Technische Universität München
Fakultät für Informatik
Lehrstuhl für Effiziente Algorithmen
Dr. Hanjo Täubig
Tobias Lieber

Sommersemester 2011
Mittelklausur
5. Juli 2011

Grundlagen: Algorithmen und Datenstrukturen

Name	Vorname	Studiengang	Matrikelnummer
.....	.....	☐ Bachelor ☐ Inform. ☐ Master ☐ Mathe. ☐ Lehramt ☐ Bio-Inf.	.....

Hörsaal	Reihe	Sitzplatz	Unterschrift
.....	.....	.....	.....

Allgemeine Hinweise zur Klausur

- Bitte füllen Sie obige Felder in Druckbuchstaben aus und unterschreiben Sie!
- Schreiben Sie nicht in *roter* oder *grüner* Farbe bzw. mit Bleistift!
- Außer Ihrem Schreibgerät und einem handbeschriebenen DIN-A4-Blatt sind keine weiteren Hilfsmittel erlaubt.
- Die Bearbeitungszeit beträgt 75 Minuten.
- Alle Antworten sind in die geheftete Angabe auf den jeweiligen Seiten (bzw. Rückseiten) der betreffenden Aufgaben einzutragen. Begründen Sie alle Ihre Schritte. Auf dem Schmierblatt können Sie Nebenrechnungen machen. Der Schmierblattbogen muss ebenfalls abgegeben werden, wird aber in der Regel nicht bewertet.

Hörsaal verlassen von ..... bis ..... / von ..... bis .....

Vorzeitig abgegeben um .....

Besondere Bemerkungen:

	A1	A2	A3	A4	A5	Σ	Korrektor
Erstkorrektur							
Zweitkorrektur							

Aufgabe 1 (9 Punkte)

Bearbeiten Sie die folgenden neun Teilaufgaben kurz, aber exakt.

a) Beweisen Sie: $n^k \in o(n^{k+2})$ durch Angabe eines passenden n_0 (eventuell in Abhängigkeit von c).

Lösungsvorschlag

Aus $n^k \leq cn^{k+2}$ folgt: $n \geq \sqrt{\frac{1}{c}}$. Somit ist jedes $n_0 \in \mathbb{N}$ größer als $\sqrt{\frac{1}{c}}$ geeignet.

b) Vergleichen Sie die Laufzeit für das Suchen auf einer sortierten Sequenz, die a) auf einem Array und b) auf einer doppelt verketteten Liste implementiert ist.

Lösungsvorschlag

Die Laufzeit einer binären Suche auf einem Array ist wie in der Vorlesung gelernt logarithmisch. Auf einer Liste haben wir keinen direkten Zugriff auf die Elemente. Daher erreichen wir bei der Suche auf doppelt verketteten Listen $\Theta(n)$ Laufzeit.

c) Wir haben in der Vorlesung selbstorganisierende Listen kennengelernt. Geben Sie an, wie gutartig sich diese Art von Listen im Vergleich zu jeder anderen Implementierung von Listen verhält und welcher Mechanismus dafür verantwortlich ist.

Lösungsvorschlag

Die erwartete Laufzeit für den Zugriff auf ein Element von Selbstorganisierenden Listen ist im Vergleich zu einer optimalen Liste (bei bekannten Zugriffswahrscheinlichkeiten) höchstens zwei Mal so schlecht. Somit ist also auch die Zugriffszeit im Vergleich zu jeder anderen Implementierung höchstens zwei Mal so schlecht.

Dafür verantwortlich ist die Move to Front Regel.

d) Geben Sie an, welche Garantie die amortisierte Analyse für eine Datenstruktur geben kann.

Lösungsvorschlag

Mittels der amortisierten Analyse wird der Worst-Case für Operationenfolgen der Länge m bewiesen.

e) In der Vorlesung haben Sie gelernt, dass (verlinkte) Listen im externen Speicher (Festplatten) in der Praxis nicht sehr performant sind. Trifft dies für Hashing im externen Speicher auch zu? Begründen Sie ihre Antwort.

Lösungsvorschlag

Diese Aussage trifft für Hashing im externen Speicher auch zu. Durch viele zufällige Zugriffe muss jeweils ein teurer Zugriff auf den externen Speicher ausgeführt werden. Dies wird explizit durch Hashing mit guter Streung provoziert.

f) Geben Sie jeweils ein Argument für und eines gegen die Verwendung von QuickSort an.

Lösungsvorschlag

Pro: QuickSort ist in der Praxis sehr schnell und benötigt im Durchschnitt nur $\mathcal{O}(n \log n)$ Elementvergleiche.

