Übungsklausur Algorithmen I

Aufgabe 1. (*Algorithm Engineering*) [2 Punkte]

Nennen Sie zwei Konzepte, die Algorithm Engineering im Gegensatz zu theoretischer Algorithmik ausmachen.

Lösung

Implementierung, Experimente, reale Eingaben, realistische Maschinenmodelle, konstante Faktoren berücksichtigen

Lösungsende

Aufgabe 2. (*Listen*) [4 Punkte]

- Nennen Sie zwei Operationen, die doppelt verkettete Listen in konstanter Worst-Case-Zeit unterstützen, einfach verkettete Listen jedoch nicht.
- Nennen Sie zwei Nachteile von verketteten Listen gegenüber beschränkten Feldern.

Lösung

- *insert*, *remove*, *popBack*
- – kein beliebiger Elementzugriff in konstanter Zeit
  – Speicheraufwand für Zeiger
  – kein cache-effizientes Iterieren

Lösungsende

Aufgabe 3. (*Hashing mit Kollisionen*) [2 Punkte]

Nennen Sie einen Vorteil und einen Nachteil von Hashing mit verketteten Listen gegenüber Hashing mit linearer Suche.

Lösung

- *Vorteile:* insert in $O(1)$, für dichte Hashtabellen besser, referentielle Integrität, Leistungsgarantien mit universellem Hashing
- *Nachteile:* nicht so platz-effizient, nicht so cache-effizient, nicht so einfach

Lösungsende

Lösungsvorschlag

Aufgabe 4. (*Dynamische Arrays*) [6 Punkte]

- Welche der folgenden Operationen eines unbeschränkten/dynamischen Arrays haben im schlimmsten Fall (ohne Amortisierung) $\Theta(n)$ Zeitkomplexität? pushBack, Folge von n pushBack, Elementzugriff, Abfrage der Größe
- Geben Sie die amortisierte Laufzeit der Operation pushBack an und zeigen Sie dies per amortisierter Analyse mit der Bankkontomethode. Nehmen Sie an, dass die Kapazität des Arrays verdoppelt wird, sobald sie nicht mehr ausreichend ist, und dass nur pushBack-Operationen ausgeführt werden.

Lösung

- *pushBack*, Folge von *n pushBack*

  Korrekturhinweis: Insbesondere sind $[\cdot]$ (Elementzugriff) und $|\cdot|$ (Größe) falsche Antworten.

- – Die Operation pushBack hat amortisiert konstante Laufzeit ($O(1)$).
  – Beweis mit der Bankkontomethode:
  Zu zeigen ist, dass man der Operation ein konstantes Gehalt g zuweisen kann, sodass für die Kosten $T(n)$ für eine Folge von n Operationen gilt: $T(n) \leq n \cdot g$.
  Wir zeigen per Induktion, dass mit einem Gehalt von 3 Tokens für die pushBack-Operation der Kontostand nach jeder pushBack-Operation ≥ 0 ist. Dazu zeigen wir, dass nach jeder Reallokation das Konto nicht überzogen wird.
    * Basisfall $n = 1$: Gegeben ist ein einzufügendes Element. Das Array wird auf Kapazität 1 alloziert (Kosten 0). Das Element wird per pushBack eingefügt (Kosten 1). Auf dem Konto verbleiben 2 von 3 Tokens, was ≥ 0 ist.
    * Induktionsschritt: Wir nehmen an, dass nach dem Einfügen des $n+1$-ten Elements der Kontostand ≥ 0 war. Jetzt ist das Array der Kapazität $2n$ voll belegt und wir wollen ein weiteres Element einfügen. Inzwischen wurden $3 \cdot n$ Tokens als Gehalt für das Einfügen der n Elemente Nummer $n+2$ bis $2n+1$ ausgezahlt. Bisher wurden $n-1$ Tokens für das Einfügen dieser Elemente bezahlt, $2n+1$ Tokens verbleiben also auf dem Konto. Wir führen eine Reallokation auf Kapazität $4n$ durch und bezahlen das Umkopieren von $2n$ Elementen (Kosten $2n$ Tokens) und das Einfügen des $2n+1$-ten Elements (Kosten 1 Token). Der Kontostand ist danach $0 \geq 0$, q.e.d.

