

Max Wisniewski , Alexander Steen

Tutor: Lena Schlipf

Aufgabe 1 Das Offline-Minimum-Problem

Wir verwalten eine Menge $T \subseteq \{1, \dots, n\}$, welche durch die Operationen *insert* und *extract-min* verändert wird. Inizial gilt $T = \emptyset$. Wir bekommen dazu eine Folge S von insert und extract Operationen, in der wir jedes Element in $\{1, \dots, n\}$ *genau einmal* einfügen.

a) Betrachten Sie die folgende Operationsfolge

$$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5 ,$$

wobei eine Zahl i bedeutet, dass *insert*(i) ausgeführt wird. Geben Sie die Ergebnisse der einzelnen *extract-min* Operationen an.

Lösung:

- 1 . extract-min: 4
- 2 . extract-min: 3
- 3 . extract-min: 2
- 4 . extract-min: 6
- 5 . extract-min: 8
- 6 . extract-min: 1

b) Gegeben ist eine Algorithmus und eine Folge von Operationen

$$I_1, E, I_2, E, I_3, E, \dots, I_m, E, I_{m-1}$$

wobei jedes I_j eine Folge von insert Operationen darstellt und E wieder eine *extract-min* Operation. $K(j)$ ist die Menge, die die Zahlen aus I_j enthält.

Ausgeführt:

Wir haben 7 Insertfolgen, dass heißt $m + 1 = 7$.

$$I_1 = 4, 8 \quad I_2 = 3 \quad I_3 = 9, 2, 6 \quad I_4 = I_5 = [] \quad I_6 = 1, 7 \quad I_7 = 5$$

$$i=1 \rightarrow j = 6 : \text{em}[6] = 1, l=7 \rightarrow K(7) = \{1, 7, 5\}$$

$$i=2 \rightarrow j = 3 : \text{em}[3] = 2, l=4 \rightarrow K(4) = \{9, 2, 6\}$$

$$i=3 \rightarrow j = 2 : \text{em}[2] = 3, l=4 \rightarrow K(4) = \{9, 2, 6, 3\}$$

$$i=4 \rightarrow j = 1 : \text{em}[1] = 4, l=4 \rightarrow K(4) = \{9, 2, 6, 3, 4, 8\}$$

$$i=5 \rightarrow j = 7 : \text{nichts, da } 7 = m + 1$$

$$i=6 \rightarrow j = 4 : \text{em}[4] = 6, l=5 \rightarrow K(5) = \{9, 2, 6, 3, 4, 8\}$$

$$i=7 \rightarrow j = 7 : \text{nichts, da } 7 = m + 1$$

$$i=8 \rightarrow j = 5 : \text{em}[5] = 8, l=7 \rightarrow K(7) = \{1, 2, 3, 4, 5, 7, 9\}$$

$i=9 \rightarrow j=7$

Für das i -te Extract-min steht sein Ergebnis nun in $em[i]$. Wir erhalten die selbe Folge von Ergebnissen, wie wir sie in a) berechnet haben.

Korrektheit:

Beh.: $\forall i \leq n : (i = E_k \Rightarrow em[k] = i) \vee i$ wird nie genommen,
wobei E_k für das Ergebnis der k -ten extract-min Operation steht.

I.A.: Wir haben noch keine Vereinigungen gemacht. D.h. jedes $K(j)$ existiert noch und ist die Menge der Zahlen, die in I_j eingefügt wurden. Wenn wir nun ein E_k haben, dass die 1 sieht, (d.h. $k < m+1$), dann wird dieses E_k es nehmen. Da 1 ein globales Minimum ist, wird ist das auch ein lokales Minimum, wenn danach noch ein extract ausgeführt wird.

Sollte kein extract mehr ausgeführt werden, so gilt $1 \in K(l), l = m+1$. Der Algorithmus trägt es nicht ein.

Damit gilt unsere Behauptung für $i = 1$.

I.S.: $i-1 \rightarrow i$

Nach Induktionsvoraussetzung wurden die Zahlen $1, \dots, i-1$ schon ihren Extract Operationen zugewiesen. Sie $R = \{E_k | E_k = \{1, \dots, i-1\}\}$. Nun wissen wir nach dem Algorithmus, dass wir die Zahl i in einem Extract erst sehen können nach einem $i \in I_{in}$. Sei k der Index, so dass $E_k = i$, wobei $k = m+1$, wenn es kein Intervall gibt. Und $E_k \notin R$

Nun müssen zwischen in und k alle Extracts schon einen Wert haben, der kleiner ist als i , sonst würde es ein $x < k$ geben, so dass dieses Extract das i genommen hätte.

Nach Induktionsvoraussetzung wurden alle Werte von x $in \leq x < k$ schon einen Wert besitzen. Der Algorithmus wird eine Menge, immer mit der nächst größeren Menge vereinigen. Da nun aber alle $K(x)$ mit genannten eigenschaften mit einer nächst größeren Menge vereinigt worden sind, muss der Algorithmus wenn er i sucht, die Menge nehmen, in der sich nun $K(in)$ befindet. Diese befindet sich, wie gezeigt in $K(k)$.

