Zusatzaufgaben 03

Algorithmen I – Sommersemester 2023

Gesamtpunkte: 35

Aufgabe 1 - To negative infinity and beyond! (3 Punkte)

In der Vorlesung wurde der ∞-Trick für (a,b)-Bäume vorgestellt. Beim $-\infty$-Trick hat das Dummy-Element den Wert $-\infty$. Wie unterscheidet sich ein (a, b)-Baum, bei dem der $-\infty$-Trick angewendet wird von einem, bei dem der ∞-Trick angewendet wird? Beachte insbesondere Veränderungen in den Operationen, die wir in der Vorlesung kennengelernt haben. (3 Punkte)

Lösung 1

Seien im folgenden $x_\perp$ das kleinste und $x_\top$ das größte Element im (a,b)-Baum T.

Das Dummy-Element ist nun nicht das Element, welches am weitesten „rechts“, sondern das Element, welches am weitesten „links“ in der Liste von T steht.

- `find`(k):
  Bei Anwendung des ∞-Tricks gibt `find` das kleinste Element größer / gleich k in T zurück, wenn $k > x_\top$, dann also `find`$(k) = \infty$.
  Bei Anwendung des $-\infty$-Tricks gibt `find` das größte Element kleiner / gleich k in T zurück,wenn $k < x_\perp$, dann also `find`$(k) = -\infty$.

- `insert`(k):
  Bei Anwendung des ∞-Tricks wird das neue Element k **vor** dem Element, welches das durch `find`(k) gefunden wird, eingefügt.
  Bei Anwendung des $-\infty$-Tricks wird das neue Element k **nach** dem Element, welches das durch `find`(k) gefunden wird, eingefügt.

Aufgabe 2 - Pseudocode (8 Punkte)

```
FIRST(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ)
    if x < y then
        u: ℕ = SECOND(A, x, y, z)
        z_1: ℕ = x + (z mod (u − x))
        z_2: ℕ = u + 1 + (z mod (y − u − 1))
        FIRST(A, x, u − 1, z_1)
        if u < y then
            FIRST(A, u + 1, y, z_2)
        end
    end
```

```
SECOND(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ): ℕ
    u: ℕ = A[z]
    SWAP(A[y], A[z])
    i: ℕ = x
    j: ℕ = x
    while j < y do
        if A[j] ≤ u then
            SWAP(A[i], A[j])
            i := i + 1
        end
        j := j + 1
    end
    SWAP(A[i], A[y])
    return i
```

1. Seien $A = \langle 4, 8, 12, 34, 19, 26 \rangle$ und $n = 6$. Gib die den Zustand von A nach einem Aufruf von FIRST mit Eingabe $(A, 0, n-1, 1)$ an. (1 Punkt)

2. Was berechnet FIRST für eine beliebiges Array A, was berechnet SECOND? Haben wir diesen Algorithmus schon in der Vorlesung kennengelernt? Wie war sein Name? (2 Punkte)

3. Ändere den Pseudocode von FIRST bzw. SECOND so ab, dass er stabil wird, sich die Laufzeit aber nicht ändert. Du darfst dabei die Funktion

$\textsc{concat}(A1, A2)$ benutzen um zwei Arrays in Linearzeit zu konkatenieren. (3 Punkte)

4. Der Algorithmus $\textsc{First}$ ist in-place, das heißt er benötigt nur konstant viel zusätzlichen Speicherplatz. Wie viel zusätzlichen Speicherplatz braucht dein Algorithmus? Gib im O-Kalkül an. (1 Punkt)

5. Nenne einen Algorithmus der das gleiche tut wie $\textsc{First}$, aber stabil und in-place ist. Was ist seine Laufzeit in O-Kalkül? (1 Punkt)

Lösung 2

1. $A = \langle 4, 8, 12, 19, 26, 34 \rangle$

2. Der Algorithmus $\textsc{First}$ beschreibt $\textsc{Quicksort}$, sortiert also ein Array A, $\textsc{Second}$ partitioniert dabei das Array in kleinere/größere Element