Lösungsende

Lösungsvorschlag

Aufgabe 5. (*Sortieren*) [4 Punkte]

- Nennen Sie einen Vorteil von Radixsort gegenüber Mergesort (bei geeigneten Schlüsseln) und einen Vorteil von Mergesort gegenüber Quicksort.
- Warum wird Quicksort in der Praxis trotzdem sehr häufig eingesetzt?

Lösung

- – *Vorteil Radixsort gegenüber Mergesort:* Zeit in $O(n)$ statt $\Theta(n \log n)$ bei ganzzahligen Schlüsseln
  – *Vorteile Mergesort gegenüber Quicksort:* stabil, $O(n \log n)$ bzw. $n \log n + O(n)$ Vergleiche im worst-case
- average-case-Laufzeit $n \log n$ bei Randomisierung, praktisch meist der schnellste Algorithmus, inplace

Lösungsende

Aufgabe 6. (*Lösen von Rekurrenzen*) [2 Punkte]

Bestimmen Sie die Lösungen der folgenden Rekurrenzen im Θ-Kalkül mit der einfachen Form des Master-Theorems:

$T(1) = 1, T(n) = 16T(n/4) + 4n, n = 4^k, k \in \mathbb{N}$
$S(1) = 55, S(n) = n + 3S(n/3), n = 3^k, k \in \mathbb{N}$

Lösung

$T(n) \in \Theta\left(n^{\log_4 16}\right) = \Theta\left(n^2\right), S(n) \in \Theta(n \log n)$

Lösungsende

Aufgabe 7. (*O-Kalkül*) [5 Punkte]

a. Zeigen Sie: $\log(n!) \in O(n \log n)$. [2 Punkte]

Lösung

Es gilt $n! \leq n^n \Rightarrow \log(n!) \leq \log(n^n) = n \log n \Rightarrow \log(n!) = O(n \log n)$.

Lösungsende

b. Zeigen Sie: $\log(n!) \in \Omega(n \log n)$. [3 Punkte]

Lösung

$n! \geq \lceil \frac{n}{2} \rceil \cdot (\lceil \frac{n}{2} \rceil + 1) \cdot \ldots \cdot n \geq \lceil \frac{n}{2} \rceil^{\lceil n/2 \rceil} \geq \left(\frac{n}{2}\right)^{n/2}$
$\Rightarrow \log(n!) \geq \frac{n}{2} \log(\frac{n}{2}) = \frac{n}{2} (\log(n) - \log(2))$
$\Rightarrow \forall n \geq 2^2 : \log(n!) \geq \left(\log(n) - \frac{1}{2} \log(n)\right) = \frac{n}{4} \log n \Rightarrow \log(n!) = \Omega(n \log n)$

Lösungsende

Lösungsvorschlag

Aufgabe 8. (*Entwurf einer Datenstruktur*) [8 Punkte]

Eine Datenstruktur D soll Paare der Form $(\mathit{Schluessel},\ \mathit{Wert})$ speichern (sowohl *Schluessel* und *Wert* seien Zahlen aus $\mathbb{Z}$). Paare (x,c) und (y,c') mit $x = y$ dürfen in D **nicht gleichzeitig** vorkommen. Weiter soll D folgende Operationen mit jeweils gegebenem Laufzeitverhalten unterstützen (n bezeichne dabei die Anzahl der in D enthaltenen Paare):

- $\mathit{insert}(x : \mathit{Schluessel},\ c : \mathit{Wert})$
  fügt (x,c) in D ein. Ist schon ein Paar (y,c') mit $y = x$ in D vorhanden, so wird D nicht verändert. Der Zeitbedarf sei erwartet $O(\log n)$.

- $\mathit{removeMin} : \mathit{Schluessel} \times \mathit{Wert}$
  entfernt aus D ein Paar (x,c) mit **minimalem Wert** c und liefert das entfernte Paar als Ergebnis zurück. Der Zeitbedarf sei erwartet $O(\log n)$.

- $\mathit{contains}(x : \mathit{Schluessel}) : \mathit{boolean}$
  stellt fest, ob D ein Paar (y,c) mit $y = x$ enthält. Der Zeitbedarf sei erwartet $O(1)$.

