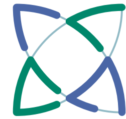


Übungsblatt 09

Algorithmen I – Sommersemester 2024

Abgabe im ILIAS bis 28.06.2024, 18:00 Uhr

Die Abgabe erfolgt alleine oder zu zweit als *eine* PDF-Datei über das Übungsmodul in der Gruppe eures Tutoriums im ILIAS. Beschriftet die Abgabe deutlich mit Matrikelnummer und Name.

- Achtet bei handschriftlichen Abgaben auf Lesbarkeit.
- Achtet darauf, ob Algorithmen in Worten oder Pseudocode beschrieben werden sollen.
- Es werden immer asymptotisch möglichst effiziente Algorithmen erwartet, wenn nicht anders angegeben.
- Wenn Korrektheits- oder Laufzeitanalysen gefordert sind, behandelt diese separat von der Algorithmenbeschreibung.

Gesamtpunkte: 20 (+ 3 Bonus)

Aufgabe 1 - Stapelweise Szenarien (4 Punkte)

Bestimme für folgende Situationen, welche möglichst einfache und effiziente Datenstruktur jeweils geeignet ist und gib an, welche Operationen der von dir gewählten Datenstruktur dabei relevant sind.
Hinweis: In der Vorlesung haben wir folgende Datenstrukturen kennengelernt: Doppelt- und einfach-verkettete Listen, (dynamische) Arrays, Hashtabellen, binäre Heaps, (2, 3)-Bäume, Adjazenzliste.

a) In einer Bibliothek sind alle Bücher stets nach dem Titel lexikographisch sortiert. Jedes Buch kann ausgeliehen und zurückgegeben werden. Außerdem soll stets nachgeprüft werden können, ob ein Buch mit gegebenem Titel aktuell verfügbar ist. (1 Punkt)

b) Dr. Meta sammelt Einklebesticker mit Fußballstars der EM. Hierfür kauft er regelmäßig kleine Tüten, die jeweils fünf zufällige Karten enthalten. Er möchte nun wissen, ob es sich lohnt, die neuen Karten zu behalten oder ob er bereits im Besitz einer neu gezogenen Karte ist. (1 Punkt)

c) Der Musikdienstleister Metaify möchte eine neuartige Wiedergabeliste anbieten. Dabei können neue Songs an einer beliebigen Stelle der Liste hinzugefügt oder auch wieder gelöscht werden. Das verwendete Gerät spielt dann stets den ersten Song der Wiedergabeliste. (1 Punkt)

d) Da Dr. Meta beim Jenga[1] spielen stets verliert, hat er einen Algorithmus entworfen, der ihm den besten Stein zum Entfernen empfehlen soll. Der Algorithmus ordnet jedem Stein die Wahrscheinlichkeit zu, dass dieser den Turm zum Einsturz bringt und empfiehlt stets den Stein mit der geringsten Wahrscheinlichkeit. Wird ein Stein in einer Runde entfernt, können sich die Wahrscheinlichkeiten von anderen Steinen, die noch im Spiel sind, ändern. (1 Punkt)

Lösung 1

a) Hier ist ein $(2,3)$-Baum geeignet, da wir Einfügen, Suchen und Entfernen in sortierten Listen wollen.

b) Hier ist eine Hashtabelle geeignet, da wir durch Suchen und Einfügen Duplikate erkennen können.

c) Hier ist eine doppeltverkettete Liste geeignet, da wir am Anfang, am Ende und vor bzw. nach gegebenen Elementen Einfügen oder Entfernen wollen.

d) Hier ist ein Heap geeignet, da wir das Minimum entfernen und Prioritäten anpassen wollen.

