Datenstrukturen (DS)

Sommersemester 2015

Dipl-Inf. Bert Besser
Prof. Dr. Georg Schnitger
Hannes Seiwert, M.Sc.

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

Institut für Informatik
AG Theoretische Informatik

Übung 2

Ausgabe: 05.05.2015
Abgabe: 19.05.2015

In den folgenden Aufgaben müssen Sie eigene Algorithmen entwerfen. Gehen Sie dabei stets in drei Schritten vor:

1. Erklären Sie kurz die **Idee** Ihres Algorithmus.
2. Beschreiben Sie Ihren Algorithmus in **Pseudocode**.
3. Analysieren Sie die benötigte **Laufzeit** Ihres Algorithmus bzw. begründen Sie, warum die Vorgaben eingehalten werden.

Aufgabe 2.1. *Die Multiplikation dünnbesetzter Matrizen* (4+6 Punkte)

(Vorlesung am 5. Mai, Abschnitt 4.1 im Skript)

Die Zeilen-Darstellung einer $n \times m$-Matrix A funktioniert wie folgt. Ein Array $Z[1], \ldots, Z[n]$ speichert n Listen, wobei die Liste $Z[i]$ genau die von Null verschiedenen Einträge der i-ten Zeile von A speichert. Das Listenelement (j, k) aus $Z[i]$ steht für $A_{ij} = k$, d.h. die j-te Spalte der i-ten Zeile von A hat den Wert k. Die Elemente in $Z[i]$ sind aufsteigend nach dem Zellenindex j gespeichert. Ein Beispiel:

$$A = \begin{pmatrix} 0 & 6 & 1 & 3 \\ 1 & 3 & 0 & 7 \\ 8 & 0 & 0 & 5 \end{pmatrix} \qquad \begin{aligned} Z[1] &= \to (2,6) \to (3,1) \to (4,3) \\ Z[2] &= \to (1,1) \to (2,3) \to (4,7) \\ Z[3] &= \to (1,8) \to (4,5) \end{aligned}$$

Die Spalten-Darstellung von A funktioniert analog: Ein Array S enthält m Listen, welche die von Null verschiedenen Einträge der Spalten von A in "aufsteigender" Reihenfolge speichern. In unserem Beispiel sieht S wie folgt aus:

$$\begin{aligned} S[1] &= \to (2,1) \to (3,8) \\ S[2] &= \to (1,6) \to (2,3) \\ S[3] &= \to (1,1) \\ S[4] &= \to (1,3) \to (2,7) \to (3,5) \end{aligned}$$

Die Struktur eines Listenelements als Datentyp sieht folgendermaßen aus:

```
typedef struct Element {

  int index;  // Index der Spalte bzw. der Zeile des Matrixelements
  int wert;   // Wert des Matrixelements
  Element *next;  // Zeiger auf das nächste Listenelement
};
```

Für eine Matrix A bezeichne $|A|$ die Anzahl der von Null verschiedenen Matrixelemente in A.

Bitte wenden!

a) Entwerfen Sie einen Algorithmus, der aus der Zeilen-Darstellung einer Matrix A die Spalten-Darstellung von A errechnet, und analysieren Sie dessen Laufzeit in Abhängigkeit von n, m und $|A|$.

b) Es seien zwei $n \times n$-Matrizen A und B jeweils in ihrer Zeilen-Darstellung gegeben.

Entwerfen Sie einen Algorithmus, der das Matrix-Produkt $C := A \cdot B$ mit $C_{ij} := \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$ in Zeilen-Darstellung berechnet.
Die benötigte Laufzeit Ihres Algorithmus soll $\mathcal{O}(n^3)$ nicht überschreiten.

Aufgabe 2.2. *Legale Klammerungen* (2+6 Punkte)

(Vorlesung am 12. Mai, Kapitel 4.2 im Skript)

Über einem Alphabet $\Sigma = \{(,), <, >\}$ mit zwei verschiedenen Klammertypen definieren wir die Sprache $\mathcal{L}$ der legalen Klammerungen wie folgt rekursiv:

- Das leere Wort ε ist legal.
- Falls k legal ist, so sind auch (k) und $<k>$ legal.
- Falls k_1 und k_2 legal sind, so ist auch ihre Konkatenation $k_1 k_2$ legal.

