

Lösungen zur Klausur vom 31.01.2025

Algorithmen und Datenstrukturen · WS 2024/25

Grundlage: 43-Klausur vom 31.01.2025_final.pdf. Ausgearbeitete Lernlösung mit Begründungen; keine offizielle Musterlösung. Alle Teilaufgaben 1a bis 6b sind enthalten. Die Bonuspunkte im Deckblatt enthalten keine eigene Aufgabenstellung.

Aufgabe 1 - Wissen und Multiple Choice

a) Begriff Algorithmus

Aussagen 1 und 3 sind richtig: Ein Algorithmus löst eine Problemklasse und muss nach der hier verwendeten Definition für jede zulässige Eingabe mit einem Ergebnis terminieren. Er ist präzise beschrieben, aber nicht an eine konkrete Programmiersprache gebunden. Er muss nicht optimal sein; eine vage Idee reicht nicht aus. Das gesonderte Zerlegen vor der Ausführung ist keine definierende Eigenschaft.

b) Asymptotische Komplexität

Aussage	Bewertung	Begründung
Quicksort: Worst und Average $O(n \log n)$	Falsch	Durchschnittlich $O(n \log n)$, im Worst Case $O(n^2)$.
Heapsort: Worst und Average $O(n \log n)$	Richtig	Aufbau und anschließende Entnahmen bleiben in dieser Schranke.
Lineare Suche: Average $O(\log \log n)$	Falsch	Durchschnittlich sind bei üblichen gleich wahrscheinlichen Suchpositionen $\Theta(n)$ Vergleiche nötig.
Hashtabelle: Worst und Average $O(1)$	Falsch	Erwartet $O(1)$ bei guter Hashfunktion und kontrolliertem Füllgrad; Worst Case $O(n)$ durch Kollisionen.

c) Datenstrukturen

Aussage	Bewertung	Begründung
Jedes Listenelement hat genau einen Nachfolger	Falsch	Das letzte Element hat keinen Nachfolger, sein Zeiger ist null.
Jeder Binärbaumknoten hat höchstens einen Elternknoten	Richtig	Die Wurzel hat keinen, jeder andere Knoten genau einen.
Jeder Graphknoten braucht mindestens eine Kante	Falsch	Isolierte Knoten sind erlaubt.

d) Rekursiv oder iterativ

Aussagen 1 und 3 sind richtig: Heapsort lässt sich iterativ implementieren. Binäre Suche ist sowohl iterativ als auch rekursiv möglich. Auch Quicksort und Mergesort besitzen beide Varianten; Fakultät lässt sich rekursiv berechnen. Eine Preorder-Traversierung kann mit einem expliziten Stapel iterativ ausgeführt werden.

Aufgabe 2 - Komplexität und Rekursion

a) Aufrufzahlen

Für $n \geq 1$ und $n = 2^k$, im üblichen Modell ohne int-Überlauf:

Methode	Aufrufe für $n = 4$	Aufrufe als Funktion von n	Zeitkomplexität
proz1	3	$\log_2(n) + 1$	$O(\log n)$
proz2	15	$2^n - 1$	$O(2^n)$

In proz1 nimmt i die Werte 1, 2, 4, ..., n an. Es gibt deshalb $\log_2(n) + 1$ Aufrufe. Bei proz2 ruft jeder Aufruf mit positivem n genau einmal `tuwas` auf und erzeugt zwei Aufrufe für $n-1$. $C(0) = 0$, $C(n) = 2C(n-1) + 1$. Daraus folgt $C(n) = 2^n - 1$. Für $n = 4$: $1 + 2 + 4 + 8 = 15$. Beide Methoden rufen bei $n = 0$ kein `tuwas` auf.

