Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Übungsblatt 03

Algorithmen I - Sommersemester 2022

Abgabe im ILIAS bis 11.05.2022, 14:00 Uhr

Bitte beschrifte Deine Abgabe gut sichtbar mit Deinem Namen und Deiner Matrikelnummer. Achte insbesondere bei handschriftlichen Abgaben auf Lesbarkeit und genügend Platz für Korrektur-Anmerkungen. Die Abgabe erfolgt über das Übungsmodul in der Gruppe Deines Tutoriums im ILIAS. Gib Deine Ausarbeitungen in *einer* PDF-Datei ab.

Aufgabe 1 - Schreib-/Lese-/Hitzkopf (9 Punkte)

Um Überhitzung einer Festplatte zu verhindern, entwerfen wir eine Datenstruktur die nicht zu häufig an die gleiche Speicherstelle schreibt. Diese Datenstruktur D besteht aus m Arrays. Das i-te Array, welches wir A_i nennen, hat die feste Größe 2^i. Damit kann D maximal $n = \sum_{i=0}^{m-1} 2^i$ Einträge halten. Der Inhalt eines Arrays A_i kann geschützt oder ungeschützt sein. Geschützte Arrays dürfen nicht direkt überschrieben werden. Um uns merken zu können, welche Arrays geschützt sind, steht ein weiteres Array PROTECTED : $[\text{Bool}, m]$ bereit. Es ist PROTECTED$[i]$ = `true` genau dann, wenn A_i geschützt ist. Wir gehen in dieser Aufgabe stets davon aus, dass es mindestens ein ungeschütztes Array in D gibt.

Um ein neues Element x in diese Datenstruktur einzufügen, nutzen wir den folgenden Algorithmus. Wir iterieren über die Arrays $A_0, \dots, A_{m-1}$ und suchen das erste ungeschützte Array A_i. An diese Stelle setzen wir das Array A_{neu}, wobei A_{neu} alle Elemente aus $\{x\} \cup \bigcup_{j<i} A_j$ enthält. Danach gilt Array A_i als geschützt und alle vorherigen Arrays A_j mit $j < i$ als ungeschützt.

Die Laufzeit des Erstellens von A_{neu} ist linear in der Anzahl der vereinigten Elemente.

1. Sei beispielsweise $m = 4$ und D liegt im folgenden Zustand vor:

$$\begin{aligned}\textsc{Protected} &= \langle \texttt{ true } , \texttt{ false } , \texttt{ false } , \texttt{ true } \rangle \\ A_0 &= \langle 5 \rangle \\ A_1 &= \langle 1, 13 \rangle \\ A_2 &= \langle 5, 7, 9, 10 \rangle \\ A_3 &= \langle 2, 3, 4, 6, 8, 57, 90, 400 \rangle\end{aligned}$$

   Nun wird Element 30 eingefügt. Gib an, welche Veränderungen in D dadurch entstehen. (1 Punkt)

2. Mit welcher Laufzeit müssen wir im schlimmsten Fall für bei einer Einfügeoperation rechnen? Gib eine Abschätzung im O-Kalkül in Abhängigkeit von n an. (1 Punkt)

Bei genauerer Betrachtung fällt auf, dass eine Einfügeoperation oft nicht diese worst-case Laufzeit benötigt. Wir wollen nun die Laufzeit mit Hilfe von amortisierter Analyse abschätzen.

3. Gib die Laufzeit einer Einfügeoperation in Abhängigkeit des Zustands der Datenstruktur im O-Kalkül an.
   Hinweis: Überlege wann eine Einfügeoperation in Abhängigkeit der Werte von Protected besonders günstig / teuer ist. (2 Punkte)

Im Folgenden betrachten wir jeweils den Fall indem zunächst alle Arrays in D ungeschützt sind und wollen nun $k \leq n$ Elemente in D einfügen.

4. Zeige mit der Aggregatmethode, dass die amortisierte Laufzeit einer Einfügeoperation in $O(\log n)$ ist. (2 Punkte)

5. Formuliere eine Potentialfunktion und zeige mit der Potentialmethode, dass die amortisierte Laufzeit einer Einfügeoperation in $O(\log n)$ ist. (3 Punkte)

Lösung 1

1.

$$\begin{aligned}\textsc{Protected} &= \langle \texttt{ false } , \texttt{ true } , \texttt{ false } , \texttt{ true } \rangle \\ A_0 &= \langle 5 \rangle \\ A_1 &= \langle 5, 30 \rangle \\ A_2 &= \langle 5, 7, 9, 10 \rangle \\ A_3 &= \langle 2, 3, 4, 6, 8, 57, 90, 400 \rangle\end{aligned}$$

