Technische Universität
Kaiserslautern

Fachbereich Informatik
AG Algorithmen und Komplexität

Ausgabedatum: 08.11.2012
Version: 2012-11-09 18:34

5. Übungsblatt zur Vorlesung Entwurf und Analyse von Algorithmen, WS 12/13

Abgabe: *Bis Donnerstag, 15.11.2012, 12:00 Uhr, Abgabenkasten vor 48-694.*

21. Aufgabe

Track ε [3]
Track AI [3]

Betrachten Sie eine Listenimplementierung, die ein Array für die Speicherung der Elemente nutzt. Dabei wird die Größe des Arrays immer dann verdoppelt, wenn das Array voll ist und ein weiteres Element eingefügt werden soll. Weil in typischen Speicherverwaltungen ein Array nicht mehr vergrößert werden kann, müssen wir ein neues Array doppelter Länge anlegen und die vorhandenen Elemente umkopieren. Zu Beginn starten wir mit einem Array der Länge 1.

Berechnen Sie die amortisierten Kosten dafür, ein Element ans Ende der Liste anzuhängen, wenn n Anhängeoperationen (direkt) nacheinander ausgeführt werden. Dabei betrachten wir als Kosten die Schreibzugriffe, die beim erstmaligen Speichern eines Elements sowie bei Umkopieren anfallen, d. h. das Anhängen eines Elements in ein ausreichend großes Array verursacht Kosten von 1 und das Verlängern des Arrays von Länge i auf $2i$ kostet i Schreibzugriffe. (Wir gehen davon aus, dass das Array nicht initialisiert wird.)

Hinweis: Die hier analysierte Datenstruktur ist Basis für viele Implementierungen in populären Bibliotheken, zum Beispiel `java.util.ArrayList` in Java Runtime Library oder `std::vector` in C++ STL.

22. Aufgabe

	a)	b)	c)	d)
Track ε	3	3	2	[1]
Track AI	3	[3]	[2]	[1]

Gegeben sei die Permutation[1] $1, 2, 3, \ldots, n$ der Zahlen 1 bis n und ein Stack. Wir können eine neue Permutation erzeugen, indem wir die Zahlen in der Reihenfolge $1, 2, \ldots$ auf einen Stack *pushen* und – sofern der Stack nicht leer ist – zwischendurch beliebig Zahlen vom Top des Stacks entnehmen. Die erste so entnommene Zahl wird die führende Position in der Ergebnispermutation, die zweite Zahl die zweite Position usw. Zur Erzeugung einer Permutation werden dabei genau n `PUSH` und n `POP` Operationen ausgeführt.

Beispiel: Wir betrachten den Fall $n = 4$. Die Folge der Operationen `PUSH(1), PUSH(2), POP, PUSH(3), PUSH(4), POP, POP, POP` erzeugt aus der Permutation $1, 2, 3, 4$ die neue Permutation $2, 4, 3, 1$.

a) Nicht jede Folge von n `PUSH` und n `POP` Operationen kann zur Erzeugung einer Permutation verwendet werden, denn für manche Folgen würde die `POP` Operation versuchen, eine Zahl vom leeren Stack zu lesen. Wir nennen eine Folge von Operationen *zulässig*, wenn keine der in ihr enthaltenen `POP` Operation auf einen leeren Stack zugreift, sonst nennen wir sie *unzulässig*.

   Formulieren Sie eine Regel, mit deren Hilfe man einfach unzulässige von zulässigen Folgen von n `POP` und n `PUSH` Operationen unterscheiden kann. Zeigen Sie außerdem, dass zwei verschiedene zulässige Folgen stets verschiedene Permutationen erzeugen.

b) Bestimmen Sie eine einfache Formel für die Anzahl a_n der Permutationen, die wir (wie oben beschrieben) aus der Permutation $1, 2, \ldots, n$ erzeugen können.