Contra: Die worst Case Laufzeit ist $\Theta(n^2)$

g) Geben Sie jeweils ein Argument für und eines gegen die Verwendung von MergeSort an.

Lösungsvorschlag

Pro: Mergesort ist ein asymptotisch optimales Sortierverfahren.

Contra: In der Praxis ist QuickSort meist schneller. MergeSort benötigt doppelt so viel Speicher wie Elemente sortiert werden.

h) Wir haben in der Vorlesung den QuickSelect-Algorithmus sowie das Radix-Sortier-Verfahren kennen gelernt. Mittels eines Sortierverfahrens kann man auch einen Selektionsalgorithmus implementieren. Mittels RadixSort würde dies in $\mathcal{O}(n)$ Zeit gehen. Welche Bedingung müssen die Elemente erfüllen, damit das beschriebene Verfahren

funktioniert? Wann würden Sie auf den QuickSelect-Algorithmus verweisen?

Lösungsvorschlag

Wenn die Schlüssel Zahlen oder Strings sind, geht dies mit RadixSort. Wenn die Elemente vergleichsbasiert sortiert werden müssen, ist ein anderer Selektionsalgorithmus nötig.

i) Geben Sie einen möglichst effizienten Algorithmus an, um den Schnitt von zwei Mengen zu berechnen ($A \cap B$). Bestimmen Sie die Laufzeit Ihres Algorithmus in Abhängigkeit der Mengengrößen ($|A| = n$ und $|B| = m$).

Lösungsvorschlag

Sortieren der Mengen, dann mittels einem Merge-Schritt einen Elementvergleich: Zeit $\mathcal{O}(m \log m + n \log n)$. Der Platzverbrauch ist linear.

Aufgabe 2 (8 Punkte)

In dieser Aufgabe betrachten wir Operationen auf einem sortierten Feld $F = [0, ..., n-1]$. Wir nehmen an, dass dieses Feld zu jedem Zeitpunkt maximal n Elemente enthält und zu Beginn bereits sortiert ist. Auf F sollen nun folgende Operationen ausgeführt werden: `insert(e)`, `remove(k)` und `find(k)`.
Nehmen Sie an, es gibt ein weiteres Feld $B = [0, ..., \lceil\sqrt{n}\rceil - 1]$ der Größe $\lceil\sqrt{n}\rceil$. Die Operationen `insert(e)` und `remove(k)` auf F werden *lazy* bearbeitet: solange das Feld B noch nicht voll ist, werden die Operationen in B (hinten) abgespeichert, statt das Feld F direkt zu bearbeiten – das Feld F bleibt also unverändert. Wir nehmen an, dass eine solche Einfügeoperation auf B konstante Kosten $\mathcal{O}(1)$ hat. Sobald das Feld B aber voll ist, muss ein Update auf F ausgeführt werden, bei welchem alle in B gespeicherten Operationen auf das Feld F angewendet werden und B geleert wird. Natürlich soll das Feld F danach wieder sortiert sein. Wir nehmen an, ein solches Update verursacht insgesamt Kosten $\mathcal{O}(n)$. Die `find()` Operation sucht immer zuerst in F nach einem Element; danach wird in B überprüft, ob das Element bereits gelöscht oder noch nicht in F eingefügt wurde.

a) Zeigen Sie, dass der amortisierte Aufwand für die Operationen `insert(e)`, `remove(k)` und `find(k)` durch diese *lazy* Bearbeitung durch $\mathcal{O}(\sqrt{n})$ beschränkt ist.

b) Beschreiben Sie, wie das Update-Verfahren in Zeit $\mathcal{O}(n)$ implementiert werden kann.

Lösungsvorschlag

a) Zur Berechnung der amortisierten Kosten verwenden wir die Kontomethode.

Betrachten wir zuerst `find(k)`: Da F sortiert ist, kann man mittels binärer Suche in logarithmischer Zeit ($\mathcal{O}(\log(n))$) das Element bestimmen. Danach muss aber noch überprüft werden, ob das Element nicht bereits gelöscht wurde. Es kann auch sein, dass das Element noch nicht im Feld F existiert. In beiden Fällen ist es nötig, das Feld B zu durchlaufen. Die dadurch entstehenden Kosten sind durch $\mathcal{O}(\sqrt{n})$ beschränkt, da das Feld nur $\sqrt{n}$ Operationen speichern kann . Die Gesamtlaufzeit für `find(k)` ist also in jedem Fall höchstens $\mathcal{O}(\log n + \sqrt{n}) = \mathcal{O}(\sqrt{n})$. Damit ist der amortisierte Aufwand auch in $\mathcal{O}(\sqrt{n})$.

