

Lösungen: Algorithmen und Datenstrukturen - 08.02.2023

Ausgangsdatei: 39-Klausur vom 08.02.2023_final.pdf

Die Lösungen behandeln alle sechs Aufgaben. Rekursionstiefen werden mit beiden üblichen Zählkonventionen angegeben. Die im Aufgabentext nicht festgelegte Behandlung doppelter Suchbaumschlüssel wird bei Aufgabe 4 erläutert.

Aufgabe 1 - Wissen

a) Drei Eigenschaften eines Algorithmus

Eine passende Auswahl ist:

- **Allgemeinheit:** Der Algorithmus löst eine ganze Problemklasse und nicht nur einen einzelnen konkreten Fall.
- **Eindeutigkeit und Ausführbarkeit:** Seine endlich beschriebene Folge elementarer Schritte ist präzise festgelegt und ausführbar.
- **Terminierung:** Für jede zulässige Eingabe beendet ein terminierender Algorithmus seine Berechnung nach endlich vielen Schritten und liefert ein Ergebnis.

Diese Eigenschaften beschreiben die übliche einführende Definition, wie sie auch in den späteren Klausuren verwendet wird. Der genaue Wortlaut der erwähnten Vorlesungsdefinition liegt in der PDF nicht vor.

b) Aussagen zur O-Notation

Nr.	Aussage	Ergebnis und Begründung
1	$(n - 1)(n + 1) = O(n)$	Falsch: Der Ausdruck ist $n^2 - 1$ und wächst quadratisch.
2	$n \log n + n^2 = O(n^2)$	Richtig: Der quadratische Summand dominiert; für $n \geq 2$ gilt $\log_2 n \leq n$.
3	$n^2 + 2^n = O(n^3)$	Falsch: 2^n wächst schneller als n^3 ; der Quotient $2^n/n^3$ ist unbeschränkt.

Bei der dritten Aussage steht in der Vorlage ausdrücklich 2 hoch n , nicht 2 mal n .

c) Listenoperationen

Operation	ArrayList mit freiem Platz	LinkedList
Anhängen ans Ende	$O(1)$	$O(1)$
Zugriff auf das i -te Element, schlechtester Fall	$O(1)$	$O(n)$

Beim Array ist die Zielposition direkt berechenbar. Da noch Platz vorhanden ist, entfällt beim Anhängen eine Vergrößerung. Die Java-LinkedList besitzt einen Endzeiger, sodass Anhängen

konstanten Aufwand hat. Für einen Indexzugriff müssen dagegen Verknüpfungen durchlaufen werden. Bei einer eigenen einfach verketteten Liste ohne Endzeiger wäre Anhängen $O(n)$; das ist eine andere Implementierung als die hier benannte LinkedList.

d) Divide-and-Conquer

Zuerst wird das Problem in kleinere Teilprobleme derselben Art zerlegt. Diese werden rekursiv gelöst; ausreichend kleine Basisfälle werden unmittelbar bearbeitet. Anschließend werden die Teilergebnisse zur Lösung des ursprünglichen Problems zusammengesetzt.

e) Zwei passende Sortieralgorithmen

Mergesort und Quicksort. Mergesort teilt die Folge und vereinigt anschließend sortierte Teilfolgen. Quicksort partitioniert anhand eines Pivots und sortiert die entstandenen Teilbereiche rekursiv.

f) $O(n \log n)$ im schlechtesten Fall

Mergesort und Heapsort. Mergesort benötigt logarithmisch viele Ebenen mit insgesamt linearem Aufwand je Ebene. Heapsort führt nach dem Heapaufbau n Entnahmen beziehungsweise Wiederherstellungen der Heapordnung mit jeweils höchstens logarithmischem Aufwand aus.

Aufgabe 2 - Komplexität und Rekursion

a) Anzahl der Aufrufe und Laufzeit

Für die allgemeinen Formeln sei $n \geq 1$.

Methode	tuwas-Aufrufe für $n = 4$	Allgemeine Anzahl	Zeitkomplexität
proz1	10	$n(n + 1)/2$	$O(n^2)$
proz2	8	$2^{(n - 1)}$	$O(2^n)$

Bei proz1 hat die innere Schleife $n, n - 1, \dots, 1$ Durchläufe. Die Summe ist $n(n + 1)/2$; für $n = 4$ ergibt sich $4 + 3 + 2 + 1 = 10$.