2. Idee: Im schlimmsten Fall gilt $\textsc{Protected}[i] =$ `true` für alle $i \in [0, m-2]$. Dann werden die ersten $m-1$ Arrays vereinigt. Alle übrigen Operationen laufen

in konstanter Zeit ab. Damit ergibt sich die folgende Abschätzung:

$$\begin{aligned} O(2^0 + 2^1 + 2^2 + \cdots + 2^{m-2} + 2^{m-1}) &= O\Big(\sum_{i=0}^{m-1} 2^i\Big) \\ &= O(n) \end{aligned}$$

3.
   - best case: PROTECTED[0] = `false` $\Rightarrow$ Einfügen benötigt $O(1)$
   - worst case: PROTECTED[i] = `true` für alle $i \in [0, m-2]$. $\Rightarrow$ Einfügen benötigt $O(n)$

   Im Allgemeinen hängt die Laufzeit davon ab, in welches Array geschrieben wird. Dies ist das Array mit dem niedrigsten Index i für das PROTECTED[i] = `false` gilt. Hier werden 2^i Elemente vereinigt, was Kosten in $\Theta(2^i)$ verursacht.

4. Aggregatmethode:
   Wir untersuchen wie oft in welches Array geschrieben wird, da hiervon die Kosten für eine Einfügeoperation abhängen. In Array A_0 wird jedes zweite Mal geschrieben, in Array A_1 jedes vierte Mal, usw. Im Allgmeinen wird in Array A_i jedes 2^{i+1}-te Mal geschrieben. Da die Kosten beim Schreiben in A_i in $O(2^i)$ sind (siehe Teilaufgabe 3), ergibt sich für k Einfügeoperationen

$$\begin{aligned} T(k) &= \sum_{i=0}^{m-1} O\Big(\frac{k}{2^{i+1}} \cdot 2^i\Big) \\ &= O(m \cdot k) \\ &= O(\log(n) \cdot k) \end{aligned}$$

   Nach Teilen durch die Anzahl der Einfügeoperationen erhält man die amortisierte Laufzeit für eine Einfügeoperation von $O(\log(n))$.

5. Die Potentialfunktion soll den Zustand D_i der Datenstruktur nach i Einfügeoperationen widerspiegeln. In unserem Fall soll sie die "Unordnung" messen, welche sich dadurch auszeichnet, wie viele Elemente demnächst in ein größeres Array kopiert werden. Intuitiv: wenn das erste ungeschützte Array einen großen Index hat, müssen auch viele Elemente kopiert werden.

   Als Potentialfunktion wählen wir

$$\Phi(i) = m \cdot i - \sum_{x \in D_i} \mathrm{pos}(x),$$

   wobei $\mathrm{pos}(x)$ für das größte i steht, sodass $x \in A_i$ gilt. Dabei ist die Intuition, dass mehr eingefügte Elemente die Datenstruktur unordentlicher machen $(m \cdot i)$, aber das Kopieren der Elemente in ein größeres Array einem Aufräumen gleicht und somit die Unordnung reduziert $(-\sum_{x \in D_i} \mathrm{pos}(x))$.

Zunächst verifizieren wir, dass dies eine gültige Potentialfunktion ist. Da jedes Element höchstens Position m hat, ist die Summe über alle Positionen durch $m \cdot i$ beschränkt. Damit ist die Potentialfunktion nicht negativ. Zu Beginn sind 0 Elemente eingefügt und $\Phi(0) = 0$.

Bei einer Einfügeoperation erhöht sich die Anzahl der Elemente um 1. Sei j der Index des ersten nicht geschützten Arrays, dann werden bei der Einfügeoperation 2^j Elemente kopiert, was den tatsächlichen Kosten c_i entrspricht. Für all diese Element erhöht sich die Position um mindestens 1. Also erhöht sich die Summe über die Positionen mindestens um 2^j.

Die amortisierten Kosten für die i-te Einfügeoperation ergeben sich dann als

$$\begin{aligned}
a_i &= c_i + \Phi(i) - \Phi(i-1) \\
&= 2^j + \left(m \cdot i - \sum_{x \in D_i} \text{pos}(x) \right) - \left(m \cdot (i-1) - \sum_{x \in D_{i-1}} \text{pos}(x) \right) \\
&= 2^j + m - \left(\sum_{x \in D_i} \text{pos}(x) - \sum_{x \in D_{i-1}} \text{pos}(x) \right) \\
&\leq 2^j + m - 2^j \\
&= m
\end{aligned}$$

Über k Operationen ergibt sich dann

$$T(k) = \sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} a_i \leq k \cdot m = O(k \cdot \log(n))$$

Nach Teilen durch die Anzahl der Einfügeoperationen erhält man die amortisierte Laufzeit für eine Einfügeoperation von $O(\log(n))$.

