}
```

Es bleiben die ursprünglichen Elemente an den Positionen 1, 3, 5, ... erhalten. Zuerst wird das zweite Element entfernt; anschließend zeigt `p.naechster` bereits auf das ursprünglich dritte Element. Deshalb wird `p` nur einmal weitergesetzt. Beispiel: `[a,b,c,d,e]` wird `[a,c,e]`. Bei leerer Liste wird die Schleife nicht betreten. Laufzeit $O(n)$, Zusatzspeicher $O(1)$.

Aufgabe 5 - Binärbaum und Heap-Schlüsseleigenschaft

Die Aufgabenstellung verwendet einen **Max-Heap**: Jeder Elternschlüssel ist mindestens so groß wie die Schlüssel seiner vorhandenen Kinder.

a) Größten Schlüssel bestimmen

```
public int groessterSchluessel() {
    assert(wurzel != null);
    assert(istHeap());
    return wurzel.schluessel;
}
```

Durch die Heapordnung ist die Wurzel mindestens so groß wie jedes Kind und damit transitiv wie jeder Nachkomme. Der Zugriff selbst kostet $O(1)$. Werden Java-Assertions eingeschaltet, führt die vorgegebene Assertion `istHeap()` zusätzlich eine vollständige Prüfung in $O(n)$ aus.

b) Lokale Heapbedingung prüfen

```
private boolean istHeapKnoten(HKnoten knoten) {
    if (knoten == null) return true;
    return (knoten.linksKind == null
        || knoten.schluessel >= knoten.linksKind.schluessel)
        && (knoten.rechtesKind == null
        || knoten.schluessel >= knoten.rechtesKind.schluessel);
}
```

Fehlende Kinder verletzen die Ordnung nicht. Bei vorhandenen Kindern darf keines größer als der Elternknoten sein. Gleich große Schlüssel sind erlaubt. Diese Prüfung kostet $O(1)$.

c) Rekursiv den gesamten Teilbaum prüfen

```
private boolean istHeap(HKnoten knoten) {
    if (knoten == null) return true;
    return istHeapKnoten(knoten)
        && istHeap(knoten.linksKind)
        && istHeap(knoten.rechtesKind);
}
```

Ein Teilbaum erfüllt die Schlüsseleigenschaft genau dann, wenn sie lokal an seiner Wurzel und rekursiv in beiden Teilbäumen gilt. Der leere Teilbaum erfüllt sie ebenfalls. Vollständigkeit und Linksbündigkeit werden auftragsgemäß nicht geprüft. Worst-Case-Laufzeit $O(n)$; Rekursionsspeicher $O(h)$.

Aufgabe 6 - Suchschritte und Insertionsort

Gezählt wird pro besuchtem Arrayelement ein Schlüsselvergleich bzw. Suchschritt. Für binäre Suche gilt $m = \lfloor (l+r)/2 \rfloor$ bei einschließlich verwendeten Grenzen. Eine andere Rundung kann ändern, welcher Schlüssel am tiefsten liegt.

1.1) Maximale Schritte bei binärer Suche

4 Schritte. Bei acht Elementen können nicht alle in den ersten drei Ebenen des Suchentscheidungsbaums liegen, denn diese enthalten höchstens $1 + 2 + 4 = 7$ Vergleichspositionen.

1.2) Schlüssel mit genau dieser Schrittzahl

Nur der Schlüssel 23, bei der genannten Abrundung. Die Suche besucht die Indizes 3, 5, 6, 7 und damit die Schlüssel 4, 21, 22, 23.

Schlüssel	Besuchte Indizes	Schritte
1	3, 1, 0	3
2	3, 1	2
3	3, 1, 2	3
4	3	1
20	3, 5, 4	3
21	3, 5	2
22	3, 5, 6	3
23	3, 5, 6, 7	4

Würde der Mittelpunkt stets aufgerundet, benötigte stattdessen der kleinste Schlüssel 1 vier Schritte. Die maximale Zahl bleibt 4.

1.3) Bedeutung von n in $O(\log_2 n)$

$n = 8$, die Anzahl der Arrayelemente. Der größte Schlüssel 23 ist für die logarithmische Abhängigkeit nicht maßgeblich.

2.1) Maximale Schritte bei linearer Suche

8 Schritte, wenn die Suche am Index 0 beginnt und nacheinander alle Elemente prüft.

2.2) Schlüssel mit genau dieser Schrittzahl

Nur 23, da er an der letzten Position steht. Dass ausschließlich vorhandene Schlüssel gesucht werden, ändert den Aufwand für diesen letzten Schlüssel nicht.

3) Vergleiche bei Insertionsort

7 Schlüsselvergleiche beim üblichen direkten Insertionsort. Für jedes der sieben Elemente ab Index 1 wird einmal mit dem Vorgänger verglichen. Da das Array bereits aufsteigend sortiert ist, endet jeder Einfügeschritt sofort, ohne Verschiebung. Schleifen- und Indexprüfungen sind hier keine Schlüsselvergleiche.