

Lösungen zur Klausur vom 20.08.2024

Algorithmen und Datenstrukturen · SS 2024

Grundlage: 42-Klausur vom 20.08.2024_final.pdf. Ausgearbeitete Lernlösung mit Begründungen; keine offizielle Musterlösung. Alle Teilaufgaben 1a bis 6d sind enthalten. Die Bonuspunkte im Deckblatt sind keine zusätzliche Aufgabenstellung.

Aufgabe 1 - Wissen und Multiple Choice

a) Begriff Algorithmus

Richtig sind Aussage 2 und 3: Ein Algorithmus löst eine allgemeine Problemklasse und liefert für jede zulässige Eingabe nach endlich vielen Schritten ein Ergebnis. Hier wird der in der Klausur verwendete Begriff eines terminierenden Algorithmus zugrunde gelegt.

Die übrigen Aussagen sind keine zutreffenden Charakterisierungen: Ein Algorithmus ist bereits aus eindeutig ausführbaren Schritten aufgebaut; er muss nicht vor jeder Ausführung erst zerlegt werden. Er muss weder optimal sein noch in einer bestimmten Programmiersprache formuliert sein. Eine vage, erst zu vervollständigende Idee reicht nicht aus.

b) $n/2$ Suchen in einem ungeordneten Array

Anzukreuzen: Mergesort mit binärer Suche (4) und Bottom-Up-Heapsort mit binärer Suche (6).

Sortieren kostet jeweils $O(n \log n)$, die $n/2$ binären Suchen zusammen ebenfalls $O(n \log n)$. Die Gesamtkosten sind daher $O(n \log n)$. Quicksort hat im Worst Case quadratische Sortierkosten; Bubble- und Insertionsort ebenso. Bei Heapsort mit linearer Suche kosten allein die $n/2$ Suchvorgänge im Worst Case $O(n^2)$. Diese Alternativen sind insgesamt quadratisch.

c) Datenstrukturen

Aussage	Bewertung	Begründung
Einfach verkettete Liste: maximal ein Nachfolger	Richtig	Das letzte Element hat keinen Nachfolger.
Binärbaum: bis zu zwei Teilbäume je Knoten	Richtig	Es gibt höchstens einen linken und einen rechten Teilbaum.
Heap: links kleinere, rechts größere Schlüssel	Falsch	Das beschreibt die Suchbaumordnung. Im Max-Heap sind beide Kinder höchstens so groß wie der Elternknoten.

d) AVL-Baum

Aussage 3 ist richtig: An jedem Knoten unterscheiden sich die Höhen des linken und rechten Teilbaums um höchstens 1. Nur die Wurzel zu prüfen genügt nicht. Die anderen Aussagen verwechseln AVL-Balance mit Heapordnung oder behaupten eine falsche Sortierreihenfolge bzw. keine allgemeingültige Charakterisierung der Höhe.

Aufgabe 2 - Komplexität und Rekursion

a) Aufrufzahlen

Für die Formeln gilt $n \geq 1$ und $n = 2^k$. Wie bei solchen Laufzeitanalysen üblich wird ohne Maschinenüberlauf gerechnet.

Methode	Aufrufe für $n = 4$	Aufrufe als Funktion von n	Zeitkomplexität
proz1	3	$\log_2(n) + 1$	$O(\log n)$
proz2	16	2^n	$O(2^n)$

proz1: i nimmt die Werte $n, n/2, \dots, 1$ an. Für $n = 4$ sind das 4, 2, 1. Jede dieser $\log_2(n) + 1$ Iterationen ruft `tuwas` genau einmal auf.

proz2: `tuwas` wird nur in den Blättern des Rekursionsbaums bei $n = 0$ aufgerufen. Daher gilt $C(0) = 1$ und $C(n) = 2C(n-1)$, also $C(n) = 2^n$. Der gesamte Rekursionsbaum enthält $2^{(n+1)} - 1$ Methodenaufrufe; deshalb ist auch die Gesamtlaufzeit exponentiell. Bei `proz1(0)` gibt es 0 Aufrufe, bei `proz2(0)` einen.

b) `tuwas(n)` kostet $O(n)$

$O(n \log n)$. Der Parameter n wird nicht verändert. In jeder der $\log_2(n) + 1$ Iterationen wird somit eine Operation mit Kosten $O(n)$ ausgeführt. Die Halbierung von i halbiert nicht die Kosten von `tuwas(n)`.

c) Maximale Rekursionstiefe

5 gleichzeitig aktive Methodenaufrufe, nämlich `proz2(4)`, `proz2(3)`, `proz2(2)`, `proz2(1)`, `proz2(0)`. Zählt man nur die rekursiven Übergänge unterhalb des ersten Aufrufs, ist die Tiefe 4. Die zwei Zweige werden nacheinander abgearbeitet und vergrößern die maximale Tiefe nicht.

