Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Zusatzaufgaben 02

Algorithmen I – Sommersemester 2023

Gesamtpunkte: 35

Aufgabe 1 - Datenstrukturen Zuordnen (15 Punkte)

Wir haben im Laufe der Vorlesung mehrere Datenstrukturen kennengelernt. In dieser Aufgabe wollen wir uns auf die folgenden fünf konzentrieren:

- Arrays
- (doppelt-verkettete) Listen
- Stacks
- Queues
- Hashtabellen

Deine Aufgabe ist es nun, jedem der folgenden Szenarien eine dieser Datenstrukturen zuzuordnen, die sich aus deiner Sicht besonders dafür eignet. Begründe dabei kurz deine Wahl.

1. Für ein Programm soll eine Undo/Redo-Funktion entwickelt werden, bei dem die vergangenen Veränderungen gespeichert werden sollen.
2. Für einen Online-Shop soll für ein Produkt (welches durch seine Produktnummer eindeutig identifizierbar ist) vor dem Verkauf überprüft werden, wie viele Exemplare noch vorhanden sind.
3. In einer App gibt es einen Content-Feed, bei dem man nach rechts/ links swipen kann, um aktuellere/ältere Nachrichten zu erhalten.
4. Für eine Veranstaltung mit 5000 durchnummerierten Tickets soll am Einlass kontrolliert werden, welche Tickets bereits entwertet wurden.

5. Für eine Anwendung soll jeweils die letzte noch nicht gelesene Nachricht auf eurem Telefon anzeigt werden.
6. Die Bücher auf einem TBR-Stapel[1] sollen organisiert werden. Dabei ist immer nur das Buch relevant, das ganz oben auf dem Stapel liegt.
7. Während der Sprechstunde sollen die Patienten in einem Wartezimmer verwaltet werden. Die Patienten werden dabei in der Reihenfolge aufgerufen, wie sie in die Praxis gekommen sind.
8. Für eine Anwendung sollen Bewegungen zwischen Zugabteilen bei einem Zug beliebiger Länge simuliert werden. Es soll dabei gespeichert werden, wie viele Personen sich zu einem Zeitpunkt in einem Zugabteil befinden.
9. In einer Bar sind leider schon einige Menschen auffällig geworden und haben Hausverbot erhalten. Um dies festzuhalten wurden ihre den Vor- und Nachnamen und die Ausweisnummer des Personalausweises gepeichert. Für eine Person soll nun festgestellt werden können, ob sie schoneinmal Hausverbot erhalten hat.
10. Für jeden Knoten in einem Graphen soll eine Farbe gespeichert werden.
11. Bei einem rundenbasierten Online-Game ohne feste Spieleranzahl soll die Spieler-Reihenfolge gespeichert werden.
12. Für eine neue Speedrun-Website soll es pro Spiel ein Leaderbord geben, in dem die 10 besten Spieler stehen sollen.
13. Für einen Music-Player soll eine Playlist abgespielt werden. Dabei soll es auch möglich sein, einen vergangenen Song anzuhören oder Songs zu skippen.
14. Zur Ausgabe eines Browserverlaufs sollen die zuletzt besuchten Websiten gespeichert werden.
15. Für eine passwortgeschützte Website sollen für Neukunden Profile angelegt werden. Bei späterem Einloggen soll überprüft werden, ob ein eingegebenes Passwort richtig war. Um Hacking zu vermeiden sollte das Passwort allerdings nicht gespeichert werden.

[1] TBR steht für „to be read“, hier liegen also alle Bücher, die man schon seit Monaten ganz bestimmt sehr bald lesen wird.

16. In einem digitalen Kochbuch sollen die Schritte eines Rezepts gespeichert werden. Dabei wird dem Anwender immer nur der aktuelle Schritt angezeigt.

Lösung 1

Dieser Lösungsvorschlag nennt für jedes Szenario nur eine Datenstruktur. Tatsächlich könnten aber für mehrere verschiedene Datenstrukturen genutzt werden. Hier kommt es darauf an, ob du deine Wahl gut begründen kannst.