   Hinweis: Für die Lösung dieser Teilaufgabe dürfen Sie die folgende Aussage verwenden. Sei $x_1, x_2, \ldots, x_n$ eine beliebige Folge von von n ganzen Zahlen, deren Summe $\sum_{i=1}^{n} x_i = 1$ erfüllt. Dann besitzt genau einer der zyklischen Shifts dieser Folge, d.h. nur genau eine der Folgen

$$
\begin{aligned}
&x_1, x_2, \ldots, x_n \\
&x_2, x_3, \ldots, x_n, x_1 \\
&x_3, x_4, \ldots, x_n, x_1, x_2 \\
&\vdots \\
&x_n, x_1, x_2, \ldots, x_{n-1}
\end{aligned}
$$

   die Eigenschaft, dass alle n Teilsummen seiner i ersten Zahlen echt positiv sind, $1 \leq i \leq n$. Zum Beispiel ist die Summe zur Folge $3, -5, 2, -2, 3, 0$ gleich $+1$, aber nur für den zyklischen Shift $3, 0, 3, -5, 2, -2$ sind die Teilsummen $3 = 3$, $3 + 0 = 3$, $3 + 0 + 3 = 6$, $3 + 0 + 3 - 5 = 1$, $3 + 0 + 3 - 5 + 2 = 3$ und $3 + 0 + 3 - 5 + 2 - 2 = 1$ alle echt positiv. Für den Shift $2, -2, 3, 0, 3, -5$ beispielsweise ist die Teilsumme $2 - 2 = 0$ nicht echt positiv.

[1] Eine Permutation der Zahlen 1 bis n ist eine Aufzählung dieser Zahlen in beliebiger Reihenfolge. In dieser Aufzählung muss jede der Zahlen genau einmal vorkommen; zur Darstellung einer Permutation listen wir die Zahlen entsprechend ihrer Anordnung innerhalb der Aufzählung durch Kommata getrennt in einer Zeile auf. So sind alle Permutationen der Zahlen 1,2 und 3 die folgenden: 1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1. Allgemein gilt: Die Anzahl der Permutationen der Zahlen $1, 2, \ldots, n$ ist gleich $n!$. Warum?

c) Wenn wir obiges Vorgehen nicht mit der Startpermutation $1, 2, \ldots, n$, sondern mit einer beliebigen Permutation der Zahlen 1 bis n als *Eingabe* durchführen, können wir dann jede mögliche Permutation sortieren, d.h. in die Ergebnispermutation $1, 2, \ldots, n$ überführen?

d) Wie viele verschiedene Permutationen kann man mit einer Queue und einer Sequenz aus Operationen `ENQUEUE(i)` und `DEQUEUE` sortieren?

23. Aufgabe

Track ε	[2]
Track AI	2

Implementieren Sie die Prozeduren `DEQUEUE` und `ENQUEUE` für die Realisierung einer Queue mittels ringförmig geschlossenem Feld.

Hinweis: Anders als bei *Entwurfs*aufgaben ist bei *Implementierungs*aufgaben tatsächlich detaillierter Code gefragt. Nutzen Sie bitte eine Sprache, die Ihr Übungsleiter versteht, also etwa eine der in der Vorlesung verwendeten. Hierbei ist in der Regel darauf zu verzichten, etwaige Bibliotheken die Arbeit machen zu lassen.

24. Aufgabe

	a)	**b)**
Track ε	1	[1]
Track AI	1	[1]

Wir betrachten Sätze von Prozeduren, die von einem Compiler übersetzt werden sollen. Der Compiler ist schon etwas älter; um einen Prozeduraufruf $\texttt{P}(\texttt{x}_1, \ldots, \texttt{x}_n)$ zu übersetzen muss `P` schon übersetzt worden sein.

Wenden Sie jeweils beide Implementierungen zum topologischen Sortieren an, um eine funktionierende Übersetzungsreihenfolge zu finden oder festzustellen, dass es keine solche gibt. Stellen Sie dazu als erstes die Prioritätenlisten auf.

a)
```
Prozedur a() {
  Aufruf von b()
  Aufruf von c()
}

Prozedur b() {
  Aufruf von c()
}

Prozedur c() {
  Aufruf von b()
  Aufruf von d()
}

Prozedur d() {
  Print("Juchhu")
}
```

b)
```
Prozedur a() {
  Aufruf von b()
  Aufruf von c()
}

Prozedur b() {
  Aufruf von d()
}

Prozedur c() {
  Aufruf von b()
  Aufruf von d()
}

Prozedur d() {
  Print("Fertig")
}
```

25. Aufgabe

	a)	b)	c)
Track ε	1	[1]	[1]
Track AI	1	[1]	[1]