Aufgabe 3 - Einfach verkettete Listen

a) Referenz auf das letzte Element

```
public Link<T> ende() {  
    Link<T> p = anfang;  
    if (p == null) return null;  
    while (p.naechster != null) {  
        p = p.naechster;  
    }  
    return p;  
}
```

Der Zeiger läuft so lange weiter, bis kein Nachfolger mehr existiert. Dann zeigt er auf das letzte Element. Bei einer leeren Liste wird null zurückgegeben. Laufzeit $O(n)$, zusätzlicher Speicher $O(1)$.

b) Zweite Liste anhängen und leeren

```
public void verketten(Liste<T> zweiteListe) {  
    assert(this != zweiteListe);  
    Link<T> letztes = ende();  
    if (letztes == null) {  
        anfang = zweiteListe.anfang;  
    } else {  
        letztes.naechster = zweiteListe.anfang;  
    }  
    zweiteListe.anfang = null;  
}
```

Bei leerer erster Liste übernimmt anfang direkt die zweite Kette. Andernfalls verbindet der letzte Knoten beide Ketten. Anschließend wird nur der Listenanfang der zweiten Liste gelöscht; ihre Knoten bleiben über die erste Liste erreichbar. Auch eine leere zweite Liste funktioniert.

Vorausgesetzt sind zwei verschiedene, nicht-null Listen mit ursprünglich getrennten, zyklensfreien Ketten. Laufzeit $O(n)$ für die Länge der ersten Liste; keine Knoten werden kopiert.

Aufgabe 4 - B-Bäume

a) Iterative Suche

```
public boolean suchenIter(int wert) {
    BKnoten k = wurzel;
    while (k != null) {
        int i = 0;
        while (i < k.schluessel.length
            && wert > k.schluessel[i]) {
            i++;
        }
        if (i < k.schluessel.length
            && wert == k.schluessel[i]) {
            return true;
        }
        k = k.kinder[i];
    }
    return false;
}
```

Innerhalb jedes Knotens wird der erste Schlüssel gesucht, der mindestens wert ist. Bei Gleichheit ist die Suche erfolgreich. Sonst enthält nur kinder[i] den passenden Wertebereich. Ist wert größer als alle Schlüssel, ist i genau die Schlüssellänge und bezeichnet das letzte Kind. Ein null-Kind bedeutet erfolglose Suche. Unterschiedliche Schlüssellängen werden jeweils direkt aus dem aktuellen Knoten gelesen.

b) Schlüssel absteigend sammeln

```
private String absteigend(BKnoten k) {
    if (k == null) return "";
    int n = k.schluessel.length;
    String s = absteigend(k.kinder[n]);
    boolean hatInhalt = k.kinder[n] != null;
    for (int i = n - 1; i >= 0; i--) {
        if (hatInhalt) s += " ";
        s += k.schluessel[i];
        hatInhalt = true;
        if (k.kinder[i] != null) {
            s += " " + absteigend(k.kinder[i]);
        }
    }
    return s;
}
```

Die Reihenfolge lautet: rechtestes Kind, größter Knotenschlüssel, nächstes Kind, nächster Schlüssel usw. Dies ist die umgekehrte Inorder-Reihenfolge und liefert wegen der B-Baum-Suchordnung eine absteigende Folge. Leerzeichen werden nur zwischen nichtleere Bestandteile gesetzt; es gibt kein führendes oder abschließendes Leerzeichen. Der leere Baum liefert die leere Zeichenkette. Jeder nichtleere Teilbaum eines gültigen B-Baums enthält mindestens einen Schlüssel. Deshalb genügt für die Leerzeichensetzung die Prüfung der Kindreferenzen. Es werden nur String-Verkettungen als Sprachmittel, keine zusätzlichen Bibliotheksmethoden verwendet.

Aufgabe 5 - Suchen und Hashing

a) Binäre Suche nach 13

Verwendet wird $m = \lfloor (l + r)/2 \rfloor$ bei einschließlich geltenden Grenzen l und r .

Schritt	Bereich $[l, r]$	m	$a[m]$	Entscheidung
1	$[0, 8]$	4	10	$13 > 10: l = 5$
2	$[5, 8]$	6	15	$13 < 15: r = 5$
3	$[5, 5]$	5	12	$13 > 12: l = 6$

Jetzt ist $l = 6 > r = 5$: **13 ist nicht enthalten**. Das leere Intervall $[6, 5]$ verursacht keinen weiteren Schlüsselvergleich.

b) Quadratisches Sondieren

Die Aufgabenstellung legt die genaue Sondierfolge nicht fest. Hier wird zunächst die verbreitete Variante $h_i(k) = (k \bmod 7 + i^2) \bmod 7$, $i = 0, 1, 2, \dots$ verwendet.