1. Wir nehmen eine **Liste**. Sie ist geeignet, um von einem Zustand zu seinem Vorgänger bzw. Nachfolger zu navigieren und um einfach bei Änderungen neue Zustände einfügen zu können, auch nach einer Änderung, die nicht nie neueste ist

2. Sind die Produktnummern dadurch entstanden, dass die Produkte duchnummeriert wurden, bietet sich ein **Array** ein. Sind die Produktnummern beliebige Ziffernfolgen einer festen Länge, verwenden wir eine **Hashtabelle**

3. Auch hier bietet sich eine **Liste** an, um einfach zwischen Nachrichten zu navigieren, neue Nachrichten einzufügen und Nachrichten zu löschen

4. Wir wählen ein **Array**, um einfach die Information für ein beliebiges Ticket zuzugreifen

5. Die Nachrichten können auf einem **Stack** gehalten werden, da stets nur die neueste Nachricht relevant ist

6. Mit der gleichen Begründung wie bei der vorherigen Aufgabe bietet sich hier ein **Stack** an.

7. Die Patienten können in einer **Queue** gespeichert werden; damit werden sie in der gleichen Reihenfolge entfernt, wie sie eingefügt wurden.

8. Mit einer **Liste** können Bewegungen von einem Zugabteil in ein benachbartes einfach dargestellt werden.

9. Die Ausweisnummer ist für jeden Menschen eindeutig, aber wir haben keinen Einfluss auf die Verteilung der Nummern, zu denen wir Informationen speichern müssen. Deswegen setzten wir hier eine **Hashtabelle** ein.

10. Wir können die Knoten durchnummerieren und damit einfach ein **Array** verwenden.

11. Wir können eine **Queue** verwenden, um die Spielerreihenfolge einfach festzuhalten und beliebig viele Spieler in Reihenfolge einfügen zu können

12. Da wir eine feste Anzahl Spieler speichern müssen und die Indizes in dieser Auflistung relevant sind, können wir ein **Array** verwenden.

13. Um einfach zwischen Songs zu navigieren, wählen wir eine **Liste**.

14. Hier bietet sich ein **Stack** an, da wir immer nur die zuletzt besuchte Website einfügen müssen.

15. Wir verwenden eine **Hashtabelle**, um zu überprüfen, ob der Hashwert des eingegebenen und des korrekten Passworts identisch sind, ohne die Passwörter selbst speichern zu müssen.

16. Hier können wir eine **Queue** verwenden, um die Schritte in der korrekten Reihenfolge durchzugehen.

Aufgabe 2 - Immer in Paaren laufen $(7\ \text{Punkte})$

Im Folgenden wollen wir in erwartet Linearzeit Paare von ganzen Zahlen in einem Array finden, die eine gewisse Eigenschaft erfüllen.

1. Beschreibe einen Algorithmus, der für eine gegeben Zahl k und ein Array A mit n Einträgen in erwartet $\mathcal{O}(n)$ Zeit entscheidet, ob es Indizes i, j gibt, sodass $A[i] \cdot A[j] = k$. (2 Punkte)

2. Beschreibe einen Algorithmus, der für eine gegebene Zahl k und ein Array A mit n Einträgen in erwartet $\mathcal{O}(n)$ Zeit entscheidet, ob es Indizes i, j gibt, sodass $\frac{2 \cdot A[i]}{3} + 4 \cdot A[j] + 5 = k$. (2 Punkte)

*. Finde ein generelles Verfahren, womit wir für eine gegebene Zahl $k \in \mathbb{Z}$, ein Array A mit n Einträgen und eine Funktion $f(x, y) : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ in erwartet $\mathcal{O}(n)$ entscheiden können, ob es zwei Indizes i, j gibt, sodass $f(A[i], A[j]) = k$. Welche Anforderungen muss die Funktion f erfüllen? (3 Punkte)