Anwendungsbedingt sei bekannt, dass in D zu jedem Zeitpunkt höchstens m Paare gleichzeitig vorhanden sind, d. h. es gilt stets $n \leq m$. Über die Beschaffenheit der auftretenden Paare wisse man im Voraus aber nichts.

a. Skizzieren Sie, wie Sie diese Datenstruktur und die drei beschriebenen Operationen realisieren würden. [6 Punkte]

Lösung

Man realisiere D mit Hilfe eines binären Heaps B mit maximaler Größe m und einer Hashtabelle H mit Kollisionsauflösung mittels linearer Listen, die über $2m$ Slots verfügt. Bei jedem „Systemstart“ werde für H eine Hashfunktion aus einer universellen Familie zufällig gewählt. Für jedes (x,c) in D müssen B und H einen Eintrag enthalten, c ist *Key* bzgl. des binären Heap B und x ist *Key* bzgl. der Hashtabelle H.
Beim *insert* von (x,c) führt man zuerst $H.\mathit{find}(x)$ aus. Ist für x bereits ein Eintrag vorhanden, so tut man nichts weiter. Sonst führe man noch $B.\mathit{insert}(x,c)$ und $H.\mathit{insert}(x)$ aus.
Bei *removeMin* führt man zuerst $B.\mathit{deleteMin}()$ aus und gibt das vormals minimale Element x und seinen Heap-Schlüssel c als (x,c) zurück. Danach führt man dann $H.\mathit{remove}(x)$ aus.
Bei $\mathit{contains}(x : \mathit{Schluessel})$ führe man $H.\mathit{find}(x)$ aus um herauszufinden, ob es ein Paar (y,c) mit $y = x$ gibt in D.

Lösungsende

b. Begründen Sie kurz, warum die drei Operationen in Ihrer Realisierung das geforderte Laufzeitverhalten aufweisen. [2 Punkte]

Lösung

Verwendete Heap-Operationen: Benötigen alle $O(\log n)$ Worst-Case-Zeit.
Hashtable-Operationen: Da H über $2m \geq 2n = \Omega(n)$ Slots verfügt und eine Hashfunktion aus einer universellen Familie zufällig gewählt wurde, dauern *find* und *remove* erwartet konstante Zeit, *insert* braucht sowieso deterministisch konstante Zeit.
Datenstruktur D: *insert* und *removeMin* brauchen erwartete Zeit $O(1)$ plus Worst-Case-Zeit $O(\log n)$, also insgesamt erwartet $O(\log n)$ Zeit. Die Operation *contains* braucht nur erwartet $O(1)$ Zeit, da ja nur $H.\mathit{find}(x)$ ausgeführt wird.

Lösungsende

Lösungsvorschlag

Aufgabe 9. (*Hashing*) [4 Punkte]

Betrachten Sie die folgende Hashtabelle:

0	1	2	3	4	5	6	7	8	9	10
05	01	27	37	44	34	19	71		99	

Zur Kollisionsauflösung wird Hashing mit linearer Suche (mit Puffer $m' = 1$) verwendet. Die Hashfunktion $h(x) = x$ DIV 10 bildet eine zweistellige Zahl $x \in \{01, \ldots, 99\}$ auf ihre höchstwertige Ziffer ab. Beispiel: $h(62) = 6$, $h(06) = 0$.

a. Sei die Tabelle in dem oben angegebenen Zustand. Geben Sie den Zustand der Tabelle nach dem Entfernen von 37 an. [2 Punkte]

0	1	2	3	4	5	6	7	8	9	10

Lösung

0	1	2	3	4	5	6	7	8	9	10
05	01	27	34	44	19		71		99	

Lösungsende

b. Sei die Hashtabelle nun leer. Fügen Sie die Zahlen 12, 17, 88, 89, 05, 22, 48, 49, 50 in dieser Reihenfolge in die Hashtabelle ein. [2 Punkte]

0	1	2	3	4	5	6	7	8	9	10

Lösung

0	1	2	3	4	5	6	7	8	9	10
05	12	17	22	48	49	50		88	89	

Lösungsende