Name:

Vorname:

Matrikelnummer:

Klausur-ID:

Lösungsvorschlag

Karlsruher Institut für Technologie

Institut für Theoretische Informatik

Prof. Jörn Müller-Quade

11. April 2018

Klausur Algorithmen I

Aufgabe 1.	Kleinaufgaben	15 Punkte
Aufgabe 2.	Sortieren	7 Punkte
Aufgabe 3.	Spannbäume	12 Punkte
Aufgabe 4.	Heaps	7 Punkte
Aufgabe 5.	Hashing	9 Punkte
Aufgabe 6.	Dynamische Programmierung	10 Punkte

Bitte beachten Sie:

- Schreiben Sie auf **alle Blätter** der Klausur und Zusatzblätter Ihren Namen und Ihre Matrikelnummer.
- Schreiben Sie auf das Deckblatt Ihre Klausur-ID und nehmen Sie nach der Klausur den Zettel mit der Klausur-ID mit. Sie benötigen diese, um später online Ihre Note zu erfahren.
- Die Klausur enthält 18 Blätter. Die Bearbeitungszeit beträgt 120 Minuten.
- Die durch Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der Bestehensgrenze hinzugezählt. Die Bonuspunkte entscheiden nicht über das Bestehen der Klausur.
- Als Hilfsmittel ist ein beidseitig handbeschriebenes DIN-A4 Blatt zugelassen.
- Bitte kennzeichnen Sie deutlich, welche Aufgabe gewertet werden soll. **Bei mehreren angegebenen Möglichkeiten wird jeweils die schlechteste Alternative gewertet.**

Aufgabe	mögliche Punkte							erreichte Punkte						
	a	b	c	d	e	f	Σ	a	b	c	d	e	f	Σ
1	1	4	1	3	3	3	15							
2	2	2	3	-	-	-	7				-	-	-	
3	2	2	1	2	2	3	12							
4	1	2	2	2	-	-	7					-	-	
5	3	2	4	-	-	-	9				-	-	-	
6	1	2	5	2	-	-	10					-	-	
Σ							60							

Bonuspunkte:		Gesamtpunktzahl:	

Lösungsvorschlag

Aufgabe 1. Kleinaufgaben [15 Punkte]

Bearbeiten Sie die folgenden Aufgaben. Begründen Sie Ihre Antworten jeweils kurz.
Reine Ja/Nein-Antworten ohne Begründung geben keine Punkte.

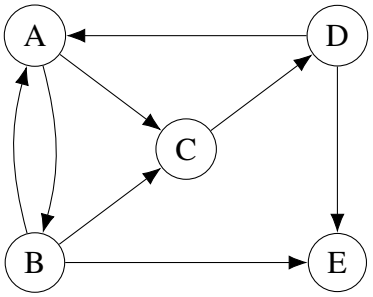
a. Gilt $2n \in o(n)$? Beweisen Sie Ihre Antwort! [1 Punkt]

Lösung

Nein, die Aussage gilt nicht, denn es gilt

$$\lim_{n\to\infty} \frac{2n}{n} = \lim_{n\to\infty} 2 = 2 \neq 0.$$

b. Im folgenden Graphen wird eine Breitensuche ausgehend vom Knoten A ausgeführt, sodass bei mehreren Möglichkeiten jeweils die Knoten in alphabetischer Reihenfolge ausgewählt werden. Geben Sie die Knoten in der Reihenfolge an, wie sie von dieser Breitensuche jeweils zum ersten Mal betrachtet werden. Geben Sie außerdem an, bei welchen Kanten es sich um *tree*-, *backward*- bzw. *cross*-Kanten handelt.



[4 Punkte]

Lösung

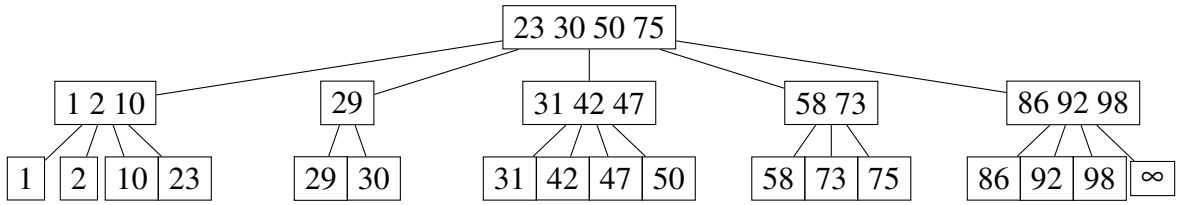
- Reihenfolge: A, B, C, E, D
- *tree*-Kanten: (A,B), (A,C), (B,E), (C,D)
- *backward*-Kanten: (B,A), (D,A)
- *cross*-Kanten: (B,C), (D,E)

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

c. Ist der folgende Baum ein (2,4)-Baum nach Definition aus der Vorlesung? Begründen Sie Ihre Antwort!