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

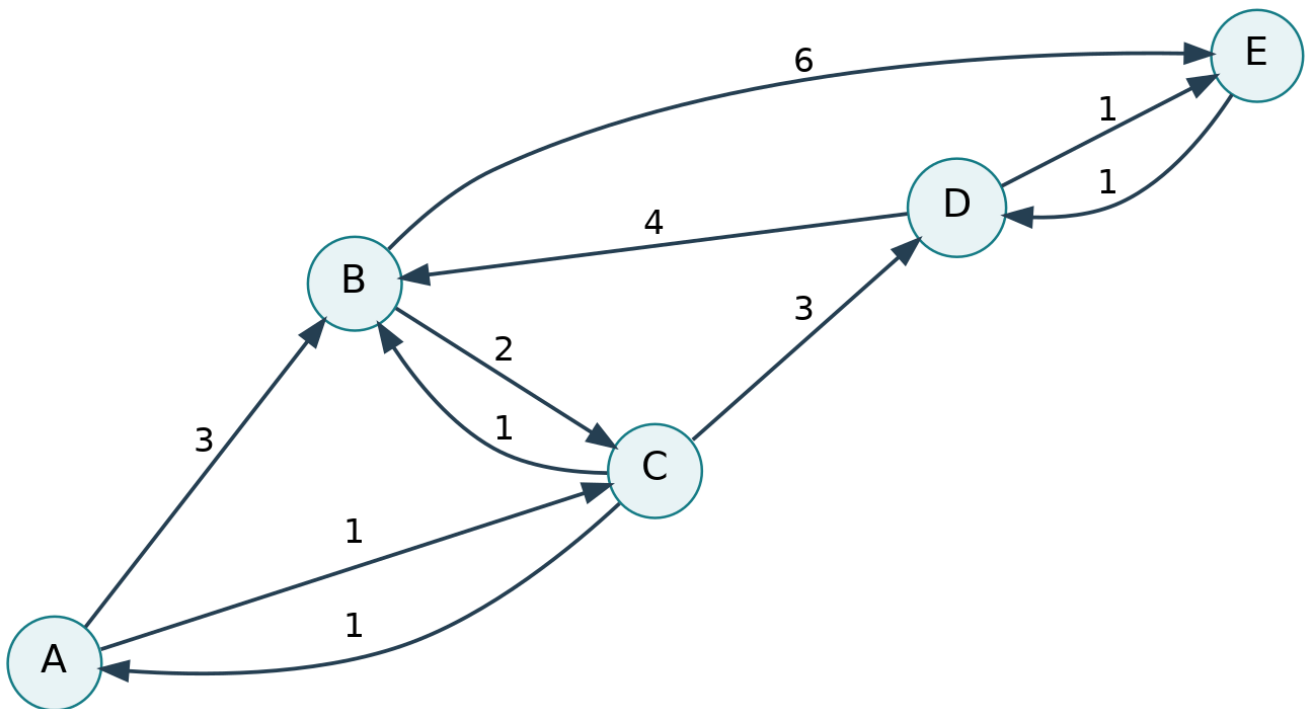
$O(n \log n)$. Der Wert n ist in allen $\log_2(n) + 1$ Schleifendurchläufen derselbe. Die Kosten $O(n)$ werden mit der Anzahl dieser Durchläufe multipliziert.

c) Maximale Rekursionstiefe

5 aktive Aufrufe auf dem Pfad $4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0$. Wer die Tiefe als Anzahl rekursiver Übergänge zählt, erhält 4. Auch der Aufruf für $n = 0$ belegt kurz einen Stapelrahmen, obwohl er `tuwas` nicht ausführt.

Aufgabe 3 - Gerichtete Graphen und Dijkstra

a) Graph aus der Matrix



Die vollständige Kantenmenge lautet: $A \rightarrow B$ (3), $A \rightarrow C$ (1), $B \rightarrow C$ (2), $B \rightarrow E$ (6), $C \rightarrow A$ (1), $C \rightarrow B$ (1), $C \rightarrow D$ (3), $D \rightarrow B$ (4), $D \rightarrow E$ (1), $E \rightarrow D$ (1). Insbesondere besitzen $B \rightarrow C$ und $C \rightarrow B$ verschiedene Gewichte.

b) Topologische Sortierung

Nein. Der Graph enthält gerichtete Zyklen, etwa $A \rightarrow C \rightarrow A$ und $D \rightarrow E \rightarrow D$. Eine topologische Sortierung müsste A vor C und gleichzeitig C vor A anordnen. Das ist unmöglich.

c) Dijkstra ab A

Notation je Zelle: Distanz / Vorgänger. ∞ bedeutet „noch nicht erreichbar bekannt“, - bedeutet „kein Vorgänger“. Jede Tabellenzeile zeigt den Zustand nach dem Markieren und dem Entspannen aller ausgehenden Kanten dieses Knotens.

Neu markiert	A	B	C	D	E
Initialisierung	0 / -	∞ / -	∞ / -	∞ / -	∞ / -
A	0 / -	3 / A	1 / A	∞ / -	∞ / -
C	0 / -	2 / C	1 / A	4 / C	∞ / -
B	0 / -	2 / C	1 / A	4 / C	8 / B
D	0 / -	2 / C	1 / A	4 / C	5 / D
E	0 / -	2 / C	1 / A	4 / C	5 / D

Zunächst verbessert C den Weg nach B von 3 auf $1 + 1 = 2$ und erreicht D mit $1 + 3 = 4$. B erreicht E zunächst mit $2 + 6 = 8$. Über D wird E auf $4 + 1 = 5$ verbessert. Alle Gewichte sind nichtnegativ, sodass Dijkstra anwendbar ist.

d) Kürzester Weg von A nach E

A → C → D → E, Länge 1 + 3 + 1 = 5. Zur Rekonstruktion folgt man von E rückwärts den Vorgängern: $E \leftarrow D \leftarrow C \leftarrow A$. Die umgekehrte Folge ist der gesuchte Weg.