Schlüssel	Geprüfte Positionen	Endposition
5	$5 \bmod 7 = 5$	5
12	$12 \bmod 7 = 5$ belegt; $(5 + 1^2) \bmod 7 = 6$	6
23	$23 \bmod 7 = 2$	2
1	$1 \bmod 7 = 1$	1
15	1 belegt; $(1 + 1^2) \bmod 7 = 2$ belegt; $(1 + 2^2) \bmod 7 = 5$ belegt; $(1 + 3^2) \bmod 7 = 3$	3

Index	0	1	2	3	4	5	6
Inhalt	leer	1	23	15	leer	5	12

Falls in der Vorlesung die alternierende Folge $0, +1^2, -1^2, +2^2, -2^2, \dots$ definiert wurde:

Nur 15 landet anders. Nach Position 1 und 2 wird $(1 - 1^2) \bmod 7 = 0$ geprüft; dort wird 15 eingefügt. Dann steht an Index 0 die 15 und Index 3 bleibt leer. Beide Ergebnisse hängen von der ausdrücklich gewählten Definition ab.

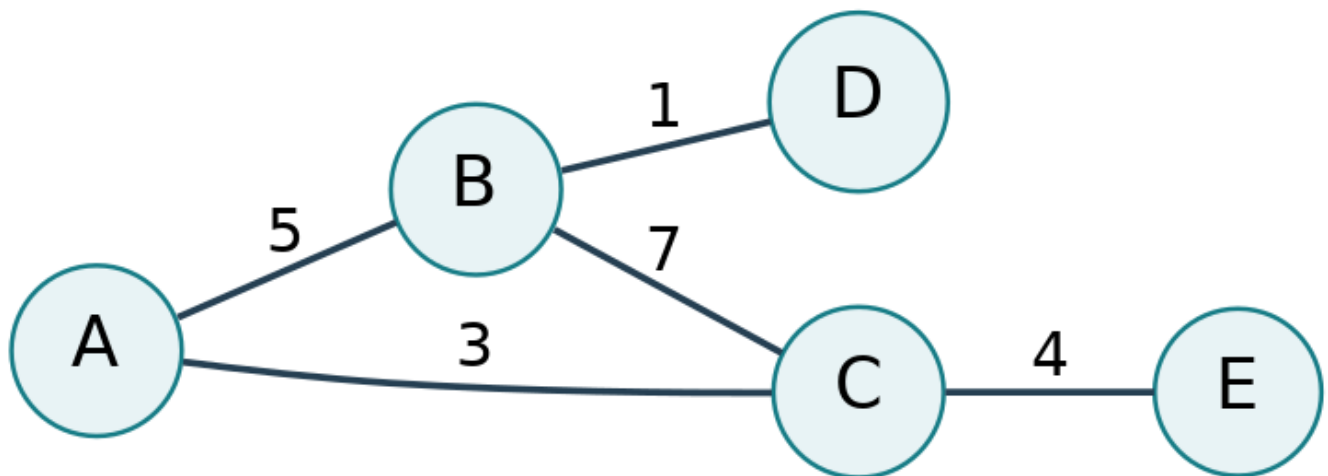
Aufgabe 6 - Graphen

a) Adjazenzmatrix

Zeilen bezeichnen den Startknoten, Spalten den Zielknoten. 0 bedeutet hier „keine Kante“; alle vorhandenen Gewichte sind positiv.

Von / nach	A	B	C	D	E
A	0	5	3	0	0
B	5	0	7	1	0
C	3	7	0	0	4
D	0	1	0	0	0
E	0	0	4	0	0

b) Graph zeichnen



Jede Linie repräsentiert zwei entgegengesetzte Pfeile mit gleichem Gewicht: A-B: 5, A-C: 3, B-C: 7, B-D: 1 und C-E: 4. Deshalb ist die ungerichtete Zeichnung gleichwertig zur gegebenen Adjazenzlistenstruktur.

c) Ungerichtete Interpretation

Ja. Die Matrix ist symmetrisch: Für alle Knoten u, v gilt $M[u,v] = M[v,u]$. Somit hat jede gerichtete Kante eine Rückkante mit demselben Gewicht.

d) Kruskal und Union-Find

Reihenfolge	Kante	Gewicht	Entscheidung
1	B-D	1	aufnehmen
2	A-C	3	aufnehmen
3	C-E	4	aufnehmen
4	A-B	5	aufnehmen
5	B-C	7	verwerfen, falls noch geprüft: erzeugt einen Kreis

Nach vier aufgenommenen Kanten sind alle fünf Knoten verbunden; Kruskal kann bereits stoppen. Der minimale Spannbaum hat die Kanten **{B-D, A-C, C-E, A-B}** und Gesamtgewicht **1 + 3 + 4 + 5 = 13**. Die Kante B-C ist die einzige ausgelassene Kante des ursprünglichen Graphen.

Eine konkrete Union-Find-Endstruktur hängt von der Vereinigungskonvention ab. Bei Vereinigung nach Größe und bei Gleichstand dem alphabetisch kleineren Repräsentanten als Wurzel entstehen nacheinander {B,D}, {A,C}, {A,C,E} und {A,B,C,D,E}. Ohne abschließende Pfadkompression:

Knoten	A	B	C	D	E
Elternzeiger	A	A	A	B	A

Die repräsentative Wurzel ist A. Nach find für alle Knoten mit Pfadkompression zeigt auch D direkt auf A. Andere korrekte Vereinigungskonventionen können eine andere Baumform, aber dieselbe gemeinsame Komponente ergeben.