Aufgabe 2 - Bereich~~s~~anfragen (11 Punkte)

Für ein Element e, das wir in einem $(2,3)$-Baum verwalten, bezeichnen wir die Anzahl der Elemente, die links von e gespeichert sind, als Rang von e. Für ein Element, das nicht im Baum gespeichert ist, definieren wir den Rang des Elementes als den Rang des nächst größeren, im Baum gespeicherten Elementes. Der Rang entspricht also genau der Anzahl der Elemente im $(2,3)$-Baum, deren Schlüssel echt kleiner sind als der Schlüssel von e.

Wir wollen nun $(2,3)$-Bäume so anpassen, dass wir zusätzlich den Rang eines Elements anfragen können.

a) Beschreibe, welche zusätzlichen Informationen in den inneren Knoten eines $(2,3)$-Baumes gespeichert werden müssen, um in $O(\log(n))$ Zeit den Rang eines Elementes bestimmen zu können. (1 Punkt)

b) Begründe, wieso das asymptotische Laufzeitverhalten von `insert`(e : Element), `find`(e : Element) und `remove`(e : Element) trotz der Ergänzung aus der vorherigen Teilaufgabe gleich bleibt. (2.5 Punkte)

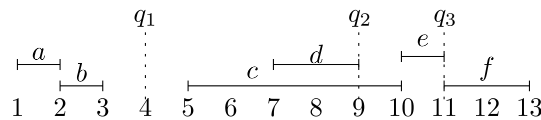
[1] `https://de.wikipedia.org/wiki/Jenga`

Hinweis: Überlege, wie sich die zusätzlichen Informationen in den Knoten durch Operationen auf dem $(2,3)$-Baum ändern können und wie sie aktuell gehalten werden können.

c) Beschreibe einen Algorithmus in Worten, der auf einem $(2,3)$-Baum mit deinen Erweiterungen aus bisherigen Teilaufgaben in Zeit $O(\log n)$ eine Anfrage des Rangs `rank`(e) eines Elements e beantwortet. Begründe, dass dein Algorithmus das geforderte Laufzeitverhalten hat. (2.5 Punkte)

Falls du bisherige Teilaufgaben nicht lösen konntest, darfst du die oben beschriebene Datenstruktur als Blackbox verwenden.

Wir nutzen nun $(2,3)$-Bäume mit Unterstützung von Ranganfragen, um eine Menge I von Intervallen mit Grenzen in $\mathbb{N}$ zu verwalten. Zusätzlich möchten wir *Intervallanfragen* beantworten. Eine Intervallanfrage besteht aus einer natürlichen Zahl x und fragt, ob es ein Intervall in I gibt, welches diese Zahl enthält.



Sechs Intervalle und drei Intervallanfragen $q_1 = 4$, $q_2 = 9$ und $q_3 = 11$, die jeweils mit “Nein”, “Ja” und “Ja” zu beantworten sind.

Gesucht ist nun eine Datenstruktur, mit der wir in $O(\log(n))$ Zeit Intervalle einfügen und löschen, sowie Intervallanfragen beantworten können, wobei n die Anzahl verwalteter Intervalle ist.

d) Betrachte das obige Beispiel mit seinen Intervallen. Gib für jede Zahl $x \in \{q_1, q_2, q_3\}$ an, wie viele Intervalle in x oder echt links von x beginnen und wie viele Intervalle echt links von x enden.
Beispiel: Für $x = 5$ beginnen drei Intervalle in x oder echt links davon und zwei Intervalle enden echt links von x. (1 Punkt)

e) Du darfst für diese Teilaufgabe annehmen, dass du für jede natürliche Zahl x weißt, wie viele der verwalteten Intervalle in x oder echt links von x beginnen und wie viele echt links von x enden. Erkläre kurz, wie du damit bestimmen kannst, ob x in einem der Intervalle liegt. (1 Punkt)