3.

```
SECOND(A: [ℕ; n], x: ℕ, y: ℕ, z: ℕ): ℕ
    B, C: [ℕ; n]
    idxB, idxC: ℕ = 0, 0
    for i ∈ {x, ..., y} do
        if A[i] ≤ A[z] then
            B[idxB] := A[i]
            idxB := idxB + 1
        else
            C[idxC] := A[i]
            idxC := idxC + 1
        end
    end
    A[x ..., y] := CONCAT(B[0 ... idxB], C[0 ... idxC])
    return x + idxB
```

4. Er braucht $\mathcal{O}(n)$ zusätzlichen Speicherplatz.

5. Der Algorithmus ist $\textsc{InsertionSort}$ und seine Laufzeit ist $\mathcal{O}(n^2)$.

Aufgabe 3 - Min-Heaps (11 Punkte)

1. Folgende Arrays stellen Max-Heaps dar. Falls die Heap-Eigenschaft verletzt wurde, gib an wo, wenn nicht, zeichne die Baumrepräsentation des Heaps. (2 Punkte)
   a) $\langle 56, 47, 56, 10, 20, 50, 51, 1, 2, 3, 18 \rangle$
   b) $\langle 56, 56, 47, 10, 20, 50, 51, 1, 2, 3, 18 \rangle$
   c) $\langle 56, 47, 56, 10, 51, 20, 50, 1, 2, 3, 18 \rangle$
   d) $\langle 56, 47, 56, 10, 5, 20, 50, 1, 2, 3, 18 \rangle$

2. Erstelle aus Array $A = \langle 7, 8, 1, 2, 6, 7, 5 \rangle$ einen binären Min-Heap mit BUILD aus der Vorlesung. Gib dabei das Array an, sobald sich Einträge darin ändern. Du musst nur die vertauschten Zahlen angeben, bei allen anderen wird angenommen, dass sie gleich bleiben. (2 Punkte)

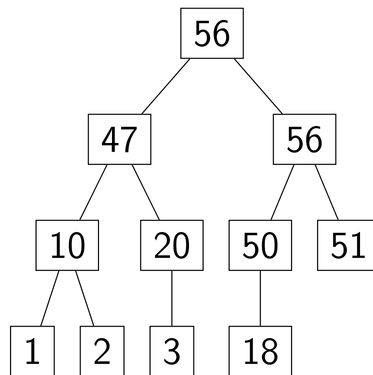
3. Füge in den Min-Heap$A = \langle 2, 7, 3, 9, 7, 5, 5 \rangle$ mit Hilfe von PUSH zuerst ein Element mit Priorität 4 und anschließend ein Element mit Priorität 1 ein. Gib deinen Heap nach jeder Einfügeoperation an. (2 Punkte)

4. Lösche aus dem Min-Heap $A = \langle 2, 7, 3, 9, 7, 5, 5 \rangle$ mit Hilfe von POPMIN zwei Mal das kleinste Element. Gib das Array nach jeder Operation an. (2 Punkte)

5. Man kann ein Array A sortieren, indem man mit BUILD einen Heap aus A ausbaut und mit POPMIN wiederholt das Minimum entfernt. Dieses Sortierverfahren heißt HEAPSORT und ist im Allgemeinen nicht stabil.
   a) Gib ein Beispiel an, bei dem die Stabilität eines Arrays verloren geht. (1 Punkt)
   b) Beschreibe, wie man ohne die Laufzeit oder Funktionsweise von HEAPSORT zu verändern Stabilität erzwingen kann. Du hast dazu $\mathcal{O}(n)$ zusätzlichen Speicher gegeben. (2 Punkte)

Lösung 3

1. a) ist ein Max-Heap:



   b) ist kein Max-Heap, da $47 < 50$ und $47 < 51$

   c) ist kein Max-Heap, da $37 < 51$

   d) ist kein Max-Heap, da $5 < 18$

2. - $\langle 7, 8, 1, 2, 6, 7, 5 \rangle$
   - $\langle 7, 2, 1, 8, 6, 7, 5 \rangle$
   - $\langle 1, 2, 7, 8, 6, 7, 5 \rangle$
   - $\langle 1, 2, 5, 8, 6, 7, 7 \rangle$

3. - Nach Einfügen von 4: $\langle 2, 4, 3, 7, 7, 5, 5, 9 \rangle$
   - Nach Einfügen von 1: $\langle 1, 2, 3, 4, 7, 5, 5, 9, 7 \rangle$