Lösung 2

1. Wir legen eine Hashmap der Größe n an, welche ganze Zahlen enthält. Wir fügen nun die Elemente aus A ein, hierbei verwenden wir für einen Eintrag $a = A[l]$ den Schlüssel a zum Wert l.
   Anschließend überprüfen wir für jeden Eintrag $b = A[i]$ ob der es einen Eintrag j zum Schlüssel $\frac{k}{b}$ in der Hashmap gibt. Sollte dies der Fall sein, so ist i, j so ein gewünschtes Paar.
   Dieses Verfahren fügt n Elemente in erwartet $\Theta(n)$ in die Hashmap ein und fragen dann maximal n mal jeweils in erwartet $\Theta(1)$ einen Wert in der Hashmap ab, also liegt die Gesamtlaufzeit in erwartet $\Theta(n)$.

2. Wieder legen wir eine Hashmap der Größe n an, welche ganzen Zahlen enthält und fügen die Elemente aus A ein wie bei der vorherigen Teilaufgabe.
   Nun überprüfen wir für jeden Eintrag $b = A[l]$ ob es einen Eintrag j zum Schlüssel $\frac{3}{2}(k - 5 - 4b)$ gibt. Sollte dies der Fall sein, so ist i, j ein gesuchtes Paar.
   Auch hier ist die Laufzeitabschätzung die gleiche, da wir die Umkehrfunktion in $\Theta(1)$ berechnen können.

*. Damit das Verfahren aus den vorherigen Teilaufgaben funktioniert, muss die Funktion f auf einem der beiden Eingaben in konstanter Zeit invertierbar sein. Invertierbar heißt hier, dass für ein $c \in \mathbb{Z}$ fest, entweder $\{x \in \mathbb{Z} \mid f(x, c) = k\}$ in $\Theta(1)$ oder $\{x \in \mathbb{Z} \mid f(c, x) = k\}$ in $\Theta(1)$ berechenbar sein muss. Zusätzlich dürfen nur konstant viele $x \in \mathbb{Z}$ diese Bedingung erfüllen, denn wir müssen im schlimmsten Fall alle testen um unser Gegenstück zu finden.
   Dann können wir das Verfahren aus den vorherigen Teilaufgaben anwenden. Wir legen eine Hashmap der Größe n an, welche ganze Zahlen enthält. Wir fügen die Elemente aus A ein. Für einen Eintrag $b = A[l]$

fügen wir den Wert l zum Schlüssel b ein.
Nun überprüfen wir für jeden Eintrag $c = A[i]$, ob es einen Wert j zu einem Schlüssel aus $\{x \in \mathbb{Z} \mid f(x, c) = k\}$ oder $\{x \in \mathbb{Z} \mid f(c, x) = k\}$ in der Hashmap gibt. Ist dies der Fall, dann ist i, j ein solches gewünschtes Paar.
Wie in der vorherigen Teilaufgaben fügen wir n Elemente in eine Hashmap in erwartet $\Theta(n)$ ein. Anschließend überprüfen wir für maximal n Elemente, ob es einen entsprechenden Partner gibt. Da es für jedes Element nur $\Theta(1)$ Partner gibt und wir diese in $\Theta(1)$ berechnen können und jedes dieser potentiellen Partner in erwartet $\Theta(1)$ abfragen können, liegt die ganze Abfrage in erwartet $\Theta(n)$. Damit liegt unser Verfahren insgesamt in erwartet $\Theta(n)$

Aufgabe 3 - Familientreffen $(4\ \text{Punkte})$

In dieser Aufgabe wollen wir möglicherweise universelle Hashfamilien untersuchen. Sei dafür $M \in \mathbb{N}$ und $U = \{0, 1, \ldots, M-1\}$ unser Universum an Schlüsseln und $m \ll M$ die Größe unserer Hashtabelle.