[1 Punkt]

Lösung

Nein, der Baum ist kein (2,4)-Baum, da die Wurzel 5 Kindknoten besitzt.

d. Betrachten Sie die folgende Aussage über Rekurrenzen:
Seien a, b, c, d positive Konstanten und $n \in \mathbb{N}$. Ist

$$T(n) = \begin{cases} a & \text{für } n = 1 \\ d \cdot T(\lceil n/b \rceil) + c \cdot n & \text{für } n > 1 \end{cases},$$

dann gilt

$$T(n) \in \begin{cases} \Theta(n) & \text{falls } d \leq b \\ \Theta(n^{\log_b d}) & \text{falls } d > b \end{cases}.$$

Ist diese Aussage wahr? Beweisen Sie Ihre Antwort! [3 Punkte]

Lösung

Nein, die Aussage ist nicht wahr. Man betrachte als Gegenbeispiel die folgende Rekurrenz:

$$T(n) = \begin{cases} 1 & \text{für } n = 1 \\ 2 \cdot T(\lceil n/2 \rceil) + n & \text{für } n > 1 \end{cases}$$

Nach der zu betrachtenden Aussage gilt $T(n) \in \Theta(n)$, woraus $T(n) \in \mathrm{O}(n)$ folgt. Nach dem Master-Theorem aus der Vorlesung gilt aber $T(n) \in \Theta(n \log n)$ und damit insbesondere $T(n) \in \Omega(n \log n)$, woraus $T(n) \notin \mathrm{O}(n)$ folgt. Da die Korrektheit des Master-Theorems bewiesen ist, ist dies ein Widerspruch und somit kann die Aussage nicht stimmen.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

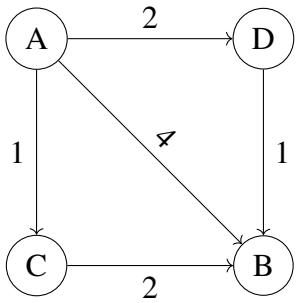
e. Es sei $G = (V, E)$ ein gerichteter und zusammenhängender Graph mit Kantengewichten aus $\mathbb{N}$. Es seien $A, B \in V$ zwei Knoten aus G und P die Menge aller Pfade von A nach B. Es sei $P_{\min}^{A,B}$ die Menge aller kürzesten Pfade von A nach B. Ein *zweitkürzester Pfad* von A nach B ist definiert als ein kürzester Pfad von A nach B aus der Menge $P \setminus P_{\min}^{A,B}$. Betrachten Sie folgenden Algorithmus:

1. Berechne Dijkstras Algorithmus in G ausgehend von A. Sei p der berechnete kürzeste Pfad von A nach B.
2. Bestimme die Kante e mit geringstem Gewicht auf p.
3. Setze $E' = E \setminus \{e\}$.
4. Berechne Dijkstras Algorithmus auf $G' = (V, E')$ ausgehend von A und gebe den hierbei berechneten kürzesten Pfad von A nach B aus.

Berechnet dieser Algorithmus immer einen zweitkürzesten Pfad zwischen A und B? Beweisen Sie Ihre Antwort! [3 Punkte]

Lösung

Nein, der Algorithmus berechnet nicht immer einen zweitkürzesten Pfad. Man betrachte dazu folgenden Graphen als Gegenbeispiel:



Die kürzesten Pfade von A nach B sind $A \to D \to B$ und $A \to C \to B$ mit Gesamtgewicht 3. Der zweitkürzeste Pfad von A nach D ist der Pfad $A \to B$ mit Gesamtgewicht 4.
Führt man Dijkstras Algorithmus aus, so wird dieser korrekt einen kürzesten Pfad von A nach B bestimmen, o.B.d.A. sei dies der Pfad $A \to C \to B$. Als nächstes enfernt der Algorithmus die Kante (A, C) aus dem Graphen und führt auf dem so entstehenden Graphen wieder Dijkstras Algorithmus aus. Dieser wird nun als Ergebnis den Pfad $A \to D \to B$ ausgeben, welcher ebenfalls ein kürzester, aber kein zweitkürzester Pfad ist.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 1

f. Es sei L eine doppelt verkette Liste mit Head-Element, deren Elemente natürliche Zahlen enthalten. Geben Sie im Pseudocode einen Algorithmus *insertMiddle(L,a)* an, der als Übergabeparameter eine Zahl $a \in \mathbb{N}$ und L erhält. Der Algorithmus soll in der Mitte von L ein neues Listenelement mit Eintrag a einfügen. Genauer sei n die Anzahl Einträge in L bevor das neue Element eingefügt wird (Head-Element nicht mitgezählt). Dann soll der Algorithmus das neue Element an Position $\lfloor n/2 \rfloor + 1$ eintragen. Die Laufzeit des Algorithmus soll in $O(n)$ liegen.

Sie können davon ausgehen, dass L einen Zeiger *head* auf das Head-Element in L bereit stellt und mindestens einen Eintrag enthält. Die Liste stellt keine anderen Funktionen oder Zeiger zu Verfügung, auch die Anzahl an Elementen in L ist nicht bekannt. Sie müssen weder die Laufzeit noch die Korrektheit ihres Algorithmus beweisen. Kommentieren Sie Ihren Algorithmus zur besseren Verständlichkeit. [3 Punkte]

Lösung

```
Function insertMiddle(L, a) :
    n := 1
    e := L.head.next
    while e ≠ L.head do                          // bestimme Anzahl Listeneinträge
        e := e.next
        n := n + 1

    middle := ⌊n/2⌋          // bestimme Position des Vorgängers des neuen Elements
    e_middle := L.head.next
    i := 1
    while i ≠ middle do
        e_middle := e_middle.next
        i := i + 1

    e_a := new Item(a)                  // erzeuge neues Listenelement mit Eintrag a
    next := e_middle.next                          // füge neues Listenelement in L ein
    e_middle.next := e_a
    next.prev := e_a
    e_a.prev := e_middle
    e_a.next := next
```

Lösungsvorschlag

Aufgabe 2. Sortieren [7 Punkte]

a. Sortieren Sie das Array $\langle 15,2,13,27,1,9,12,4\rangle$ mit dem MergeSort-Algorithmus. Verwenden Sie zur Darstellung das Schema aus der Vorlesung und geben Sie bei jedem Schritt an, ob es sich um einen split- oder merge-Schritt handelt. [2 Punkte]

Lösung

$\langle 15,2,13,27,1,9,12,4\rangle$
split
$\langle 15,2,13,27\rangle$, $\langle 1,9,12,4\rangle$
split
$\langle 15,2\rangle$, $\langle 13,27\rangle$, $\langle 1,9\rangle$, $\langle 12,4\rangle$
split
$\langle 15\rangle$, $\langle 2\rangle$, $\langle 13\rangle$, $\langle 27\rangle$, $\langle 1\rangle$, $\langle 9\rangle$, $\langle 12\rangle$, $\langle 4\rangle$
merge
$\langle 2,15\rangle$, $\langle 13,27\rangle$, $\langle 1,9\rangle$, $\langle 4,12\rangle$
merge
$\langle 2,13,15,27\rangle$, $\langle 1,4,9,12\rangle$
merge
$\langle 1,2,4,9,12,13,15,27\rangle$

b. Nennen Sie zwei Vorteile von vergleichsbasiertem Sortieren im Vergleich zu ganzzahligem Sortieren! [2 Punkte]

Lösung

Vergleichbasiertes Sortieren ...

- ... benötigt weniger Annahmen (z.B. wichtig für Algorithmenbibliotheken)
- ... ist robust gegen beliebige Eingabeverteilungen
- ... hat weniger Schwierigkeiten bzgl. Cache-Effizienz
- ... ist bei langen Schlüsseln oft schneller

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 2

c. Ein Sortieralgorithmus wird *stabil* genannt, wenn er die Reihenfolge von gleichwertigen Elementen nicht verändert, d.h. sind die Elemente A und B gleich und A steht vor dem Sortieren vor B, so wird A auch nach dem Sortieren vor B stehen.