4. - Nach dem ersten Mal popMin: $\langle 3, 7, 5, 9, 7, 5 \rangle$
   - Nach dem zweiten Mal popMin : $\langle 5, 7, 5, 9, 7 \rangle$

5. a) Ein Gegenbeispiel ist $\langle 1, 2, 2 \rangle$

   b) Vor dem Start des Algorithmus bilden wir jeden Eintrag $A[i]$ auf ein Tupel $(A[i], i)$ ab. Das hat einen Zeitverbrauch von $\mathcal{O}(n)$, da man nur das Array von vorne nach hinten durchlaufen muss. Zusätzlich speichern wir nur $\mathcal{O}(n)$ zusätzliche Zahlen. Falls während des Sortierens zwei Elemente, also Tupel, die gleiche erste Komponente haben, dann sei das Element mit kleinerer zweiten Komponente, das also zuerst im Array stand, kleiner.

Aufgabe 4 - Stamm-Haltung (9 Punkte)

Auf dem ersten Zusatzblatt wurde beschrieben, dass Dr. Meta jeden Biberfreund und -feind sehr genau im Auge behält. Hierfür möchte er eine Datenbank D erstellt bekommen, welche es ihm ermöglicht alle gesammelten Informationen zu den ihm bekannten Bibern zu sammeln. Ein Eintrag in D soll hierbei ein k-Tupel sein, zum Beispiel (Name, Alter, Beruf, Sektion, verdächtig, ...). Eine Zeile in dieser Datenbank ist also ein k-Tupel und eine Spalte i stellt den i-ten Eintrag von allen Tupeln dar. Die Anzahl der k-Tupel in D wird mit n bezeichnet.
Da Dr. Meta immer genauestens informiert sein möchte, greift er sehr häufig auf diese Datenbank zu, um sich Einträge nach einer bestimmten Spalte gefiltert anzusehen.

1. Beschreibe eine Methode FIND(`filter`, `zeile`), welche in $\mathcal{O}(\log(m) + j)$ alle k-Tupel ausgibt, die in der Zeile `zeile` den Eintrag `filter` haben. Dabei bezeichnet m die Anzahl verschiedener Einträge in dieser Zeile und j die Anzahl der k-Tupel mit dem Eintrag `filter`. Beschreibe zusätzlich hierfür eine Datenstruktur, auf welcher die Laufzeit von FIND gewährleistet werden kann. (5 Punkte)

2. Beschreibe in Abhängigkeit von n, m und j eine möglichst laufzeiteffiziente Methode INSERT(`tupel`), welche ein k-Tupel der Datenstruktur hinzufügt. Gib dessen Laufzeit so genau wie möglich an und begründe diese. (2 Punkte)

3. Begründe warum deine INSERT Methode keine bessere Laufzeit haben kann. Gehe hierbei auf die von dir gewählte Datenstruktur sowie Funktionsweise von FIND ein. (2 Punkte)

Lösung 4

1. Als Datenstruktur verwenden wir eine modifizierte Liste, in welcher wir pro Listenelement als Datum ein k-Tupel speichern. Zusätzlich dazu hat jedes Listenelement pro Spalte zwei Pointer, ähnlich einer doppelt verketteten Liste. Jede dieser Spaltenpointer ermöglichen die Handhabung der modifizierten Liste wie die doppelt verkettete Liste bei der Standardversion des ab-Baums aus der Vorlesung.