Aufgabe 4 - Listenlänge und Aufteilen

a) Länge bestimmen

```
public int laenge() {  
    int n = 0;  
    for (Link<T> p = anfang; p != null; p = p.naechster) {  
        n++;  
    }  
    return n;  
}
```

Jeder Knoten wird genau einmal gezählt. Für die leere Liste bleibt $n = 0$. Laufzeit $O(n)$, Zusatzspeicher $O(1)$.

b) Liste in zwei Hälften teilen

```
public Liste<T> teilen() {  
    Liste<T> zweite = new Liste<T>();  
    int n = laenge();  
    int vorne = n / 2 + n % 2;  
    if (vorne == 0) return zweite;  
  
    Link<T> p = anfang;  
    for (int i = 1; i < vorne; i++) {  
        p = p.naechster;  
    }  
    zweite.anfang = p.naechster;  
    p.naechster = null;  
    return zweite;  
}
```

Die erste Hälfte bekommt $\lceil n/2 \rceil = n/2 + n\%2$ Elemente. Nach der Schleife zeigt p auf deren letzten Knoten. Dessen bisheriger Nachfolger wird Anfang der zweiten Liste; das Setzen auf null trennt die Ketten. Beispiele: [a,b,c,d,e] wird zu [a,b,c] und [d,e]; [a] bleibt [a] und liefert eine leere Liste zurück. Eine leere Ausgangsliste liefert ebenfalls eine leere Liste. Die Laufzeit ist $O(n)$, es werden keine Listenknoten kopiert.

Aufgabe 5 - B-Bäume

a) Iterative Höhenbestimmung

Hier wird die Höhe in **Knotenebenen** gezählt: leerer Baum 0, Wurzel ohne Kinder 1.

```
public int hoehe() {
    int h = 0;
    BKnoten k = wurzel;
    while (k != null) {
        h++;
        k = k.kinder[0];
    }
    return h;
}
```

In einem gültigen B-Baum liegen alle Blätter auf derselben Ebene. Es genügt deshalb, einen einzigen Pfad zum Blatt zu verfolgen, hier stets das erste Kind. Die Kindfelder eines Blatts sind gemäß Konstruktor null. Laufzeit $O(h)$, zusätzlicher Speicher $O(1)$. Wird Höhe stattdessen in Kanten gezählt, ist für einen nichtleeren Baum vom Ergebnis 1 abzuziehen; für den leeren Baum muss die jeweilige Konvention gesondert festgelegt werden.

b) Ordnung m prüfen

Es gilt die Definition der Aufgabe: Wurzel 1 bis $2m$ Schlüssel, sonst m bis $2m$ Schlüssel; der leere Baum erfüllt die Bedingung. Die Ordnung m wird als positive ganze Zahl vorausgesetzt.

```
private boolean hatOrdnungM(BKnoten k, int m,
                           boolean istWurzel) {
    if (k == null) return true;
    int n = k.schluessel.length;
    int minimum = istWurzel ? 1 : m;
    if (n < minimum || n > 2L * m) return false;
    for (int i = 0; i < k.kinder.length; i++) {
        if (!hatOrdnungM(k.kinder[i], m, false)) {
            return false;
        }
    }
    return true;
}
```

Die lokale Prüfung kontrolliert die zulässige Schlüsselanzahl; die Rekursion kontrolliert alle Kinder. Für jeden rekursiven Aufruf ist `istWurzel` false. Null-Kinder sind die normalen Enden der Rekursion. $2L$ vermeidet einen unnötigen int-Überlauf bei der Obergrenze. Geprüft wird genau die verlangte Belegungsbedingung; die übrigen B-Baum-Eigenschaften werden durch die gegebene Datenstruktur vorausgesetzt.

Aufgabe 6 - Suchen und Hashing

a) Interpolationssuche nach $k = 7$

Die im Original gezeigten Klammern sind Abrundungsklammern. Es gilt $m = l + \lfloor ((k - a[l]) / (a[r] - a[l])) \cdot (r - l) \rfloor$. Der Bruch ist mathematisch auszuwerten, nicht vorher durch ganzzahlige Java-Division auf 0 zu kürzen.

Schritt	l	r	Rechnung für m	Vergleich
1	0	8	$0 + \lfloor ((7-2)/(18-2)) \cdot 8 \rfloor = 2$	$a[2] = 6 < 7$
2	3	8	$3 + \lfloor ((7-7)/(18-7)) \cdot 5 \rfloor = 3$	$a[3] = 7$: gefunden

Nach dem ersten Vergleich ist $l = m + 1 = 3$. Der gesuchte Schlüssel wird im zweiten Schritt an **Index 3** gefunden.

b) Lineares Sondieren mit Schrittweite 3

Es wird $h_i(k) = (k \bmod 7 + 3i) \bmod 7$ verwendet, beginnend mit $i = 0$.

Schlüssel	Geprüfte Positionen	Endposition
5	$5 \bmod 7 = 5$	5
12	5 belegt; $(5+3) \bmod 7 = 1$	1
20	$20 \bmod 7 = 6$	6
1	1 belegt; $(1+3) \bmod 7 = 4$	4
15	1 belegt; 4 belegt; $(1+6) \bmod 7 = 0$	0

Index	0	1	2	3	4	5	6
Inhalt	15	12	leer	leer	1	5	20

Bei jeder Kollision wird genau drei Positionen weitergegangen und am Tabellenende modulo 7 fortgesetzt. Weil 3 und 7 teilerfremd sind, kann diese Sondierfolge grundsätzlich jeden Tabellenplatz erreichen.