Aufgabe 2 - Dr. Meta traut niemandem (6 Punkte)

Zunächst die gute Nachricht: Mit deiner Hilfe konnte Dr. Meta die Spur des Eindringlings durch sein Labor bis zu ihrem Beginn zurückverfolgen. Bravo!
Doch nun die schlechte Nachricht: Der Weg des Eindringlings begann an einem der Seitenausgänge. Dort finden sich jedoch keine Anzeichen für ein gewaltsames Eindringen. Dr. Meta wittert Verrat in den eigenen Reihen.
Er will deswegen seine Handlanger noch einmal gründlich durchleuchten. Dazu benötigt er die Daten, die er über seine Handlanger gesammelt hat. Da Dr. Meta zu seinem eigenen Leidwesen sehr unorganisiert ist, hat er die Daten seiner n Handlanger in einem unsortierten Array A gespeichert. Jedem Handlanger ist genau ein Eintrag in A zugeordnet. Dieser beinhaltet insbesondere dessen Identifikationsnummer. Auf die an $A[i]$ gespeicherte Identifikationsnummer kannst du mit $A[i].id$ in $\Theta(1)$ zugreifen.

Einen Handlanger in $O(n)$ suchen zu müssen, dauert Dr. Meta viel zu lange. Er hat seinen Handlangern schließlich nicht grundlos eindeutige Identifikationsnummern (IDs) gegeben.
Nun liegt es wieder an dir! Konstruiere eine Datenstruktur, mit deren Hilfe der Eintrag x in A gefunden werden kann, der zu einer gegebenen ID $x.id$ gehört. Dabei soll die Datenstruktur in $O(\log n)$ den Index von x in A zurückgeben und A dabei nicht verändert werden. Du darfst davon ausgehen, dass A keine Duplikate enthält. Zudem darfst du annehmen, dass du ein Feld der Größe n in $O(n \cdot \log(n))$ sortieren kannst.

Hinweis: Denk daran, dass die Aufgabenstellung „Beschreibe“ von dir eine textuelle Beschreibung verlangt. Demzufolge ist Pseudocode keine gültige Lösung.

1. Beschreibe, wie eine solche Datenstruktur aufgebaut ist und wie mit ihrer Hilfe ein Eintrag in A gesucht wird. (2 Punkte)

2. Welche asymptotische Laufzeit benötigt die Konstruktion der Datenstruktur, die du in Teilaufgabe 1 beschrieben hast? (1 Punkt)

3. Welchen Speicherbedarf hat deine Datenstruktur? Gib eine möglichst enge Abschätzung im O-Kalkül an. (1 Punkt)

4. Begründe, warum das Suchen eines Eintrags in A mit Hilfe der neuen Datenstruktur nun in $O(\log n)$ funktioniert. (1 Punkt)

5. Angenommen, du bekommst die Garantie, dass die IDs natürliche Zahlen kleiner als eine Konstante $c_{\max}$ sind. Kannst du die Laufzeit für die Suche nach einem Eintrag in A weiter verringern? Begründe deine Antwort. (1 Punkt)

Lösung 2

1. Zusätzlich zu A wird eine weitere Folge B mit der gleichen Kapazität wie A angelegt. Diese enthält Tupel $(i, A[i].id)$, und wird (aufsteigend) nach dem zweiten Eintrag sortiert. Um den Index i zu finden, an dem sich ein gesuchtes Element x zu der Identifikationsnummer id in A befindet, benutzen wir eine binäre Suche in B anhand der Identifikationsnummern, also den zweiten Komponenten der Tupel. Haben wir den Eintrag (i^*, id) in B gefunden, geben wir $x = A[i^*]$ zurück.

2. Wir benötigen zunächst $O(n)$ Zeit, um zu jedem Eintrag $A[i]$ das Tupel $(i, A[i].id)$ zu erstellen und in B einzufügen. Anschließend sortieren wir B in $O(n \cdot \log(n))$. Insgesamt kommen wir also auf einen Zeitbedarf in $O(n \cdot \log(n))$.

3. Der zusätzliche Speicherbedarf ist linear in der Kapazität von A, also in $O(n)$, denn B hat die gleiche Größe wie A und alle Einträge in B haben eine feste Größe.