Der QuickSort-Algorithmus aus der Vorlesung ist nicht stabil. Geben Sie eine Modifikation von QuickSort an, sodass der Algorithmus stabil ist. Hierbei soll die grundsätzliche Funktionsweise und Laufzeit von QuickSort erhalten bleiben. Ihre Modifikation muss die In Place Eigenschaft des QuickSort Algorithmus *nicht* erhalten. Begründen Sie kurz, warum Ihre Modifikation funktioniert.
Für diese Aufgabe ist kein Pseudocode notwendig, es genügt, wenn Sie Ihre Modifikation eindeutig beschreiben. [3 Punkte]

Lösung

Um den Algorithmus stabil zu machen, geben wir die In Place Eigenschaft auf. Nachdem wir ein Pivot-Element P gewählt haben, legen wir zwei neue Arrays L und R an. Sei p der Index des Pivot-Elements. Wir iterieren nun ausgehend vom Anfang über die Elemente (und überspringen dabei das Pivot-Element). Sei i der Index des aktuell betrachteten Elements und $A[i]$ dieses Element. Wir vergleichen nun jedes Element mit dem Pivot-Element und sortieren die Elemente folgendermaßen:

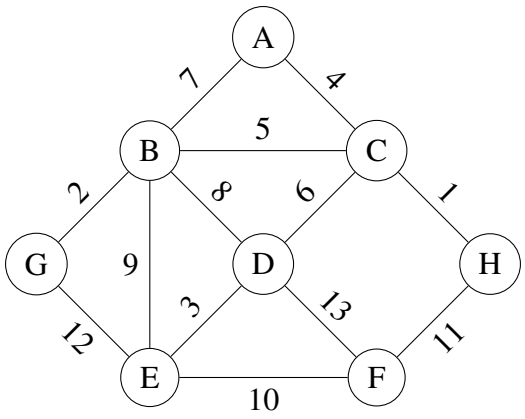
- Gilt $A[i] < P$, so fügen wir $A[i]$ ans Ende von L ein.
- Gilt $A[i] > P$, so fügen wir $A[i]$ ans Ende von R ein.
- Gilt $A[i] = P$, so unterscheiden wir zwei Fälle:
  - Ist $i < p$, so fügen wir $A[i]$ ans Ende von L ein.
  - Ist $i > p$, so fügen wir $A[i]$ ans Ende von R ein.

Nun rufen wir rekursiv QuickSort auf L und R auf und hängen danach L, P und R wieder zusammen.
Da wir von links nach rechts über das Ausgangsarray iterieren und im obigen Aufteilungsschritt die Reihenfolge von gleichen Elementen erhalten, ist der Algorithmus in dieser Form stabil (aber nicht mehr In Place).

Lösungsvorschlag

Aufgabe 3. Spannbäume [12 Punkte]

a. Berechnen Sie einen minimalen Spannbaum des unten angegebenen Graphen mit dem Algorithmus von Kruskal. Geben Sie dabei die Kanten in der Reihenfolge an, in der sie der Algorithmus von Kruskal ausgibt.



[2 Punkte]

Lösung

Kruskals Algorithmus wählt die Kanten in folgender Reihenfolge aus:
$\{C,H\},\{B,G\},\{D,E\},\{A,C\},\{B,C\},\{C,D\},\{E,F\}$

b. Geben Sie je ein Pro-Argument für den Algorithmus von Jarník-Prim und den Algorithmus von Kruskal an. [2 Punkte]

Lösung

Pro Jarník-Prim:

- Asymptotisch gut für alle $|E|,|V|$.
- Sehr schnell für $|E| \gg |V|$

Pro Kruskal:

- Gut für $|E| = O(|V|)$
- Braucht nur Kantenliste
- Profitiert von schnellen Sortierern (ganzzahlig, parallel, ...)
- Verfeinerungen auch gut für große $|E|/|V|$

(weitere Teilaufgaben auf den nächsten Blättern)

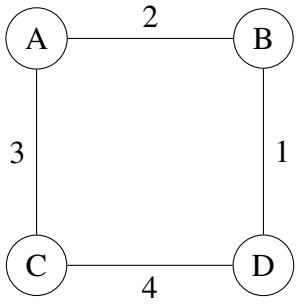
Lösungsvorschlag

Fortsetzung von Aufgabe 3

c. Ergänzen Sie den folgenden Graphen um Kantengewichte, sodass bei der Berechnung eines minimalen Spannbaums durch die Algorithmen von Kruskal und Jarník-Prim beide Algorithmen die Kanten in unterschiedlicher Reihenfolge auswählen, wenn der Algorithmus von Jarník-Prim beim Knoten A beginnt. [1 Punkt]

Lösung

Folgende Kantengewichte erfüllen das Gewünschte:



Der Algorithmus von Kruskal wählt die Kanten in der Reihenfolge $\{B,D\},\{A,B\},\{A,C\}$, der Algorithmus von Jarník-Prim wählt sie dahingegen in der Reihenfolge $\{A,B\},\{B,D\},\{A,C\}$.