1. Ist die folgende Menge an Funktionen eine universelle Familie an Hashfunktionen? (2 Punkte)

$$\mathcal{F} = \{f : x \mapsto a^x \bmod a \bmod m \mid a \in U\}$$

2. Ist die folgende Menge an Funktionen eine universelle Familie an Hashfunktionen? (2 Punkte)

$$\mathcal{H} = \{h : x \mapsto (ax + 5) \bmod m \mid a \in U\}$$

Lösung 3

1. Leider ist $\mathcal{F}$ keine universelle Hashfamilie. Bis auf $x = 0$ werden alle Werte bereits nach $a^x \bmod a$ auf 0 abgebildet und somit ist z.B für $k_1 = 1$ und $k_2 = 2$ $\mathsf{Pr}[f(k_1) = f(k_2)] = 1$.

2. Auch $\mathcal{H}$ ist leider keine universelle Hashfamilie. Denn für ein beliebiges

$k_1 \in U$ und $k_2 = k_1 + m$ ist für ein zufälliges $h \in \mathcal{H}$:

$$\begin{aligned} h(k_2) &= (a(k_1 + m) + 5) \bmod m \\ &= (ak_1 + am + 5) \bmod m \\ &\equiv (ak_1 + 5) \bmod m \\ &= h(k_1) \end{aligned}$$

Aufgabe 4 - Auf heißer Spur (9 Punkte)

Alle Tiere sind vereinigt im Aufstand gegen den hinterhältigen Dr. Meta.
Alle Tiere? Nein! Eine Hand voll Dachse steht immer noch an der Seite des Superschurken (vermutlich diejenigen, die hoffen, irgendwann von ihm noch ihre Smartphones zurück zu bekommen). Natürlich hat Dr. Meta Wind von der Intrige in den Alpen bekommen. Außer sich vor Wut schickt er sämtliche ihm verbleibenden Lakaien auf die Baustelle, um die Fertigstellung des Damms zu sabotieren.
Dr. Meta hat jedoch die Rechnung ohne die Seeadler gemacht. Sie fliegen unablässig Patrouille und notieren sich verdächtige Dachse. Sie haben nur ein Problem: Selbst die auf höchstem Niveau ausgebildeten Agenten des NSA können Dachse nicht voneinander unterscheiden. Otter, Dachs, Biber. . . die sehen doch alle gleich aus mit ihrem Fell! In den Augen der Adler sind sie alle ~~Beute~~ Säugetiere. Um herauszufinden, welchen Dachs er gesichtet hat, muss ein Seeadler also den Verdächtigten befragen.
Das nutzen die von Dr. Meta eingeschleusten Dachse geschickt aus: Werden sie von verschiedenen Seeadlern nach ihrem Namen gefragt, können sie verschiedene Antworten geben. Zwei Seeadler können den gleichen Dachs also unter verschiedenen Namen kennen. Um nicht zu sehr aufzufallen, benutzen verschiedene Dachse aber nie den gleichen Decknamen.
Doch der NSA wäre nicht der NSA, könnte er damit nicht umgehen. Die Störenfriede müssen schnellstmöglich identifiziert und von der Baustelle entfernt werden!
Aktuell liegen dem NSA sämtliche Sichtungen von verdächtigen Dachsen in Form einer Liste `sightings` der Länge n vor. Ein Eintrag (t, c, d) in `sightings` drückt aus, dass zum Zeitpunkt t an Koordinate c ein Dachs gesichtet wurde, der behauptet hat, den Decknamen d zu tragen. Damit wis-

sen die Adler: Gibt es zwei Einträge $(t, c, d_1), (t, c, d_2)$ in `sightings`, sodass $d_1 \neq d_2$, so war der gesichtete Dachs ein Spion, der zwei Adler-Agenten unterschiedliche Namen genannt hat.
Zunächst will der NSA wissen, welche Decknamen überhaupt in Umlauf sind.

1. Beschreibe, wie eine Liste sämtlicher Decknamen in `sightings` erstellt werden kann, die keine Duplikate enthält. Dein Algorithmus soll dabei eine asymptotische Laufzeit haben, die besser als $\Theta(n^2)$ ist. (2 Punkte)

Das ist doch gleich viel übersichtlicher! Die Adler sind jedoch nicht unfehlbar (auch wenn sie das nie zugeben würden) und befragen auch unschuldige Dachse, deren Namen damit auch in `sightings` auftauchen. Welche Decknamen gehören nun zu welchem Spitzel? Und wie viele Maulwürfe sind tatsächlich unter den Dachsen?