Bei den Operationen `insert(e)` und `remove(k)` müssen zwei Fälle unterschieden werden: Entweder wird die Operation in das Feld B eingefügt oder es wird ein Update ausgeführt. Der zweite Fall ist der „teure“ Fall. D.h., wir müssen vorher genug auf das Konto eingezahlt haben, damit das Update „bezahlt“ werden kann. Wird ein Update ausgeführt, so wurde vorher $\sqrt{n}$-mal `insert(e)` oder `remove(k)` ausgeführt.

Für jedes `insert(e)` und jedes `remove(k)` zahlen wir $\sqrt{n}$ auf das Konto ein. Diese Operationen benötigen, wenn F nicht aktualisiert werden muss, $\mathcal{O}(1)$ Zeit. Insgesamt entstehen in diesem Fall also Kosten in

$$\mathcal{O}(\underbrace{\sqrt{n}}_{\text{Einzahlung}} + \underbrace{1}_{\text{Einfügen in } B}) = \mathcal{O}(\sqrt{n}).$$

Sobald durch `insert(e)` oder `remove(k)` ein Update auf F ausgeführt wird, ist der Wert des Kontos bei $\sqrt{n} \cdot \sqrt{n} = n$. Daher können die Kosten für das Update gedeckt

werden. In diesem Fall entstehen also Kosten in

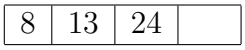
$$\mathcal{O}(\underbrace{\sqrt{n}}_{\text{Einzahlung}} + \underbrace{n}_{\text{Kosten } \textit{update}} - \underbrace{n}_{\text{Guthaben}}) = \mathcal{O}(\sqrt{n}).$$

Damit ist der amortisierte Aufwand für alle drei Operationen in $\mathcal{O}(\sqrt{n})$.

b) Wir sortieren den Puffer mit einem stabilen quadratischen Sortierverfahren nach den Elementen, die die Operation betreffen. Nun wird ähnlich wie bei Mergesort der Puffer mit dem Feld vereinigt, was aufgrund der Größe des Feldes in $\mathcal{O}(n)$ Zeit geht.

Aufgabe 3 (8 Punkte)

a) Gegeben sei das folgende dynamische Array mit $n = 3$ und $w = 4$ sowie $\alpha = 3$ und $\beta = 2$ als Parameter für das Verkleinern bzw. Vergrößern des Arrays. Führen Sie auf dem Feld die folgenden Operationen aus:



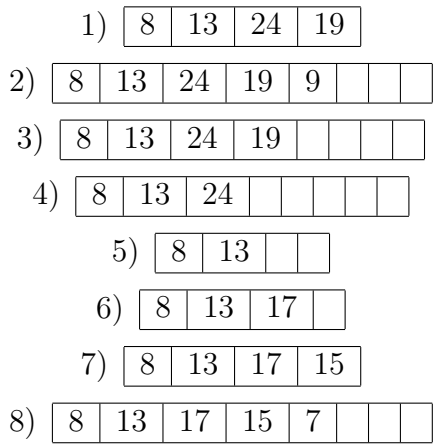
1) `push_back(19)`
2) `push_back(9)`
3) `pop_back()`
4) `pop_back()`
5) `pop_back()`
6) `push_back(17)`
7) `push_back(15)`
8) `push_back(7)`

Zeichnen Sie jeden Zwischenschritt und machen Sie die Größe des Arrays deutlich.

b) Begründen Sie kurz, warum es bei dynamischen Arrays nicht sinnvoll ist, den Parameter β größer als den Parameter α zu wählen.

Lösungsvorschlag

a)



b) Das Array würde bei jeder folgenden pop-Back-Operation umkopiert werden (Wenn kein pushBack ausgeführt wird). Dies würde die gesamte amortisierte Analyse zunichtemachen.