Es sei im Folgenden $G=(V,E)$ ein ungerichteter und zusammenhängender Graph mit ganzzahligen Kantengewichten. Ein *maximaler* Spannbaum von G ist ein Spannbaum von G, bei dem die Summe der Kantengewichte maximal unter allen möglichen Spannbäumen von G ist.

d. Beweisen Sie: Die leichteste Kante auf einem Kreis in G wird nicht für einen maximalen Spannbaum von G benötigt. [2 Punkte]

Lösung

Es sei C ein Kreis in G und e' die leichteste Kante auf C. Angenommen es existiert ein maximaler Spannbaum T', sodass $e' \in T'$. Dann existiert mindestens eine Kante $e \in C \setminus T'$, sodass $e \notin T'$ (ansonsten wäre T' kein Baum). Wir betrachten nun $T = T' \setminus \{e'\} \cup \{e\}$.

Da T' ein Spannbaum war, ist auch T ein Spannbaum. Da e' die leichteste Kante auf C ist, ist das Kantengewicht von e größer oder gleich dem Kantengewicht von e'. Somit ist das Gesamtgewicht von T größer oder gleich dem Gesamtgewicht von T, womit folgt, dass T' kein maximaler Spannbaum war oder das T ebenfalls ein maximaler Spannbaum ist. In beiden Fällen sieht man ein, dass e' nicht für einen maximalen Spannbaum benötigt wird.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 3

e. Beweisen Sie: Für alle $S \subset V$ kann die schwerste Kante im Schnitt $C = \{\{u,v\} \in E : u \in S, v \in V \setminus S\}$ für einen maximalen Spannbaum verwendet werden. [2 Punkte]

Lösung

Es sei T' ein maximaler Spannbaum und $e \in C$ die schwerste Kante des Schnitts. Ist $e \in T'$, so ist nichts zu zeigen und wir sind fertig. Wir betrachten den Fall das $e \notin T'$. Dann gilt: $T' \cup \{e\}$ enthält einen Kreis K, da T' mindestens eine andere Kante des Schnitts enthalten muss (sonst wäre T' nicht zusammenhängend und damit kein Baum). Betrachte eine Kante $e' \in (C \cap K) \setminus \{e\}$. Dann ist $T = T' \setminus \{e'\} \cup \{e\}$ ein Spannbaum, der nicht leichter ist, da nach Voraussetzung das Gewicht von e' kleiner oder gleich dem Gewicht von e ist. Somit ist auch T ein maximaler Spannbaum.

f. Beschreiben Sie einen Algorithmus, der in Zeit $O(|E| \log |E|)$ in G einen maximalen Spannbaum berechnet. Erklären Sie kurz, wieso Ihr Algorithmus wirklich einen maximalen Spannbaum berechnet und in der gewünschten Zeit läuft. Für diese Aufgabe ist kein Pseudocode und kein formaler Beweis notwendig. [3 Punkte]

Lösung

Der Algorithmus funktioniert exakt wie Kruskals Algorithmus, mit der einzigen Änderung, dass die Kanten nach Kantengewicht absteigend sortiert betrachtet werden. Da Kruskal in Zeit $O(|E| \log |E|)$ implementiert werden kann, ist dies auch hier möglich. Dass der Algorithmus einen maximalen Spannbaum berechnet, kann mit den Eigenschaften aus den vorherigen zwei Teilaufgaben bewiesen werden, da diese im Prinzip die Schnitt- und Kreiseigenschaften für maximale Spannbäume darstellen. Damit ist der Beweis für die Korrektheit dieses Algorithmus auch analog zum Korrektheitsbeweis von Kruskals Algorithmus möglich.