Bei proz2 wird tuwas nur im Basisfall $n \leq 1$ aufgerufen. Für $n \geq 1$ gilt $A(1) = 1$ und $A(n) = 2A(n - 1)$, also $A(n) = 2^{(n - 1)}$. Die insgesamt $2^n - 1$ Aufrufe von proz2 ergeben dieselbe exponentielle Laufzeitklasse.

b) Rekursionstiefe für $n = 4$

Der längste Pfad lautet $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$. Daher sind maximal 4 Methodenaufrufe gleichzeitig aktiv. Werden nur rekursive Abstiege gezählt, lautet die Tiefe 3.

c) Rekursionstiefe in Abhängigkeit von n

Für $n \geq 1$ beträgt die maximale Zahl aktiver Aufrufe n beziehungsweise die Zahl rekursiver Abstiege $n - 1$. Die beiden rekursiven Aufrufe erfolgen nacheinander; sie erhöhen die Zahl der Aufrufe insgesamt, nicht die maximale Tiefe.

Aufgabe 3 - Unsortierte Prioritätsliste

a) Am Anfang einfügen

```
public void einfuegen(String aktion, int prioritaet) {  
    assert(aktion != null);  
    anfang = new Link(aktion, prioritaet, anfang);  
}
```

Der neue Knoten verweist auf den bisherigen Kopf und wird dann selbst zum Kopf. Der gleiche Ausdruck behandelt auch eine leere Liste. Die Liste ist hier absichtlich nicht nach Priorität sortiert. Aufwand: $O(1)$.

b) Erstes Element mit maximaler Priorität finden

```
private Link findeHoechstePrioritaet() {  
    Link bester = null;  
    for (Link aktuell = anfang; aktuell != null;  
         aktuell = aktuell.naechster) {  
        if (bester == null  
            || aktuell.prioritaet > bester.prioritaet) {  
            bester = aktuell;  
        }  
    }  
    return bester;  
}
```

Nach jedem Schleifendurchlauf verweist bester auf das erste maximale Element des bisher betrachteten Präfixes. Der strenge Vergleich $>$ ist entscheidend: Bei gleicher Priorität bleibt das früher gefundene Element erhalten. Bei leerer Liste bleibt bester gleich null. Auch negative Prioritäten funktionieren, weil kein willkürlicher Startwert wie 0 benutzt wird. Aufwand: $O(n)$.

c) Aktion mit maximaler Priorität entfernen

```
public String entferneAktionMitHoechsterPrioritaet() {  
    Link ziel = findeHoechstePrioritaet();  
    if (ziel == null) {  
        return null;  
    }  
    if (ziel == anfang) {  
        anfang = anfang.naechster;  
    } else {  
        Link vorgaenger = anfang;  
        while (vorgaenger.naechster != ziel) {  
            vorgaenger = vorgaenger.naechster;  
        }  
        vorgaenger.naechster = ziel.naechster;  
    }  
    return ziel.aktion;  
}
```

Die Hilfsmethode liefert genau den zu entfernenden Knoten. Beim Kopf genügt ein Verschieben von anfang; sonst wird der Vorgänger gesucht und seine Verknüpfung am Ziel vorbei auf dessen Nachfolger gesetzt. Der Referenzvergleich ist hier richtig, weil genau der gefundene Knoten entfernt werden soll. Laufzeit trotz zweier möglicher Durchläufe $O(n)$.

Aufgabe 4 - Beliebige Binärbäume

a) Minimum rekursiv bestimmen

```
private int minimum(Knoten knoten) {
    assert(knoten != null);
    int ergebnis = knoten.schluessel;
    if (knoten.linkesKind != null) {
        ergebnis = Math.min(ergebnis,
                            minimum(knoten.linkesKind));
    }
    if (knoten.rechtesKind != null) {
        ergebnis = Math.min(ergebnis,
                            minimum(knoten.rechtesKind));
    }
    return ergebnis;
}
```

Da der Baum kein Suchbaum sein muss, kann das Minimum in jedem Teilbaum liegen. Daher werden der aktuelle Schlüssel und beide Teilbaumminima berücksichtigt. Die Rekursion endet bei Blättern. Jeder Knoten wird einmal untersucht, also $O(n)$.