4. Das Suchen mit Hilfe von B besteht aus einer Binären Suche, von der wir wissen, dass sie in $O(\log(n))$ abläuft, und dem Zugriff auf einen Eintrag in A mit konstantem Zeitaufwand. Damit benötigen wir $O(\log(n))$ Zeit.

5. Ja: In diesem Fall geben wir B die Größe $c_{\max}$. Dann können wir in $O(n)$ für jeden Eintrag $a = A[i]$ in $B[a.id]$ den Wert i speichern. (Hier gehen wir, wie in der Aufgabenstellung genannt, davon aus, dass A keine Duplikate enthält.) Anschließend können wir den Index einer gegebenen Zahl id in A in $O(1)$ bestimmen, indem wir $B[id]$ abfragen.

Aufgabe 3 - Queues (5 Punkte)

In dieser Aufgabe wollen wir uns näher mit der Modellierung einer Queue beschäftigen (siehe Folie 10 der vierten Vorlesung). Diese soll die folgenden Operationen unterstützen:

- pushBack(elem) fügt elem in die Queue ein
- popFront() gibt ältestes Element in Queue zurück und löscht es aus der Queue

1. Wie kann eine solche Queue mit Hilfe einer einfach verketteten Liste modelliert werden? Gib an, welche zusätzlichen Daten benötigt werden und wie pushBack und popFront funktionieren. Beschreibe beide Operationen kurz. Bestimme und begründe ihre asymptotische Laufzeit. (2 Punkte)
2. Wie kann eine solche Queue mit Hilfe *eines* dynamischen Arrays modelliert werden? Gib an, welche zusätzlichen Daten benötigt werden und wie pushBack und popFront funktionieren. Beschreibe beide Operationen kurz. Bestimme und begründe ihre asymptotische Laufzeit.
   Hinweis: Beide Operationen sind amortisiert in $\Theta(1)$ durchführbar. (3 Punkte)

Lösung 3

1. Wir gehen davon aus, dass die einfach verkettete Liste bereits einen Zeiger auf den head-Knoten (Dummy-Knoten wie in der Vorlesung vorgestellt) besitzt. Dann benötigen wir zusätzlich nur noch einen Zeiger auf den letzten, also den tail-Knoten der Liste.
   - pushBack: Lege einen neuen Knoten an, der als Wert elem erhält. Der next-Pointer des tail-Knoten wird auf diesen neuen Knoten gesetzt. Der next-Pointer des neuen Knoten wird auf den head-Knoten der Liste gesetzt.
     $\Rightarrow \Theta(1)$, da für jede pushBack-Operation eine konstante Anzahl Zeiger verändert wird und Anlegen eines Knoten konstante Zeit benötigt
   - popFront: Sollte die Liste leer sein, gibt es nichts zu tun. Ansonsten speichere den Wert des ersten Knoten in der Liste ab. Setze den next-Pointer des head-Knoten auf den nächsten Knoten und gib den zwischengespeicherten Wert zurück.
     $\Rightarrow \Theta(1)$, analog zu pushBack

2. Neben dem Array benötigen wir zwei weitere Variablen `first` und `last`, um uns Indizes im Array zu merken. `first` wird den Index halten, an dem das Element steht, welches als nächstes von popFront() zurück gegeben wird. `last` wird den Index halten, an dem das zuletzt eingefügt Element in der Queue steht.
   Ausgehend von einem Array mit Kapazität c und Größe n:

   - pushBack: Wenn $n < c$, dann setze `last` auf (`last` + 1) `mod` c und füge das neue Element am Index `last` ein. Ansonsten erstelle ein neues Array der Kapazität $2c$ und füge die Elemente aus dem kleineren Array in der folgenden Reihenfolge ein: Falls `last` < `first`, dann

     a) Elemente an den Indizes im Intervall [`first`, $n-1$]

     b) Elemente an den Indizes im Intervall [0, `last`]

     Ansonsten kopiere direkt die Elemente an den Indizes im Intervall [`first`, `last`]. Setze dann `first` auf 0 und `last` auf $c+1$, füge das neue Element am Index `last` ein.
     $\Rightarrow$ Laufzeit in amortisiert $\Theta(1)$, wie beim Einfügen in ein „normales“ dynamisches Array. Der einzige Unterschied ist die die Kopier-Reihenfolge, welche jedoch keinen zusätzlichen Zeitaufwand benötigt

   - popFront: eine Strategie ist, die Kapazität des Arrays nie zu verkleinern. Dann : Falls `first` $< n-1$, inkrementiere `first` um 1, ansonsten setze `first` auf 0.
     $\Rightarrow \Theta(1)$, da nur eine Variable verändert wird