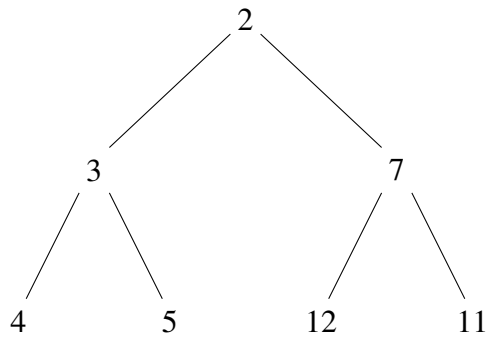
Lösungsvorschlag

Aufgabe 4. Heaps [7 Punkte]

Hinweis: In dieser Aufgabe handelt es sich bei Heaps stets um Min-Heaps.
a. Gegeben sei folgender binärer Heap in impliziter Darstellung. Zeichnen Sie den Heap in Baumdarstellung. [1 Punkt]

2	3	7	4	5	12	11

Lösung



b. Führen Sie auf folgendem Array die Methode *buildHeapBackwards* aus der Vorlesung aus, um einen binären Heap aufzubauen. Geben Sie dabei das Array nach jeder Veränderung an. Einträge, die sich nicht verändern, dürfen Sie dabei leer lassen. In diesem Fall werten wir die letzte Zahl, die in einer der darüberliegenden Spalten steht. [2 Punkte]

Lösung

10	4	6	2	5	9	3
10	4	**3**	2	5	9	**6**
10	**2**	3	**4**	5	9	6
2	**10**	3	4	5	9	6
2	**4**	3	**10**	5	9	6

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 4

c. Führen Sie auf folgendem binärem Heap die Methode *deleteMin* aus der Vorlesung aus. Geben Sie dabei das Array nach jeder Veränderung an. Einträge, die sich nicht verändern, dürfen Sie dabei leer lassen. In diesem Fall werten wir die letzte Zahl, die in einer der darüberliegenden Spalten steht. [2 Punkte]

Lösung

1	2	4	6	9	8	5	7	11	10
10	2	4	6	9	8	5	7	11	
2	**10**	4	6	9	8	5	7	11	
2	**6**	4	**10**	9	8	5	7	11	
2	6	4	**7**	9	8	5	**10**	11	

d. Geben Sie möglichst genau die Zeitkomplexität der Methoden *min*, *insert*, *deleteMin* und *buildHeap* eines binären Heaps im O-Kalkül an. [2 Punkte]

Lösung

- *min*: $O(1)$
- *insert*: $O(\log n)$
- *deleteMin*: $O(\log n)$
- *buildHeap*: $O(n)$

Lösungsvorschlag

Aufgabe 5. Hashing [9 Punkte]

a. Gegeben sei folgende Hashtabelle, welche lineare Suche zur Kollisionsauflösung verwendet:

0	1	2	3	4	5	6	7	8	9
34				24	65	44	75	18	37

Es ist bekannt, dass 18 als *erstes* eingefügt wurde und dass 65 zeitlich *vor* 24 eingefügt wurde. Als Hashfunktion wurde

$$h(x) := x \; mod \; 10$$

verwendet.

Bestimmen Sie die vollständige Reihenfolge, in welcher die Schlüssel in die Hashtabelle eingefügt wurden. Begründen Sie ihre Antwort.

[3 Punkte]

Lösung

Wir wissen, dass 18 als erstes eingefügt wurde.
Außer 24 und 65 wurde kein anderes Element an seine (durch die Hashfunktion bestimmte) Position geschrieben. Daher müssen für alle anderen Elemente Kollisionen aufgetreten sein, die durch lineare Suche aufgelöst wurden. Aus diesem Grund muss 65 als zweites eingefügt worden sein.
Das einzige Element, dass jetzt zu einer Kollision führen würde ist 75. Würde 75 als nächstes eingefügt werden, so würde es an Position 6 landen. Da dies nicht der Fall ist, muss 24 als nächstes eingefügt worden sein.
Jetzt würden 34, 44, 75 zu Kollisionen führen. Allerdings würden 34 und 75 beim Einfügen in Zelle 6 landen. Daher muss 44 als nächstes eingefügt werden.
Würde man jetzt 34 einfügen, würde das Element in Zelle 7 landen. Aus diesem Grund muss 75 als nächstes eingefügt worden sein.
Übrig bleiben 37 und 34. Würde man 34 einfügen, so würde 34 an Position 9 geschrieben werden, was falsch ist. Daher muss 37 vor 34 eingefügt werden.
Die vollständige Einfügereihenfolge ist daher: 18, 65, 24, 44, 75, 37, 34.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 5

b. Nennen Sie zwei Vorteile von Hashing mit verketteten Listen gegenüber Hashing mit linearer Suche. [2 Punkte]

