Algorithmen und Datenstrukturen 1

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

Wörterbücher, Hashing (Woche 11)

Eigenständige Vorbereitung:
Lies 📖 CLRS Kapitel 11 ohne 11.5 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 11.1 (Von Hand laufen lassen und Eigenschaften 👍).

a) (einfach) Füge die Schlüsselsequenz 7, 18, 2, 3, 14, 25, 1, 11, 12, 1332 in eine Hashtabelle der Länge 11 ein, die verkettetes Hashing und die Hashfunktion $f(k) = k \mod 11$ benutzt.

b) (einfach) Füge die Schlüsselsequenz 2, 32, 43, 16, 77, 51, 1, 17, 42, 111 in eine Hashtabelle der Länge 17 ein, die lineares Sondieren und die Hashfunktion $f(k) = k \mod 17$ benutzt.

c) Lösche 111 und 51 aus der in b) erzeugten Hashtabelle.

d) Angenommen, wir löschen ein Element x bei linearem Sondieren, ohne die Elemente im Cluster rechts von x wieder neu einzufügen. Gib eine kürzestmögliche Sequenz von Wörterbuchoperationen an, sodass diese modifizierte Variante zu einem falschen Ergebnis führt.

e) Sei S eine Sequenz von Schlüsseln, die in einer Hashtabelle A mittels verkettetem Hashing gespeichert sind. Gegeben A, ist es möglich den größten Schlüssel aus S effizient zu finden?

Aufgabe 11.2 (Teiler in der Divisionsmethode 🔑). Bei der *Divisionsmethode* wählen wir die Hashfunktion als $h(k) = k \mod m$.

a) Betrachte die Hashfunktion $h(k) = k \mod 10$ und die Schlüsselsequenz

$$K = 0, 5, 20, 40, 65, 15, 90, 95, 80, 55\,.$$

Warum ist diese Wahl der Hashfunktion problematisch für K? Inwiefern wäre die Hashfunktion $h(k) = k \mod 11$ besser für K?

b) Konstruiere eine Schlüsselsequenz K_m, die schlecht für $h(k) = k \mod m$ ist.

Aufgabe 11.3 (Faules Löschen bei linearem Sondieren 🔑). Die Methode aus 11.1d) hat nicht funktioniert, wir versuchen es also nochmal anders. Wenn wir ein Element an Position p löschen, dann hinterlassen wir jetzt eine Markierung, dass dort ein Element gelöscht worden ist.

a) Wie können `Search` und `Insert` modifiziert werden, damit diese Methode funktioniert?

b) Welche Vor- und Nachteile hat diese Methode im Vergleich zu der Methode aus der Vorlesung hat?

Aufgabe 11.4 (Spielserverstatistiken 🔑). Für dein neues, extrem erfolgreiches Onlinespiel *Hashnite* willst du ermitteln, ob die vielen gespielten Spielsitzungen von einer kleinen Gruppe extrem aktiver Spieler:innen kommt oder aus einer großen Gruppe verschiedener Spieler:innen, die unregelmäßig spielen. Jede Spieler:in hat eine eindeutige ID, und von deinem Spieleserver aus kannst du auf die Liste aller IDs aus allen vergangenen Spielesitzungen zugreifen.

a) Entwirf einen Algorithmus, der die Anzahl an *unterschiedlichen* Spieler:innen ermittelt, die jemals auf dem Server gespielt haben.
b) Entwirf einen Algorithmus, der die Spieler:in ermittelt, die die meisten Spielsitzungen gespielt hat.

Aufgabe 11.5 (Bitvektoren ✏️). Eine naive Implementierung würde einen *Bitvektor* $B \in \{0,1\}^n$ als Array von `int`s darstellen, in dem jeder Eintrag 0 oder 1 ist:

```
int[] B = new int[n]; // in C++
```

Diese Darstellung verschwendet eine ganze Menge Platz, da für jedes Bit ein ganzer Integer genutzt wird, der 32 oder 64 Bit lang ist.[1] Wir wollen nun *Bit-Operationen* nutzen, um Platz zu sparen. Angenommen wir arbeiten auf einem *w-Bit Computer*, das heißt, die Register und Speicherzellen speichern jeweils w Bits und primitive Datentypen wie Integer, Gleitkommazahlen und Zeiger werden mit w Bits dargestellt. Viele Programmiersprachen unterstützen Bit-Manipulationen mit konstantem Zeitaufwand, wie *bit shifts* (`<<` und `>>`) und bitweise logische Operationen (`|&^`). Löse die folgenden Teilaufgaben.

a) 👍 Für $w = 8$, schreibe 2^4, `1<<4`, $2^8 - 1$, $(2^7 \text{ \^{} } 2^4)$ und $(2^8 - 1)$ `&` 2^4 in Binärdarstellung.
b) 🔑 Wir betrachten zunächst den Spezialfall $n = w$. Zeige, wie ein Bitvektor B der Länge w **kompakt** dargestellt werden kann, sodass das i-te Bit in konstanter Zeit ausgelesen oder geflippt werden kann. (Die schlimmste Laufzeit darf also nicht von w abhängen.)
c) 🔑 Wir betrachten nun den allgemeinen Fall $n \geq w$. Wie kann ein Bitvektor B der Länge n **kompakt** dargestellt werden, sodass das i-te Bit in konstanter Zeit ausgelesen oder geflippt werden kann? (Die schlimmste Laufzeit darf also weder von n noch von w abhängen.)
d) 🔑 Entwirf eine **kompakte** Datenstruktur, die eine dynamische Menge $S \subseteq \{0, \ldots, t\}$ darstellt und dabei die folgenden Operationen in konstanter Zeit unterstützt:
   - `insert`(a) fügt der Menge a hinzu, setzt also $S \leftarrow S \cup \{a\}$.
   - `remove`(a) löscht a aus der Menge, setzt also $S \leftarrow S \setminus \{a\}$.
   - `has`(a) liefert 1 wenn $a \in S$ und 0 sonst.

Aufgabe 11.6 (Sortieren in kleinen Universen 👍). Sei $A[1 \ldots n]$ ein Feld von Zahlen aus $\{1, \ldots, n\}$ sind. Entwirf einen Algorithmus, der A in Zeit $O(n)$ sortiert.

Aufgabe 11.7 (Nicht initialisierte Felder 🔭, sehr schwer). Wir wollen eine abstrakte Datenstruktur implementieren, die sich so verhält wie ein Feld A von ganzen Zahlen, das heißt, es werden die folgenden Operationen unterstützt:

a) `init`$(n,$ `default`$)$ initialisiert das Feld und legt den Integer `default` als Standardwert fest.
b) `set`(i, v) setzt den i-ten Eintrag auf den Integer v.
c) `get`(i) liefert den i-ten Eintrag oder `default`, falls dieser noch nicht gesetzt wurde.

Allerdings wird das Feld *riesig* sein, deshalb wollen wir die Datenstruktur *nicht* direkt als Feld implementieren, da wir keine Zeit darauf verschwenden wollen, alle Einträge auf `default` zu initialisieren.

Entwirf eine Datenstruktur, die nur $O(n)$ Platz benötigt, Auslesen und Ändern in erwarteter konstanter Zeit pro Eintrag unterstützt und nur konstante Zeit für die Initialisierung benötigt. *Hinweis: Hashtabellen und die Lösung zu Aufgabe 4.9 b) „Dynamische Felder" könnten hilfreich sein.*

[1] `bool` spart leider auch keinen Platz: https://stackoverflow.com/questions/2064550/c-why-bool-is-8-bits-long