Algorithmen und Datenstrukturen 1

ALGO1 · SoSe-2023 · `tcs.uni-frankfurt.de/algo1/` · 2023-07-06 · *ed6968a*

Prioritätswarteschlangen, Heaps (Woche 7)

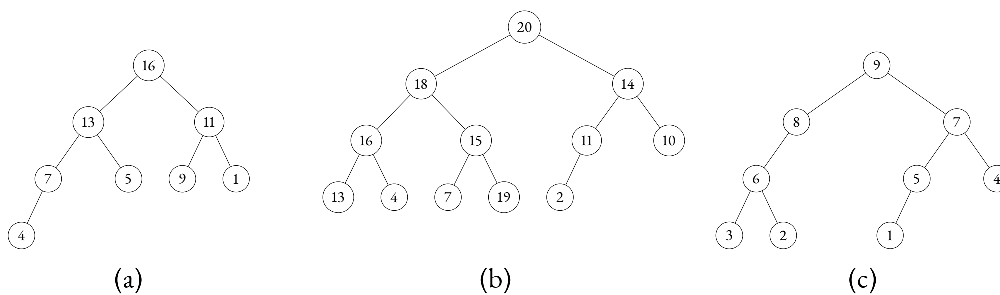
Eigenständige Vorbereitung:
Lies 📖 CLRS Kapitel 6, sowie Appendix B.5 und schau dir das 📺 Video der Woche an.

Zeichenlegende:

- ✏️ Schriftliche Aufgabe, die du fristgerecht in Moodle abgibst. In der Klausur wirst du alle Aufgaben schriftlich bearbeiten, daher ist das Feedback der Tutoren wichtig, damit du deine Schreibfähigkeiten verbessern kannst.
- 👍 Diese Art von Aufgabe musst du sicher können, um die Klausur zu bestehen.
- 🔑 Diese Art von Aufgabe musst du weitgehend können, um die Klausur zu bestehen.
- 🔭 Diese Art von Aufgabe musst du können, um eine gute Note zu erhalten.
- 🌈 Diese Aufgabe ist als Knobelspaß gedacht, der das algorithmische Verständnis vertieft.

Aufgabe 7.1 (Heap-Eigenschaften). Löse die folgenden Teilaufgaben.

a) 👍 Welche der folgenden Bäume erfüllen die Heap-Eigenschaft?



b) 👍 Welche der durch folgende Felder repräsentierten Bäume erfüllen die Heap-Eigenschaft? Index 0 wird nicht benutzt und ist deshalb mit – markiert.

$$A = [-, 9, 7, 8, 3, 4] \qquad B = [-, 12, 4, 7, 1, 2, 10] \qquad C = [-, 5, 7, 8, 3]$$

c) 👍 Sei $S = 4, 8, 11, 5, 21, \star, 2, \star$ eine Sequenz von Operationen, wobei eine Zahl für das Einfügen dieser Zahl in den Heap steht und $\star$ für eine `ExtractMax` Operation. Wie sieht der Heap H nach jeder einzelnen Operation aus, wenn H anfangs leer ist?

d) 👍 Erfüllt ein absteigend sortiertes Feld die Heap-Eigenschaft?

e) 👍 Wo befindet sich in einem (Max-)Heap das kleinste Element?

f) 🔑 Zeige, dass `Insert`, `ExtractMax` und `IncreaseKey` die Heap-Eigenschaft aufrechterhalten.

g) 🔭 Angenommen wir erhalten k absteigend sortierte Felder mit **insgesamt** n Elementen als Eingabe. Zeige, wie sich alle k Felder mit Hilfe eines Max-Heaps in Zeit $O(n \log k)$ zu einem einzelnen absteigend sortierten Feld der Länge n verflechten lassen.

Aufgabe 7.2 (Priogruppen-Politik 👍). Die Kakistokratische Partei will deine Hilfe, um ihre neue „Frischluft“-Politik umzusetzen. Entwirf ein Bürgerregister, das alle Bürger:innen und ihre Gehälter so speichert, dass man die Person mit dem geringsten Einkommen möglichst schnell finden und ausbürgern kann.

Das System soll die folgenden Operationen unterstützen:

- `Insert(c,i)` fügt eine Person mit der Sozialversicherungsnummer c und dem jährlichen Gehalt i ein.
- `DeportLowestIncome()` Gibt die Person mit dem niedrigsten Einkommen aus und entfernt sie aus dem System.

Entwirf eine möglichst effiziente Datenstruktur, die das System implementiert.