Lösung

- insert in $O(1)$
- Kollisionen wirken sich nicht auf andere Buckets aus
- referentielle Integrität
- Leistungsgarantien mit universellem Hashing
- Verkettete Listen sind weniger empfindlich für "Vollaufen"

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 5

c. In einem Lebensmittelladen wurde ein automatisches Bestellungssystem installiert, das mit einer FIFO Queue arbeitet. Jedes mal, wenn ein Kunde eine Bestellung aufgibt, werden die Bestelldaten (Produkt, Menge, Name des Kunden) in die Queue eingefügt. Der Besitzer des Ladens arbeitet die einzelnen Aufträge in der Queue nacheinander ab. Um zu vermeiden, dass ein Produkt ausverkauft ist, möchte er jeder Zeit in seinem System abfragen können, wie viele Einheiten eines Produktes sich gerade in der Queue befinden.

Beschreiben Sie, wie Sie das Bestellsystem so modifiziern würden, dass folgende Operationen in erwartet $O(1)$ Zeit ausgeführt werden können.

- `enqueue`(B): Fügt eine Bestellung $B := \langle$Produkt p, Menge m, Name des Kunden$\rangle$ an das Ende der Queue ein.
- `dequeue`(): Extrahiert die aktuelle Bestellung B aus der Queue.
- `query`(p): Gibt die Menge von Produkt p zurück, die sich aktuell in der Queue befindet.

Hinweis: Es ist bekannt, dass der Laden insgesamt n verschiedene Produkte führt. Die im Bestellungssystem installierte Queue kann maximal q Bestellungen enthalten und es gilt $q < n$. Ihre Datenstrukturen dürfen $O(q)$ Platz verwenden. Für diese Aufgabe ist kein Pseudocode notwendig.

[4 Punkte]

Lösung

Ergänzend zur FIFO-Queue des Bestellsystems verwenden wir eine Hashtabelle der Größe $O(q)$. Zur Kollisionsauflösung werden verkettete Listen verwendet. In der Hashtabelle speichern wir Elemente der Form (Produkt, Gesamtmenge), wobei Produkt als Schlüssel verwendet wird.

- `enqueue`(B): Zuerst fügen wir die Bestellung B in die Queue ein. Anschließend suchen wir das Produkt in der Hashtabelle. Wenn es bereits in der Tabelle ist, addieren wir die Menge m zur Gesamtmenge. Wenn es noch nicht in der Tabelle ist, fügen wir es mit der aktuellen Menge m in die Hashtabelle ein.
- `dequeue`(): Wir extrahieren die aktuelle Bestellung aus der Queue. Anschließend suchen wir in der Hashtabelle nach dem Produkt und subtrahieren die Menge m der aktuellen Bestellung von der Gesamtmenge. Wenn die Gesamtmenge 0 ist, entfernen wir das Element aus der Hashtabelle.
- `query`(p): Wir suchen Produkt p in der Hashtabelle und geben die entsprechende Gesamtmenge oder 0 zurück, falls das Element nicht vorhanden ist.

Lösungsvorschlag

Aufgabe 6. Dynamische Programmierung [10 Punkte]

Gegeben sei eine binäre Matrix A der Größe $M \times N$, d.h., eine Matrix, die nur 0 und 1 enthält. Gesucht ist die Größe $\text{lcss}_1(A)$ der größten zusammenhängenden **quadratischen** Teilmatrix, die nur 1 enthält.
Beispielsweise ist $\text{lcss}_1(A) = 3$ für die folgende Matrix $A \in \{0,1\}^{5\times 6}$:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 1 & 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

a. Markieren Sie eine größte zusammenhängende quadratische 1er-Teilmatrix in der untenstehenden 6×8 Matrix und geben Sie deren Größe an. [1 Punkt]

Lösung

$$\begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 1 \\ 1 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 1 \\ 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 1 & 1 & 1 \\ 0 & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 1 & 1 \end{pmatrix} \qquad \text{lcss}_1(A) = 4$$

