Technische Universität
Kaiserslautern

Fachbereich Informatik
AG Algorithmen und Komplexität

Ausgabedatum: 12.12.2013
Version: 2013-12-13 15:33

9. Übungsblatt zur Vorlesung Entwurf und Analyse von Algorithmen, WS 13/14

Abgabe: *Bis* ***Freitag****, 20.12.2013, 12:00 Uhr, Kasten im Treppenhaus 48-6.*

Der Fehler der Woche

[Speicherbedarf] $\omega(n)$ trifft nicht zu, da kein unendlicher Speicher benötigt wird.

Was ist falsch?

Basisaufgaben

B1: Hashverfahren

2 Punkte

Wir betrachten Hashing mit dem Universum $\mathbb{N}$ und der Hash-Funktion $h(x) = x^2 \mod 17$. Unsere Hash-Tabelle habe entsprechend die Adressen $\{0, 1, 2, \ldots, 16\}$. Gegeben ist die Schlüsselfolge $16, 22, 5, 19, 4, 12, 1, 7, 10, 3$.

a) Wie sieht die Hash-Tabelle nach dem Einfügen der Schlüsselfolge in die anfangs leere Hash-Tabelle bei Verwendung von *linear probing* für $\gamma = 3$ aus?

b) Wie sieht die Hash-Tabelle nach dem Einfügen der Schlüsselfolge in die anfangs leere Hash-Tabelle bei Verwendung von *quadratic probing* aus?

c) Wie sieht die Hash-Tabelle nach dem Einfügen der Schlüsselfolge in die anfangs leere Hash-Tabelle bei Verwendung von *add to hash* aus?

B2: Speicherbedarf der Graphtraversierungen 3 Punkte

Welche der Graphtraversierungalgorithmen, die Sie aus der Vorlesung kennen, brauchen wie viel Speicher zusätzlich zur Eingabe und dem Array `besucht`?

Geben Sie für die Algorithmen

- `Tiefensuche` und
- `Breitensuche`

jeweils an, in welchen der Komplexitätsklassen $\mathcal{O}(1)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n)$ und $\omega(n)$ der zusätzliche Speicherbedarf im

- Best-Case und im
- Worst-Case liegt.

Hier ist n die Anzahl Knoten im Eingabegraph, den wir als zusammenhängend annehmen.

B3: Worst-Case für Löschen in AVL-Bäumen 3 Punkte

a) Geben Sie eine unendliche Klasse von AVL-Bäumen an, für die das Löschen eines geeigneten Blattes v (Ihre Wahl) für jeden Knoten auf dem Weg von der Wurzel zu v eine Rotation benötigt. Begründen Sie Ihre Behauptung.

b) Illustrieren Sie Ihre Lösungen zu B4 und a), indem Sie Ihre Implementierung aus B4 auf einen Baum gemäß a) anwenden, der mindestens Höhe 7 hat.

B4: Löschen in AVL-Bäumen 4 Punkte

Implementieren Sie die Operation `DELETE` für AVL-Bäume und begründen Sie die Korrektheit.

Benutzen Sie die gleiche Baumrepräsentation wie `INSERT` im Buch, also mehrfache Verkettung ohne Vaterzeiger. Sie können eine geeignete Implementierung der Rotationsoperationen und der Neuberechnung des Balancegrades eines einzelnen Knotens als gegeben annehmen; bitte spezifizieren Sie deren Schnittstellen.

Aufbauaufgaben

26. Aufgabe

2 + 3 Punkte

Ein *Treap* T ist ein binärer Baum über Einträgen $(k_i, p_i) \in \mathbb{N}^2$, $0 \leq i \leq n$, mit den folgenden Eigenschaften:

1. T ist ein binärer Suchbaum bezüglich der *Schlüssel* k_i.
2. T ist ein min-heap-geordneter Baum bezüglich der *Prioritäten* p_i, das heißt die Priorität eines jeden inneren Knotens ist nicht größer als die seiner Kinder.

a) Zeigen oder widerlegen Sie: für jede Menge von Einträgen mit paarweise verschiedenen Schlüsseln und Prioritäten gibt es genau einen Treap.

b) Implementieren Sie die Einfügeoperation auf Treaps. Ist der einzufügende Schlüssel bereits enthalten, so soll er die neue Priorität erhalten.

Welche Laufzeit hat Ihr Algorithmus?

27. Aufgabe

2 + 1 Punkte

a) Sei Σ ein endliches Alphabet. Entwerfen Sie eine *fast* uniforme Hashfunktion für Wörter der Länge n über Σ, also

$$\text{hash} : \Sigma^n \to [0..m] \,,$$

und begründen Sie die Korrektheit Ihrer Funktion. Fast uniform soll hier heißen, dass

$$|\{w \mid \text{hash}(w) = i\}| \in \left\{ \left\lfloor \frac{|\Sigma|^n}{m+1} \right\rfloor, \left\lceil \frac{|\Sigma|^n}{m+1} \right\rceil \right\}$$

für alle $i \in [0..m]$.

b) Entwerfen Sie eine fast uniforme Hashfunktion

$$\text{hash} : \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}^3 \to [1..5] \,,$$

für die $\text{hash}(\texttt{FCK}) = 2$, und begründen Sie die Korrektheit Ihrer Funktion.

28. Aufgabe

5 Punkte

Entwerfen Sie eine Datenstruktur für ganze Zahlen, die die folgenden Operationen mit den gegebenen asymptotischen Laufzeiten (für n die Anzahl der gespeicherten Elemente) unterstützt:

`FIND(e)` prüft in erwarteter Zeit $\mathcal{O}(\log n)$, ob das gegebene Element e enthalten ist.

`INSERT(e)` fügt ein neues Element e in erwarteter Zeit $\mathcal{O}(\log n)$ hinzu.

`UNDO` entfernt in Zeit $\mathcal{O}(1)$ das Element aus der Datenstruktur, das (unter allen enthaltenen) zuletzt hinzugefügt wurde, und gibt es zurück.

Erwartete Zeit heißt hier, dass wir annehmen, dass

1. die untersuchte Operation nach n `INSERT`-Operationen mit paarweise verschiedenen Schlüsseln aus E, $|E| \geq n$ durchgeführt wird, wobei die Permutation dieser Schlüssel uniform zufällig aus allen möglichen gewählt ist, und dass außerdem
2. der Parameter `e` uniform zufällig aus E bzw. im Falle von `INSERT` aus E ohne die bereits enthaltenen Elemente gewählt wird.

Der Speicherbedarf der Datenstruktur soll in $\Theta(n)$ sein.