Aufgabe 7.3 (Operationen für Prioritätswarteschlangen). Wir erinnern uns, dass Prioritätswarteschlangen die Operationen `Max()`, `ExtractMax()`, `IncreaseKey(`x, k`)` und `Insert(`x`)` unterstützen. Wir wollen nun weitere Operationen zur Verfügung stellen, und zwar die Folgenden:

- `RemoveLargest(`m`)` entfernt das mt-größte Element der Prioritätswarteschlange.
- `Delete(`x`)` entfernt Element x aus der Prioritätswarteschlange.
- `Fusion(`x, y`)` entfernt Elemente x und y aus der Prioritätswarteschlange und fügt ein neues Element z mit Schlüssel $x.\text{key} + y.\text{key}$ ein.
- `FindLarger(`k`)` gibt all jene Elemente der Prioritätswarteschlange aus, deren Schlüssel mindestens so groß wie k ist.
- `ExtractMin()` gibt das Element der Prioritätswarteschlange mit dem kleinsten Schlüssel aus und entfernt es.

Wir wollen diese Operationen effizient implementieren, ohne dass sich die Komplexität der Standardoperationen `Insert`, `IncreaseKey`, `Max` und `ExtractMax` ändert.

Sei n die Anzahl der Elemente in der Prioritätswarteschlange. Löse die folgenden Teilaufgaben:

a) 🔑 Erkläre wie sich `RemoveLargest(`m`)` mit Zeitbedarf $O(m \log n)$ implementieren lässt.

b) 🔑 Erkläre wie sich `Delete(`x`)` und `Fusion(`x, y`)` mit Zeitbedarf $O(\log n)$ implementieren lässt.

c) 🔑 Erkläre wie sich `FindLarger(`k`)` mit Zeitbedarf $O(m)$ implementieren lässt, wobei m die Anzahl der Elemente mit Schlüssel $\geq k$ ist.

d) 🔭 Erkläre wie sich `ExtractMin()` mit Zeitbedarf $O(\log n)$ implementieren lässt.

Aufgabe 7.4 (Zusätzliche Daten 🔑). Sei $A[1..n]$ ein als Feld gespeicherter Heap. Jedes Element `x` in dem Heap wird durch einen Index i repräsentiert und hat einen Schlüssel `x.key`, der als $A[i]$ gespeichert ist. Es ist oftmals nützlich, zusätzliche Daten `x.data` zu speichern, die mit einem Element `x` assoziiert sind. Modifiziere die Datenstruktur so, dass eine neue Operation `Data(`i`)` in Zeit $O(1)$ die zusätzlichen Daten des Elements mit Index i zurückliefert. Hierbei dürfen sich die asymptotischen Laufzeiten der Standardoperationen des Heaps nicht verändern.

Aufgabe 7.5 (Eigenschaften von Heaps 🔑). Sei $T = (V, E)$ ein **vollständiger** Binärbaum von Höhe h. Löse die folgenden Teilaufgaben, die für die Laufzeitanalyse von `heapify` benötigt werden.

a) Zeige, dass für die Anzahl an Knoten $|V| = 2^{h+1} - 1$ gilt.
Hinweis: Begründe, dass $|V| = 1 + 2 + 4 + \ldots + 2^h$ gilt und betrachte diesen Wert als Binärzahl.

b) Zeige: Für die Summe S mit $S = n/4 \cdot 1 + n/8 \cdot 2 + n/16 \cdot 3 + n/32 \cdot 4 + \ldots$ gilt $S = \Theta(n)$.
Hinweis: Berechne $S - S/2$

Aufgabe 7.6 (Summen). Sei $A[0..n-1]$ ein Feld von ganzen Zahlen. Wir interessieren uns für die folgenden Operationen:

- `Sum`(i, j) gibt $A[i] + A[i+1] + \ldots + A[j]$ aus.
- `Change`(i, x) setzt $A[i]$ auf den Wert x.

Löse die folgenden Teilaufgaben:

a) 👍 Entwirf eine Datenstruktur, die `Sum` mit $O(1)$ Zeit und $O(n^2)$ Platz unterstützt.
b) 🔑 Entwirf eine Datenstruktur, die `Sum` mit $O(1)$ Zeit und $O(n)$ Platz unterstützt.
c) 🔭 (sehr schwer) Entwirf eine Datenstruktur, die `Sum` und `Change` beide mit $O(\log n)$ Zeit und $O(n)$ Platz unterstützt.

Aufgabe 7.7 (Sitze in einem Parlament ✏️ 🔑). Schreibe ein Programm in C/C++, Java, Python oder einer anderen gängigen Programmiersprache (kein Pseudocode), welches das folgende Problem löst: Verteile die m Sitze in einem Parlament nach einer Wahl auf n Parteien.

Die Platzvergabe verläuft nach dem *D'Hondt-Verfahren*: für $i \in \{1, \ldots, n\}$ bezeichne $v_i \in \mathbb{N}$ die Anzahl der Stimmen für Partei i. Für jede Partei i wird ein Quotient q_i berechnet, welcher anfangs auf $q_i := v_i/1$ gesetzt wird. Hat Partei j den größten Quotienten, wird ihr ein Sitz zugeteilt. Anschließend wird ihr Quotient folgendermaßen aktualisiert:

$$q_j := \frac{v_j}{s_j + 1},$$

wobei s_j die Anzahl der Sitze, welche bisher Partei j zugeordnet wurden, bezeichnet. Anfangs wird die Anzahl der zugeordneten Sitze für alle Parteien auf 0 gesetzt. Dieser Vorgang wird wiederholt, bis alle m verfügbaren Sitze vergeben sind.