Implementieren Sie die folgenden Mengenoperationen für Mengen in verketteter (Listen-) Darstellung analog zu der Implementierung des Durchschnitts aus der Vorlesung. Insbesondere soll auch hier die resultierende Liste wieder eine sortierte Liste sein.

a) Vereinigung ($A \cup B$)

b) Differenz ($A \setminus B$)

c) Symmetrische Differenz ($A \Delta B$)

26. Aufgabe

	a)	b)
Track ε	[5]	[2]
Track AI	[5]	[2]

In der Vorlesung haben wir gesehen, wie man Mengen mit Bitvektoren – Arrays aus Booleans – implementieren kann, was schnelle Operationen auf Kosten von Speicher erlaubt. Ein unangenehmer Nachteil entsteht bei sehr großen Universen und kleinen Mengen, auf denen nur wenige Operationen ausgeführt werden; in diesem Fall ist die Initialisierungszeit – jeder Eintrag im Array muss explizit auf `0` gesetzt werden – dominant, nicht die eigentlichen Operationen.

a) Sei $\mathcal{U} \subseteq \mathbb{N}$ beliebig mit $|\mathcal{U}| = N \in \mathbb{N}$. Entwerfen Sie eine Mengenimplementierung für Teilmengen von $\mathcal{U}$ auf Basis von Arrays, die

- die Basisoperationen ebenso in Zeit $O(1)$ ermöglicht,
- dabei $\mathcal{O}(mN)$ Speicher benötigt, aber
- in Zeit $O(1)$ initialisiert werden kann – was das eigentliche Ziel ist.

Wir nehmen an, dass Speicherallokation in konstanter Zeit vom Betriebssystem durchgeführt wird. Sie können jedoch keine Annahmen darüber machen, welchen Inhalt eine nicht initialisierte Speicherstelle direkt nach der Allokation hat. m ist hierbei die *Wortbreite*, also die Anzahl Bits, die zur Speicherung eines Integers benötigt wird.

Begründen Sie, warum die gewünschten Kriterien – also Korrektheit, Speicherbedarf und Laufzeiten – von Ihrer Datenstruktur erfüllt werden.

b) Skizzieren Sie, wie man mit Ihrer Datenstruktur aus a) die Vereinigungsoperation, also $C = A \cup B$, in Zeit $\mathcal{O}(|A| + |B|)$ durchführen kann. Sollte das nicht gehen, begründen Sie, warum nicht.

27. Aufgabe

	a)	**b)**
Track ε	1	1
Track AI	1	[1]

Wir betrachten Graphen mit der Knotenmenge $\{1, 2, \ldots, n\}$. Bestimmen Sie

a) die Anzahl der einfachen unmarkierten Digraphen in Abhängigkeit von n.

b) die Anzahl der einfachen unmarkierten Graphen in Abhängigkeit von n.

Hinweis: Beachten Sie die genauen Definitionen von Graph und Digraph aus der Vorlesung!

Hinweis: Wir gehen hier gemäß Definition davon aus, dass Knoten eine Identität haben. Anders gesprochen, wir wollen isomorphe Graphen einzeln zählen und nicht zusammenfassen.

28. Aufgabe

	a)	**b)**
Track ε	[2]	[2]
Track AI	2	2

Wir haben die Links-Sohn-Darstellung für Bäume behandelt, in der die Felder `KRaum` und `ZRaum` verwendet werden, um alle Knoten und die Struktur der Bäume darzustellen. Dabei ist es möglich, diese beiden Felder zu einem Feld mit drei Komponenten (Markierung, linkester Sohn, nächster Bruder) zu verschmelzen.

a) Wie müssen in dieser verschmolzenen Darstellung die Einträge *linkester Sohn* und *nächster Bruder* verwendet werden, um einen Baum zu repräsentieren?

Stellen Sie dazu erklärend einen beliebigen erweiterten Binärbaum mit mindestens sieben Knoten und einer Höhe von mindestens drei in dieser Darstellung dar.

b) Wie kann die Operation `create`(x, T_1, T_2, T_3) bei Verwendung dieser Baumdarstellung realisiert werden? Gehen Sie dabei davon aus, dass die drei Teilbäume T_1, T_2 und T_3 bereits im Array abgespeichert sind und der Operation als Cursor auf ihre Wurzelknoten übergeben werden.

Wo liegt der Nachteil dieser Darstellung zu der zuvor behandelten Darstellung mit den zwei Feldern `KRaum` und `ZRaum`?