Auf dieser Datenstruktur wird für jede Spalte separat ein ab-Baum erzeugt. Als Keys verwendet der ab-Baum für Spalte i den i-ten Eintrag eines k-Tupels. Die Blätter zeigen immer auf das erste Vorkommen von gleichen Einträgen. k-Tupel mit gleichen Einträgen stehen, erreichbar über die i-ten Spaltenpointer, immer hintereinander.
Die Methode FIND funktioniert ähnlich zu unserer Version aus der Vorlesung. Wir suchen im i-ten ab-Baum mit FIND aus der VL das erste Element $\geq$ `filter`. Von dort aus iterieren wir über unsere Liste mittels den i-ten Spaltenpointer, um alle k-Tupel mit i-ten Eintrag gleich `filter` auszugeben. Sollte das von FIND gefundene k-Tupel nicht den Eintrag `filter` beinhalten, hört unsere Methode auf.
Da jeder unterschiedliche Eintrag einer Spalte nur einmal in dem entsprechenden ab-Baum beinhaltet ist, hat dieser die Höhe $\log(m)$. Da wir FIND aus der VL verwenden hat diese eine Laufzeit von $\mathcal{O}(\log(m))$. Zusätzlich iterieren wir bis zu j-1 Elemente weiter, um jedes der k-Tupel mit gleichem Eintrag auszugeben. Dies läuft in $\mathcal{O}(j)$. Insgesamt hat also unsere neue FIND Methode eine Laufzeit von $\mathcal{O}(\log(m) + j)$

2. Bei INSERT wird für das einzufügende k-Tupel t ein neues Listenelement erzeugt. Hierfür müssen alle Spaltenpointer richtig gesetzt werden. Wir rufen also auf jedem ab-Baum FIND mit dem i-ten Eintrag $filter$ in t aus der VL auf, um das erste k-Tupel mit einem Eintrag an Stelle $i \geq filter$ zu finden. Dann wird das neue Listenelement vor diesem Element eingefügt, indem die i-ten Spaltenpointer korrekt gesetzt werden.
Wir rufen insgesamt n-mal FIND aus der Vorlesung auf. Die Laufzeit ist dementsprechend $\sum_{l=1}^{k}(\log(m_l)) \leq k * \log(\text{MAX}(m_1, \ldots, m_k))$. Dies ist die Laufzeit von unserer Methode, da pro ab-Baum nur konstant viele Zeigeroperationen durchgeführt werden.

3. Damit FIND eine Laufzeit von $\mathcal{O}(\log(m) + j)$ hat, darf kein Spalteneintrag in einer Spalte mehrfach vorkommen. Sonst wäre die Höhe des ab-Baums $\log(n)$. Einen großen ab-Baum für alle Spalten zu verwenden ist für FIND ineffizienter, da dessen Höhe in $\Theta \log(k * n)$ liegt. Somit müssen k-Tupel mit gleichen Einträgen nach konstant vielen FIND aufrufen ausgegeben werden. Dies geht nur dann, wenn diese in einer Datenstruktur an berechenbaren, in konstanter Zeit erreichbaren Stellen

stehen. Eine Liste wie oben beschrieben eignet sich dafür.
Ein Aufruf von INSERT muss somit k verschiedene ab-Bäume anpassen. Deshalb hat INSERT auf der gewählten Datenstruktur keine effizientere Laufzeit.

Aufgabe 5 - Kleinaufgaben (4 Punkte)

1. Ordne die folgenden Algorithmen absteigend nach ihrer Worst-Case Laufzeitkomplexität: Quicksort, Bucketsort, Mergesort (1 Punkt)

2. Wie lautet die untere Schranke für die Worst-Case Laufzeitkomplexität bei vergleichsbasiertem Sortieren? Gilt diese untere Schranke auch für Sortieralgorithmen für ganze Zahlen? Begründe. (1 Punkt)

3. Was macht ein stabiles Sortierverfahren aus? Nenne ein Beispiel für ein stabiles Sortierverfahren. (1 Punkt)

4. Nenne die Heap-Eigenschaft bezüglich eines Arrays $A[0, \ldots, n-1]$, welches implizit einen binären Heap darstellt. (1 Punkt)

Lösung 5

1. Quicksort ($\mathcal{O}(n^2)$) $>$ Mergesort ($\mathcal{O}(n \log n)$) $>$ Bucketsort($\mathcal{O}(n)$).

2. Die untere Schranke für vergleichsbasiertes Sortieren ist $\mathcal{O}(n \log n)$. Diese gilt nicht beim ganzzahligen Sortieren. Dort kann man bei guter Eingabeverteilung ziffernweise in ($\mathcal{O}(n)$ sortieren.

3. Die relative Position gleich großer Elemente zueinander bleibt unverändert. Beispiel: Insertionsort, Mergesort, Bubblesort, LSD-Radixsort

4. $\forall i > 0 : A[\lfloor \frac{i-1}{2} \rfloor] \leq A[i]$