b) Änderung für das Maximum

minimum wird in maximum umbenannt, alle rekursiven Aufrufe werden entsprechend angepasst und Math.min wird durch Math.max ersetzt:

```
private int maximum(Knoten knoten) {
    assert(knoten != null);
    int ergebnis = knoten.schluessel;
    if (knoten.linkesKind != null) {
        ergebnis = Math.max(ergebnis,
                            maximum(knoten.linkesKind));
    }
    if (knoten.rechtesKind != null) {
        ergebnis = Math.max(ergebnis,
                            maximum(knoten.rechtesKind));
    }
    return ergebnis;
}
```

Auch das Maximum erfordert in einem beliebigen Binärbaum eine Untersuchung beider Teilbäume.

c) Suchbaumeigenschaft rekursiv überprüfen

Hier gilt die strenge Suchbaumeigenschaft: Alle Schlüssel im linken Teilbaum sind kleiner, alle im rechten größer als der aktuelle Schlüssel. Doppelte Schlüssel sind damit nicht erlaubt.

```
private boolean istSuchbaum(Knoten knoten) {  
    if (knoten == null) {  
        return true;  
    }  
    if (knoten.linksKind != null  
        && maximum(knoten.linksKind)  
        >= knoten.schluessel) {  
        return false;  
    }  
    if (knoten.rechtesKind != null  
        && minimum(knoten.rechtesKind)  
        <= knoten.schluessel) {  
        return false;  
    }  
    return istSuchbaum(knoten.linksKind)  
        && istSuchbaum(knoten.rechtesKind);  
}
```

Der Vergleich mit dem Maximum links und dem Minimum rechts prüft jeweils den gesamten Teilbaum. Ein bloßer Vergleich mit den beiden unmittelbaren Kindern wäre unzureichend. Zusätzlich müssen beide Teilbäume selbst Suchbäume sein. Der leere Baum erfüllt die Eigenschaft.

Falls die Vorlesung gleiche Schlüssel ausschließlich rechts erlaubt, wird im rechten Test $\leq$ durch $<$ ersetzt. Falls Gleichheit auf beiden Seiten zugelassen wird, werden links $\geq$ durch $>$ und rechts $\leq$ durch $<$ ersetzt. Ohne Vorgabe muss diese Konvention ausdrücklich genannt werden.

Die geforderte Verwendung von minimum und maximum führt bei einem entarteten gültigen Baum zu $O(n^2)$, weil Teilbäume mehrfach durchlaufen werden. Eine Prüfung mit fortgeführten Wertebereichen wäre $O(n)$, ist aber hier nicht die verlangte Hilfsmethodennutzung.

Aufgabe 5 - Suchen

a) Binäre Suche nach 17

Es werden inklusive Grenzen und $m = (\text{links} + \text{rechts}) / 2$ mit ganzzahliger Abrundung verwendet.

Schritt	Suchbereich	Vergleichsindex	Vergleichswert	Ergebnis
1	[0, 8]	4	10	$17 > 10$, weiter in [5, 8]
2	[5, 8]	6	15	$17 > 15$, weiter in [7, 8]
3	[7, 8]	7	16	$17 > 16$, weiter in [8, 8]
4	[8, 8]	8	18	$17 < 18$, weiter in [8, 7]

Der Bereich [8, 7] ist leer. Der Schlüssel 17 ist nicht enthalten; eine typische Suchmethode würde -1 zurückgeben. Eine andere Rundung des Mittelindex kann eine andere korrekte Vergleichsfolge erzeugen.

b) Lineares Sondieren mit Schrittweite 3

Die Sondierfunktion ist $h_i(k) = (k \bmod 11 + 3i) \bmod 11$ mit $i = 0, 1, 2, \dots$