Eingabe. Die Datei besteht aus mehreren Zeilen. In der ersten Zeile sind n und m durch ein Leerzeichen getrennt gegeben. Beide Zahlen sind in der Menge $\{1, \ldots, 2\,000\,000\}$ enthalten. In der i-ten der n darauffolgenden Zeilen ist die ganze Zahl v_i gegeben. Jede Partei erhält mindestens eine Stimme. Die Anzahl aller Stimmen ist mindestens so groß wie die Anzahl der zu verteilenden Sitze, es gilt also $v_1 + \cdots + v_n \geq m$. Unsere Eingaben sind so konstruiert, dass der letzte Sitz eindeutig zugeordnet werden kann – es muss keine Logik eingebaut werden, welche bei Gleichstand entscheidet.

Ausgabe. Die Sitzverteilung, wobei sich die Reihenfolge in der Ausgabe mit der Reihenfolge in der Eingabe vertragen soll. Betrachte hierzu die Beispiele.

Beispiele.

`1.in`
```
2 2
10
10000000
```

`1.ans`
```
0
2
```

`2.in`
```
2 3
12
11
```

`2.ans`
```
2
1
```

`3.in`
```
2 4
12
11
```

`3.ans`
```
2
2
```

`4.in`
```
2 4
17
10
```

`4.ans`
```
3
1
```

Erklärung zu Beispiel 4. Es traten 2 Parteien zur Wahl an, und insgesamt sollen 4 Sitze vergeben werden. Partei 1 erhält den ersten Sitz, da sie die meisten Stimmen erhalten hat. Anschließend wird ihr Quotient auf $q_1 = \frac{17}{2}$ gesetzt. Partei 2 erhält den nächsten Sitz, denn $10 > \frac{17}{2}$. Anschließend wird ihr Quotient auf $q_2 = \frac{10}{2} = 5$ gesetzt. Der dritte Sitz geht an Partei 1, denn es gilt $q_1 = \frac{17}{2} > 5 = q_1$. Der Quotient von Partei 1 wird anschließend auf $q_1 = \frac{17}{3}$ gesetzt. Auch der letzte Sitz geht an Partei 1, denn es gilt $q_1 = \frac{17}{3} > 5 = q_2$. Insgesamt erhält Partei 1 also drei Sitze und Partei 2 erhält einen.

`5.in`

```
4 14
38
35
36
37
```

`5.ans`

```
4
3
3
4
```

Größere Beispiele. Teste dein Programm! Hier sind größere Beispiele: https://files.tcs.uni-frankfurt.de/algo1/seatallocation-tests.zip. Stell sicher, dass dein Programm für alle Eingaben `X.in` *exakt* die entsprechende Ausgabedatei `X.ans` erzeugt.

Tipps. Achte auf Rundungsfehler und möglichen Überlauf von `ints`. Es ist keine Überraschung, dass diese Aufgabe mit einer Prioritätswarteschlange gelöst werden soll.

Hinweise zur Abgabe. Die Datei `v2-015-secret.ans` fehlt. Deine Abgabe soll die SHA1-Summe der korrekten Datei enthalten. Zum Beispiel erhält man die SHA1-Summe der Datei `v2-014-19.ans` wie folgt:

```
$ sha1sum v2-014-19.ans
22d43373a4104696005cb3db1fa8f3f0c873090a  v2-014-19.ans
```

Deine Abgabe soll wie immer per PDF erfolgen und die grobe Idee, den diesmal echten Code, den Korrektheitsbeweis, die Laufzeitanalyse, und die SHA1-Summe von `v2-015-secret.ans` enthalten. Weiterhin zu beachten:

- Der Code darf maximal **60** Zeilen lang sein (jede Zeile mit maximal 100 Zeichen). Möglichst kurz und elegant! (Eine 20-Zeilen Lösung in Python ist möglich. Kommentare zählen nicht dazu.)
- In Python kann man einen Min-Heap mit dem Modul `heapq` nutzen, siehe z.B. hier: geeksforgeeks.org/heap-queue-or-heapq-in-python/. Um ein Element `x` mit Priorität 7 auf einen heap `heap` einzufügen, verwende beispielsweise `heapq.heappush((7,x), heap)`. Um einen Min-Heap zu einem Max-Heap zu machen, kann man die Vorzeichen der Prioritäten ändern. Falls du deine eigene Implementierung der Prioritätswarteschlange nutzen möchten, zählt diese Implementierung nicht zum Zeilenlimit dazu. Wichtig: Die Datenstruktur darf nur so benutzt werden, wie die in der Vorlesung beschriebene abstrakte Datenstruktur das erlaubt.