Aufgabe 4 (7 Punkte)

Wir betrachten Hashing mit Linear Probing für eine Hashtabelle der Größe m. Die Hashfunktion für Linear Probing für den i-ten Einfügeversuch für Schlüssel k kann wie folgt aufgeschrieben werden:

$$h_\ell(i, k) := h(k) + i \bmod m.$$

Allgemeiner kann man zum Beispiel Double Hashing definieren:

$$h(i, k) := h(k) + i \cdot h'(k) \bmod m$$

In dieser Aufgabe verwenden wir Double Hashing mit den Hashfunktionen $h(k) = 2k \bmod 11$ und $h'(k) = 3k \bmod 10$.
Führen Sie folgende Operationen auf der Hashtabelle der Größe 11 aus:

1) `insert(3)`

2) `insert(17)`

3) `insert(14)`

4) `insert(33)`

5) `insert(23)`

6) `insert(11)`

7) `insert(15)`

Lösungsvorschlag

a) i=0:

1)

0	1	2	3	4	5	6	7	8	9	10
						3				

2)

0	1	2	3	4	5	6	7	8	9	10
	17					3				

3)

0	1	2	3	4	5	6	7	8	9	10
	17					3		14		

4)

0	1	2	3	4	5	6	7	8	9	10
33	17					3		14		

5)

0	1	2	3	4	5	6	7	8	9	10
33	17	23				3		14		

6)

0	1	2	3	4	5	6	7	8	9	10
33	17	23	11			3		14		

7)

0	1	2	3	4	5	6	7	8	9	10
33	17	23	11			3	15	14		

b) i=1:

1)

0	1	2	3	4	5	6	7	8	9	10
				3						

2)

0	1	2	3	4	5	6	7	8	9	10
		17		3						

3)

0	1	2	3	4	5	6	7	8	9	10
		17		3				14		

4)

0	1	2	3	4	5	6	7	8	9	10
		17		3				14	33	

5)

0	1	2	3	4	5	6	7	8	9	10
23		17		3				14	33	

6)

0	1	2	3	4	5	6	7	8	9	10
23		17	11	3				14	33	

7)

0	1	2	3	4	5	6	7	8	9	10
23		17	11	3			15	14	33	

Aufgabe 5 (8 Punkte)

Wir betrachten die Suche nach einem Element auf einem (ganzzahligen) Zahlenstrahl. Der Zahlenstrahl ist wie die ganzen Zahlen nach oben und unten unbeschränkt.
Auf der Suche nach einer unbekannten Zahl x, die auf dem Zahlenstrahl markiert ist, dürfen Sie, neben Steueroperationen, wie `for, while` und `if`, nur die Operation `Move(int n)` benutzen. Ein Aufruf von `Move(y)` kostet y (Zeit-)Einheiten und verschiebt die aktuelle Position a auf dem Zahlenstrahl um y Zahlen nach rechts, wenn y positiv ist, und anderenfalls nach links. Wenn während der `Move` Operation die Zahl x vorkommt, was an der Markierung der Zahl erkennbar ist, dann bricht die Funktion ab und meldet den Fund. Das Verhalten der Funktion `Move` finden Sie im folgenden Algorithmus.

Algorithmus 1: bool Move(int n)

```
   // aktuelle Position a ist als globale Variable gespeichert
 1 if a.isMarked() then
 2 |   return true
 3 if n=0 then
 4 |   return false
 5 if n>0 then
 6 |   a ← a + 1
 7 |   return Move(n-1)
 8 else
 9 |   a ← a − 1
10 |   return Move(n+1)
```

Geben Sie einen Algorithmus an, der die Zahl x findet und mit der Zahl $a := 0$ (Null) beginnt. Achten Sie darauf, dass Ihr Algorithmus die Zahl x in $\mathcal{O}(|x|)$ (Zeit-)Einheiten findet und beweisen Sie diese Eigenschaft Ihres Algorithmus.

Lösungsvorschlag

```
1 bool result ← false
2 int ii ← 0
3 while ! result do
4 |   result |= Move(2^ii)
5 |   result |= Move(-2^(ii+1))
6 |   Move(2^ii)
```

Zur Analyse nehmen wir o.B.d.A an, dass x im positiven Bereich ist. Betrachten wir nur die gemachten Schritte im Positiven.
Damit im Move-Aufruf vorher nicht gefunden wurde, kann davor höchstens $x-1$ angesprungen worden sein. Bis dorthin wurde also eine Seqenz an Aufrufen zu $\frac{x-1}{2}, \frac{x-1}{4}, ..., 2, 1$ gemacht.
Die Laufzeitschritte summieren sich also zu:

$$x + (x-1) \sum_{i=0}^{\log x - 1} \frac{1}{2^i} \leq x + (x-1)2 \leq 3x.$$

Diese mit 4 multipliziert geben eine obere Schranke für die Laufzeitschritte (immer wieder zu 0 zurück (mal 2) und das ganze für den negativen Strahl auch noch (mal 2)). Somit sind die Schritte auf dem Zahlenstrahl also sicher kleiner als $12x \in \mathcal{O}(|x|)$, wie gefordert.