Wir nutzen nun zwei $(2,3)$-Bäume T_S und T_E, die Ranganfragen in $O(\log(n))$ unterstützen. Für jedes Intervall, das durch unsere Datenstruktur verwaltet wird, speichern wir seinen Startpunkt in T_S und seinen Endpunkt in T_E. Im Folgenden darfst du annehmen, dass an jedem Punkt höchstens ein Intervall startet und höchstens ein Intervall endet. Wir bezeichnen die Datenstruktur, die aus T_S und T_E besteht, als *Intervallbaum*. Der Intervallbaum soll das Einfügen (`insert`$([a,b] \colon \mathbb{N} \times \mathbb{N})$) und Löschen (`remove`$([a,b] \colon \mathbb{N} \times \mathbb{N})$) von Intervallen, sowie Intervallanfragen (`intervalQuery`$(x \colon \mathbb{N})$) unterstützen.

f) Beschreibe, wie `insert` und `remove` im Intervallbaum umgesetzt wird. (1 Punkt)

g) Beschreibe in Worten einen Algorithmus, der auf einem Intervallbaum in Zeit $O(\log n)$ eine Intervallanfrage `intervalQuery` beantwortet. Begründe, dass dein Algorithmus das geforderte Laufzeitverhalten hat. (2 Punkte)

Lösung 2

a) Wir merken uns in jedem Knoten des $(2,3)$-Baumes wie viele Elemente in dem Bereich, der über den Knoten erreicht werden kann, gespeichert sind. Die Grenzen sind hierbei jeweils mit enthalten.

b) Wir betrachten die Operationen einzeln:

- `find` ignoriert die zusätzlichen Daten und kann somit ohne Anpassung weiterverwendet werden. Da wir am Datenbestand nichts ändern, müssen auch die zusätzlichen Informationen nicht angepasst werden.
- `insert` muss nun die zusätzlichen Informationen anpassen, nachdem das Element an der korrekten Stelle eingefügt wurde. Wir passen die Daten beim rekursiven Aufteilen der Knoten an. Wir erhöhen zuerst den Zähler des Knotens unter dem wir das neue Element eingefügt haben um eins. Wenn wir nun einen Knoten aufteilen, können wir seine neue Elementanzahl aus den Elementanzahlen seiner Kinder brechnen. Genauso verfahren wir mit dem neuerstellten Knoten. Dies setzen wir fort bis kein Aufteilen mehr nötig ist. Danach müssen wir noch rekursiv bis zur Wurzel gehen und die Elementanzahl um eins für das neue Element erhöhen. Falls wir die Wurzel aufteilen, müssen wir auch für die neue Wurzel die Daten neu berechnen.
- `remove` Wir gehen ähnlich zu `insert` vor, in dem wir das Element entfernen und die entsprechenden Daten anpassen. Wir senken zuerst die Anzahl des direkten Elternknotens um eins. Danach gehen wir rekursiv vor. Bei `fuse` addieren wir die gespeicherten Anzahlen und bei `rebalance` passen wir die Anzahl in beiden Knoten basierend auf den Daten der Kindknoten an. Auch hier müssen wir im Anschluss die Änderung um eins bis zur Wurzel hochpropagieren.

Somit erhöht sich die Anzahl der benötigten Operationen nur um einen konstanten Faktor. Das rekursive Vorgehen bis zur Wurzel hat ebenfalls eine Laufzeit linear in der Baumhöhe.

c) Wir gehen ähnlich zu `find` in klassischen $(2,3)$-Bäumen vor. Wir suchen das nächstgrößere Element in der Liste mithilfe der Schlüssel und speichern uns dabei wie viele Elemente wir links von uns liegen lassen. Betrachte beim Traversieren eines Knotens v alle Kinder von v und sei u das erste Kind mit $key(u) \geq key(e)$. Dann summieren wir die Anzahl der Knoten, die über die Kinder von v, die links von u sind, erreicht werden können. Diese Informationen sind in den jeweiligen

Knoten gespeichert. Somit erhalten wir die Anzahl der Elemente, die echt links des gefundenen Elementes liegen und können damit den Rang bestimmen.