2. Wir wollen im Folgenden einen Algorithmus entwerfen, der die Namen in `sightings` zu Dachsen zuordnet. Welche Datenstruktur eignet sich am Besten, um die Namen dazu zu verwalten? Begründe deine Wahl. (1 Punkt)

3. Beschreibe einen Algorithmus, der für alle gesichteten Dachse die Menge der von ihnen angegebenen Namen bestimmt. (3 Punkte)

4. Gib die asymptotische Laufzeit deines Algorithmus aus Teilaufgabe 3 an und begründe deine Antwort. (2 Punkte)

5. Beschreibe einen Algorithmus, der die Anzahl an Dachsen bestimmt, die auf jeden Fall zu Dr. Metas Spionen gehören. Die Laufzeit dieses Algorithmus sollte nicht schlechter sein als die deines Algorithmus aus Teilaufgabe 3. (1 Punkt)

Lösung 4

1. Wir können zunächst in $\Theta(n)$ alle Decknamen in `sightings` in einer Liste `names` sammeln und anschließend in $\Theta(n \log(n))$ (lexikografisch) sortieren. Nun können wir ein Mal über `names` iterieren und dabei Duplikate löschen. Dazu merken wir uns stets das Vorgänger-Element und löschen das aktuelle Element, wenn beide übereinstimmen. Insgesamt benötigen wir somit Zeit in $\Theta(n \log(n))$.

2. Da wir die Menge der Decknamen in disjunkte Teilmengen zerlegen wollen, eignet sich die UnionFind-Datenstruktur für unsere Zwecke. Sie erlaubt es, die Teilmenge der zugehörigen Decknamen für jeden Spion zu verwalten.

3. Wir erstellen zunächst eine duplikatfreie Liste `names` aller Decknamen in L wie in Teilaufgabe 1 beschrieben. Nun legen wir eine Hashmap `info` und eine UnionFind-Datenstruktur `badgers` an. Die Hashmap verwendet dabei Tupel der Form (Zeit, Koordinate) als Schlüssel. `badgers` wird auf `names` initialisiert.
   Nun iterieren wir über `sightings`. Für jedes Tupel (t, p, d) überprüfen wir, ob bereits ein Eintrag in `info` für den Schlüssel (t, p) existiert. Wenn nicht, dann fügen wir d als Eintrag für den Schlüssel (t, p) ein. Ansonsten existiert bereits ein Eintrag d'. Dann rufen wir `union` für die Elemente d und d' in `badgers` auf.
   Nun korrespondiert jeder Baum in `badgers` zu einem Dachs. Wir können nun für jeden Repräsentanten in `badgers` eine Liste anlegen und dort diejenigen Namen aus `names` einfügen, die im gleichen Baum liegen. Damit haben wir für jeden Dachs genau eine Liste mit den vom ihm verwendeten Namen.

4. Wie bereits in Teilaufgabe 1 beschrieben, benötigt das Erstellen von `names` Zeit in $\Theta(n \log(n))$. Das anschließende Iterieren über `sightings` benötigt Zeit in erwartet $\Theta(n \cdot \log^*(n))$, denn: Wir müssen jedes Element in `info` suchen. Dies geschieht in erwartet konstanter Zeit. Zudem rufen wir bis zu $n-1$ Mal `union` mit Zeitbedarf $\Theta(\log^*(n))$ auf. Für das erstellen der Listen, die unsere Ausgabe bilden, müssen wir für jedes Element in `names` `find` aufrufen. Dies benötigt Zeit in $\Theta(n \log^*(n))$. Insgesamt hat unser Algorithmus einen Zeitbedarf in erwartet $\Theta(n \log(n))$.

5. Wir können unseren Algorithmus aus Teilaufgabe 3 unverändert verwenden. Anschließend zählen wir die Anzahl der Listen, die mehr als ein Element enthalten. Die benötigt zusätzlich Zeit in $\Theta(n)$.