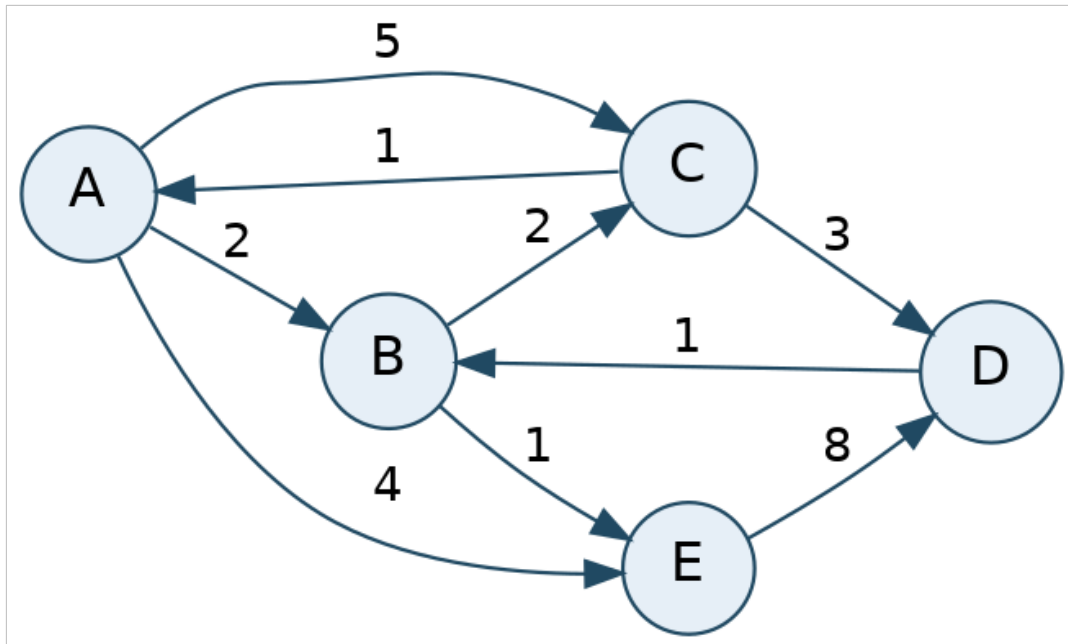
Schlüssel	Geprüfte Indizes	Endindex
4	$4 \bmod 11 = 4$	4
13	$13 \bmod 11 = 2$	2
26	4 belegt; $(4 + 3) \bmod 11 = 7$ frei	7
8	$8 \bmod 11 = 8$	8
24	2 belegt; $(2 + 3) \bmod 11 = 5$ frei	5
2	2 belegt; 5 belegt; 8 belegt; $(2 + 9) \bmod 11 = 0$ frei	0

Index	0	1	2	3	4	5	6	7	8	9	10
Inhalt	2	leer	13	leer	4	24	leer	26	8	leer	leer

Der Modulo-Schritt setzt das Sondieren am Feldanfang fort, sobald der Index das Tabellenende überschreitet.

Aufgabe 6 - Gewichteter Graph

a) Graph aus der Adjazenzlistenstruktur



Die Pfeile tragen ihre Kantengewichte. Die Verkettung zwischen Elementen einer Adjazenzliste bedeutet nur „nächster Nachbar“, keine zusätzliche Graphkante zwischen den Nachbarn.

b) Dijkstra ab A

Zellen: Entfernung/Vorgänger; ∞ = noch unerreichbar, - = kein Vorgänger. Jede Zeile gilt nach dem Markieren und Entspannen der ausgehenden Kanten.

Neu markiert	A	B	C	D	E
Initialisierung	0/-	∞ /-	∞ /-	∞ /-	∞ /-
A	0/-	2/A	5/A	∞ /-	4/A
B	0/-	2/A	4/B	∞ /-	3/B
E	0/-	2/A	4/B	11/E	3/B
C	0/-	2/A	4/B	7/C	3/B
D	0/-	2/A	4/B	7/C	3/B

Markierungsreihenfolge: A, B, E, C, D. Nach B verbessern sich C auf $2 + 2 = 4$ und E auf $2 + 1 = 3$. E liefert zunächst für D den Wert $3 + 8 = 11$; C verbessert ihn später auf $4 + 3 = 7$. Alle Gewichte sind positiv, sodass Dijkstra anwendbar ist.

c) Kürzeste Wege nach D und E

Vorgänger vom Ziel rückwärts bis A verfolgen und die Folge umkehren: Aus $D \leftarrow C \leftarrow B \leftarrow A$ wird $A \rightarrow B \rightarrow C \rightarrow D$ mit Länge $2 + 2 + 3 = 7$. Aus $E \leftarrow B \leftarrow A$ wird $A \rightarrow B \rightarrow E$ mit Länge $2 + 1 = 3$; die direkte Kante $A \rightarrow E$ wäre mit Gewicht 4 länger.