Die Laufzeit ist logarithmisch, da wir einen Pfad im Baum einmal von oben nach unten durchlaufen und dabei konstant viel Zeit pro Schicht benötigen, um die Daten aufzuaddieren. Dies ist dadurch bedingt, das jeder Knoten maximal drei Kinder hat.

d) Es ergeben sich die folgenden Zahlen:

Zahl x	in x oder echt links beginnend	echt links endend
q_1	2	2
q_2	4	2
q_3	6	4

e) Die Zahl x ist genau dann *nicht* in einem Intervall enthalten, wenn die Anzahl der Intervalle, die in x oder echt links davon beginnen, gleich ist wie die Anzahl der Intervalle, die echt links von x enden.

f) Für `insert`$([a, b]\colon \mathbb{N} \times \mathbb{N})$ fügen wir a in T_S und b in T_E ein. Für `remove`$([a, b]\colon \mathbb{N} \times \mathbb{N})$ entfernen wir a aus T_S und b aus T_E. Beachte, dass wir Ranganfragen unterstützen, also die Implementierung der Operationen aus Teilaufgabe b) nutzen.

g) Wir nutzen nun die Ranganfrage, um den Rang des Start- und des Endpunktes zu bestimmen. Somit erhalten wir die Anzahl der Intervalle, die echt links von x beginnen, i_s und die Anzahl der Intervalle, die echt links von x, i_e enden. Falls ein Intervall in x beginnt, inkrementieren wir i_s um eins. Wir prüfen nun, ob $i_s = i_e$. Ist dies der Fall, gibt es kein Intervall, sonst gibt es eines.

Die Laufzeit ist logarithmisch, da zweimal ein Rang und einmal die Existenz eines Elements angefragt wird.

Aufgabe 3 - Von 2-Bienchen und 3-Blümchen - Wie entstehen Bäume? (5 Punkte)

In dieser Aufgabe wollen wir uns mit der Konstruktion von $(2, 3)$-Bäumen beschäftigen.

a) Beschreibe, wie in Linearzeit aus einer gegebenen sortierten Folge ein $(2, 3)$-Baum konstruiert werden kann. Begründe, warum dein Algorithmus das geforderte Laufzeitverhalten hat. (3 Punkte)

b) Beschreibe, wie in Linearzeit zwei gegebene $(2, 3)$-Bäume zu einem $(2, 3)$-Baum zusammengefügt werden können. Begründe, warum dein Algorithmus das geforderte Laufzeitverhalten hat. (2 Punkte)

Hinweis: Achte darauf, dass Werte, die in beiden $(2,3)$-Bäumen vorkommen, nicht verloren gehen.

Lösung 3

a) Sei S die gegebene sortierte Folge. Zunächst fügen wir das Dummy-Element mit Schlüssel ∞ an das Ende von S ein. Nun betrachten wir die Elemente von S in aufeinander folgenden Paaren, d.h. zunächst das erste und das zweite Element, darauf das dritte und das vierte etc. Zu jedem dieser Paare legen wir einen neuen Knoten an, der die beiden Elemente als Kinder erhält. Als Schlüssel erhält der neue Knoten den Wert des größeren der beiden Elemente. Sollte S eine ungerade Anzahl an Elementen enthalten, setzt sich das letzten „Paar" aus drei Elementen, darunter das Dummy-Element, zusammen.
Nun wird die Kontruktion des Baumes rekursiv weitergeführt, bis wir bei der Wurzel angelangt sind. Dazu wird stets zu zwei (bzw. drei) aufeinander folgenden Knoten einer Lage ein Elter-Knoten auf der darüberliegenden Lage erstellt.
Dieser Algorithmus hat eine Laufzeit in $\Theta(n)$, wobei n die Anzahl der Elemente in S ist. Das Anlegen eines Knoten benötigt konstanten Zeitaufwand. Auf der i-ten Lage des Baumes, wobei der Wurzelknoten auf Lage 0 liegt, müssen maximal $\frac{n}{2^{\lceil \log n \rceil - i}}$ Knoten angelegt werden, d.h. es ergibt sich ein Gesamtaufwand von

$$\sum_{i=0}^{\lceil \log n \rceil} \frac{n}{2^i} \cdot \Theta(1) \in \Theta(n)$$