Ein paar Beispiele: Die drei Ausdrücke `()()` und `<<<>>>` und `(<()><>)<>` sind legal; die drei Ausdrücke `)(` und `><` und `(>` sind nicht legal.

Ein Klammerausdruck k sei in einem Array $K[1 \ldots n]$ gespeichert, sodass jede Zelle genau eine Klammer enthält. Gesucht ist ein Algorithmus, der prüft, ob k legal ist.

Ein solcher Legalitätstest ist einfach, wenn entweder *nur* runde oder *nur* spitze Klammern vorkommen: Wenn wir den Ausdruck von der ersten zur letzten Klammer lesen und dabei für jede öffnende Klammer den (auf Null initialisierten) Zähler Z inkrementieren und für jede schließende Klammer Z dekrementieren, dann ist der Klammerausdruck genau dann legal, wenn Z niemals negativ wird und am Ende $Z = 0$ gilt.

Können wir diesen Ansatz für zwei Klammertypen verallgemeinern? Wir versuchen es mit folgendem Algorithmus:

1) Initialisiere zwei Zähler R und S mit 0. // zähle runde und spitze Klammern
2) Lies k von vorne nach hinten, dabei mache folgendes:
   2.1) Für jedes gelesene `(` inkrementiere R.
   2.2) Für jedes gelesene `)` dekrementiere R.
   2.3) Für jedes gelesene `<` inkrementiere S.
   2.4) Für jedes gelesene `>` dekrementiere S.
3) Falls in *jedem* Schritt $R \geq 0$ und $S \geq 0$ gilt und *am Ende* $R{=}S{=}0$ gilt, dann gib LEGAL aus. Andernfalls gib NICHT_LEGAL aus.

a) Geben Sie ein Beispiel an, für das dieser Algorithmus versagt. Finden Sie dazu einen Klammerausdruck k mit **beiden** Klammertypen, den der Algorithmus fälschlicherweise als legal klassifiziert.

b) Entwerfen Sie einen eigenen Algorithmus, der die Legalität von k für Klammerungen mit beiden Klammertypen korrekt prüft. Verwenden Sie dazu eine der in der Vorlesung behandelten Datenstrukturen Stack oder Queue.

Bitte wenden!

Aufgabe 2.3. *Schichtbetrieb* (8 Punkte)

(Vorlesung am 12. Mai, Abschnitt 4.3 im Skript)

Ein Binärbaum B sei in Binärbaum-Darstellung gegeben, die Struktur eines Knotens sieht folgendermaßen aus:

```
typedef struct Knoten {
  int wert;
  Knoten *links, *rechts;
};
```

Der Zeiger `root` auf die Wurzel r ist gegeben. Wir wollen B schichtweise traversieren, d.h. wir wollen zuerst den Wert der Wurzel r ausdrucken, gefolgt von den Werten der Kinder von r, gefolgt von den Werten aller Knoten in Tiefe zwei, etc.: Bevor also der Wert irgendeines Knotens u gedruckt wird, müssen vorher die Werte aller Knoten v mit geringerer Tiefe als u gedruckt werden.

Entwerfen Sie einen Algorithmus, der B schichtweise traversiert. Erklären Sie auch, welche Datenstruktur Sie (zusätzlich zur Binärbaum-Darstellung) verwenden: Zur Auswahl stehen Stacks und Queues.

Aufgabe 2.4. *Ahnenkunde* (6 Punkte)

(Vorlesung am 12. Mai, Abschnitt 4.3 im Skript)

Ein Baum B sei in Eltern-Array-Darstellung gegeben: Für jeden Knoten u ist der Elternknoten von u in Zelle Baum[u] gespeichert.

Entwerfen Sie einen Algorithmus, der für zwei gegebene Knoten u, v ihren letzten gemeinsamen Vorfahren bestimmt, d.h. den *tiefsten* Knoten w, in dessen Teilbaum sich sowohl u als auch v befinden. Ihr Algorithmus soll höchstens die Laufzeit $\mathcal{O}(\text{Tiefe}(u) + \text{Tiefe}(v))$ benötigen.

Hinweis: Sie können das Array "Baum" verwenden, um einen Knoten u zu markieren, indem Sie den Eintrag Baum[u] überschreiben.