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

b. Angenommen, Sie kennen zu einer gegebenen Matrix A lediglich Position und Größe der **eindeutig bestimmten** größten zusammenhängenden quadratischen 1er-Teilmatrix, die restlichen Einträge sind Ihnen nicht bekannt. Dies sei hier beispielhaft visualisiert (ein „?“ steht dabei für ein unbekanntes Bit, die angegebenen Bits stellen die größte zusammenhängende quadratische 1er-Teilmatrix dar):

$$\begin{pmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix} \qquad \begin{pmatrix} ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & 1 & 1 & 1 & 1 & ? & ? \\ ? & ? & ? & ? & ? & ? & ? & ? & ? & ? \end{pmatrix}$$

Können Sie in beiden Fällen eindeutig die

1. Positionen
2. Größen

der jeweils größten zusammenhängenden quadratischen 1er-Teilmatrizen der grau markierten Flächen angeben, obwohl die restlichen Einträge der Matrizen nicht bekannt sind? Begründen Sie Ihre Antworten. [2 Punkte]

Lösung

1. In beiden Fällen ist die rechte untere Ecke eine mögliche Position der 1er-Teilmatrix (wie auch in den Beispielen ersichtlich), allerdings sind diese nicht eindeutig, da weitere 1er-Teilmatrizen der gleichen Größe in den unbekannten Bits auftreten können.
2. In beiden Fällen ist die Größe eindeutig $\text{lcss}_1(A) - 1 = 3$. Durch die Eindeutigkeit der größten zusammenhängenden quadratischen Teilmatrix existiert keine andere 1er-Teilmatrix der Größe $\text{lcss}_1(A) = 4$. Gleichzeitig muss die Größe mindestens 3 sein, da maximal eine Zeile und eine Spalte von der ursprünglichen Teilmatrix „abgeschnitten“ wurden und damit in beiden Matrizen quadratische Teilmatrizen der Größe 3 vorhanden sind.

(weitere Teilaufgaben auf den nächsten Blättern)

Lösungsvorschlag

Fortsetzung von Aufgabe 6

c. Entwerfen Sie nach dem Entwurfsprinzip der dynamischen Programmierung einen Algorithmus, welcher zu einer gegebenen Matrix $A \in \{0,1\}^{M \times N}$ die gesuchte Größe $\mathrm{lcss}_1(A)$ berechnet. Die Laufzeit des Algorithmus soll dabei in $\mathrm{O}(M \cdot N)$ liegen. Für diese Aufgabe ist kein Pseudocode notwendig, es genügt, wenn Sie Ihren Algorithmus eindeutig beschreiben. [5 Punkte]

Lösung

Wir definieren ein 2-dimensionales Array S der Größe $M \times N$, wobei $S_{i,j}$ der Größe der größten zusammenhängenden quadratischen 1er-Teilmatrix von A entspricht, die an Position $A_{i,j}$ endet, d.h., deren rechte untere Ecke auf $A_{i,j}$ fällt. Wir initialisieren die erste Zeile bzw. Spalte mittels

$$\begin{aligned} S_{i,1} &:= A_{i,1} && 1 \leq i \leq M \\ S_{1,j} &:= A_{1,j} && 1 \leq j \leq N \end{aligned}$$

und berechnen die übrigen Einträge mittels

$$S_{i,j} := \begin{cases} \min\left\{S_{i-1,j}, S_{i-1,j-1}, S_{i,j-1}\right\} + 1 & \text{falls } A_{i,j} = 1 \\ 0 & \text{falls } A_{i,j} = 0. \end{cases}$$

Die gesuchte Lösung ist dann der maximale Eintrag in S.

d. Analysieren Sie die Laufzeit und den Speicherverbrauch Ihres Algorithmus im O-Kalkül. [2 Punkte]

Lösung

Erstellen des Arrays benötigt Zeit $\mathrm{O}(M \cdot N)$, Initialisierung der ersten Zeile und Spalte $\mathrm{O}(M+N)$, Berechnung der restlichen Einträge jeweils $\mathrm{O}(1)$, Finden des Maximums $\mathrm{O}(M \cdot N)$; insgesamt ergibt sich also eine Laufzeit von $\mathrm{O}(M \cdot N)$. Da neben S keine weiteren zusätzlichen Datenstrukturen benötigt werden, liegt der Speicherbedarf ebenfalls in $\mathrm{O}(M \cdot N)$.