Der Wert der Summanden nimmt exponentiell ab, weswegen die Summe vom ersten Summanden, also n, dominiert wird. Damit ergibt sich diese Laufzeitabschätzung.

b) Seien n und m die Anzahl der Elemente, die in den beiden $(2,3)$-Bäumen enthalten sind. Wir nutzen aus, dass beide $(2,3)$-Bäume bereits sortierte Folgen darstellen. Diese können wir mit Hilfe von MERGE in $\Theta(n+m)$ zu einer sortierten Folge zusammenfügen. Anschließend nutzen wir den Algorithmus aus der ersten Teilaufgabe, um auf der entstandenen Folge einen $(2,3)$-Baum zu konstruieren. Insgesamt benötigen wir dazu Zeit in $\Theta(n+m)$.

Aufgabe 4 - Zum Knobeln (0 Punkte + 3 Bonus)

In der Vorlesung wurde erwähnt, dass Suchbäume eine Art eierlegende Wollmilchsau unter den Datenstrukturen sind. In dieser Aufgabe sollst du diese Behauptung unterstreichen, indem du eine Priority-Queue der Größe n mit $(2,3)$-Bäumen umsetzt. In dieser Aufgabe, darfst du nur die in der Vorlesung vorgestellten Methoden `find`(x), `insert`(x) und `remove`(x) auf $(2,3)$-Bäumen aufrufen. Du sollst nun angeben, wie du die Operationen `push`, `popMin` und `decPrio` einer Priority-Queue in Zeit $\mathcal{O}(\log(n))$ umsetzen kannst.

Du darfst davon ausgehen, dass wir die natürlichen Zahlen $\mathbb{N}_0$ als Prioritäten verwenden und nur natürliche Zahlen in der Datenstruktur speichern. Um die Aufgabe einfacher zu halten, darfst du außerdem annehmen, dass gespeicherte Elemente und ihre Prioritäen identisch sind. Somit fügt `push`(x) die Zahl x in die Priority-Queue ein und `decPrio`(x, k) senkt die Priorität der Zahl x um k.

a*) Beschreibe die Umsetzung der Operationen in Worten. (2 Punkte)

b*) Begründe mit einem Satz, warum es weiterhin sinnvoll ist, binäre Heaps einzusetzen. (1 Punkt)

Lösung 4

a) Wir nutzen einen $(2, 3)$-Baum der Größe n. In diesem speichern wir die Prioriäten. Da diese den gespeicherten Elementen entsprechen, müssen wir die Elemente nicht separat speichern. Die Operation `push` setzen wir durch einen Aufruf von `insert` um. Wir können das Minimum der gespeicherten Werte finden, indem wir im Suchbaum nach der Priorität 0 suchen (also `find`(0) aufrufen). Das liegt an der Tatsache, dass 0 der kleinste Wert des Wertebereichs ist und der Suchbaum den nächst größeren gespeicherten Wert zurückgibt, falls 0 nicht enthalten ist. Somit setzen wir `popMin` um, indem wir das Minimum wie oben beschrieben finden und dann diesen Wert löschen. Anmerkung: Der Aufruf `remove`(0) löscht das Minimum nicht, da die Operation keinen Wert enfternt, falls der übergebene Wert nicht in der Datenstruktur gespeichert ist. Als letztes können wir `decPrio` umsetzen indem wir das Element entfernen und dann mit geänderter Priorität wieder einfügen.

b) Die konstanten Faktoren in der Laufzeit von binären Heaps sind sehr klein. Außerdem sind Heaps viel einfacher zu implementieren.