Unser Algorithmus hat also für $i = E_k$ das Feld $em[k] = i$ gesetzt. Sollte $k = m+1$ sein wurde kein Extract auf diesem Element ausgeführt und der Algorithmus hätte die if Block übersprungen und auch den Wert niemals zugewiesen.

□

- c) Beschreiben Sie, wie man den Algorithmus aus b) effizient mit einer Union-Find Struktur implementieren kann und analysieren Sie die Laufzeit.

Lösung:

Für die effiziente Berechnung brauchen wir zunächst eine Unionfind Struktur **UF**. Diese wird initial so vereinigt, dass wir alle Elemente, die in der selben Insertfolge liegen vereinigt werden. Prinzipiell können wir auch dies als Startpartition aussuchen.

Im nächsten Schritt packen wir alle Representanten als Schlüssel in eine SortedMap **SM** und eine HashMap **HM** um die Umkehrung zu finden, die eine Menge mit einem

Extract identifizieren kann.

Dafür rufen wir auf einem beliebigen $x \in I_j$, $find(x)$ auf und packen es mit dem j in die Map:

Dazu bauen wir das Programm, dass danach folgendes erfüllt ist:

$forall 1 \leq j \leq m + 1 : x \in I_j \Rightarrow (x, j) \in SM \wedge (j, x) \in HM$

Der Algorithmus sieht nach dieser Vorverarbeitung folgender Maßen aus:

```

for i := 1 to n do
  j' := UF.find(i);
  j := SM.get(j');
  if j != m + 1 do
    em[j] := i;
    l := SM.ceilKey(j);
    l' := HM.get(l);
    UF.union(j', l');
    nJ := UF.find(j');
    HM.del(i');
    HM.del(j');
    HM.add(nJ, l);
    SM.del(j);
    SM.replace(l, nJ);

```

Laufzeit: Die Vorverarbeitung kostet uns $O(n + m \log m)$. Wir müssen n Elemente in ihre Mengen in der UnionFind Struktur stecken. Da wir linear durch die Liste laufen können, ist es unmöglich ein neues Element (Baum mit Rank 0) mit dem Repräsentanten zu vereinigen. Daher ist der Rank jeder Komponente nach der Initialisierung 1. Die Laufzeit zum reinen joinen ist. Wenn wir ein Intervall haben, so müssen wir die Repräsentanten danach in eine SortedMap eintragen. Dies können wir beispielsweise über einen RotSchwarzBaum machen. Einfügen hat hier drauf die Laufzeit $O(\#Elemente)$. Bis zum Schluss also bei m Inserts M Elemente.

Im gegebenen veränderten Algorithmus haben wir eine Schleife, die n mal durchlaufen wird. Zu Beginn jedes Durchlaufens brauchen wir $find$ auf einem beliebigen Element auszuführen. Dies kann im schlimmsten Fall der Schlechtkonditioniertest Fall sein, so dass wir $O(\alpha(n, n))$ benötigen. Die nächste Operation kostet uns $\log m$ Zeit.

Sind wir nicht in der letzten Komponente so müssen wir uns mit $ceilKey$ ($O(\log m)$) und get ($O(\log m)$) die Mengen und Intervalle zu finden. Die del haben die selbe Laufzeit müssen also konstant oft ausgeführt werden.

Das Union kostet $O(1)$ und $find$ wurde auf den Repräsentanten der gerade vereinigten Mengen ausgeführt, diese stehen auf maximal 1 unter der neuen Wurzel. Die Operation kostet daher $O(1)$.

Insgesamt ergibt sich eine Laufzeit von $O(m \log m + n \log m)$.

Aufgabe 2 Amortisierte Analyse

- a) Gegeben sei ein elektisches Binärzählwerk mit beliebig vielen Ziffern aus der Menge $\{0, 1\}$. Das Umschalten einer Ziffer kostet eine Stromheinheit. Wie viele Stromeinheiten kostet es insgesamt, wenn man das Zählwerk von 0 bis n aufsteigend zählen

lässt? Was sind die amortisierten Kosten pro Zählvorgang?

Lösung:

Gesamtkosten: Wenn einen binären Zähler hochzählen, dann müssen wir eine Ziffer an der Stelle k genau dann ändern, wenn wir die Ziffer an der Stelle $k - 1$ von 1 auf 0 ändern. Da wir jede Ziffer einmal auf 0 und einmal auf 1 setzen können, wird die Ziffer an der Stelle k halb so oft umgesetzt werden, wie die Ziffer an der Stelle $k - 1$ werden. Die erste Ziffer an der Stelle 0, wird bei jedem erhöhen verändert. Dies ergibt für uns, dass wir die 0te Stelle n mal ändern müssen, die erste Stelle $\frac{n}{2}$, die zweite Stelle $\frac{n}{4}$ und an der k ten Stelle wird das Bit $\frac{n}{2^k}$ geändert.

In der Summe werden wir also:

$$\sum_{i=0}^{\log n} \frac{n}{2^i} = n \cdot \sum_{i=0}^{\log n} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i = n \cdot 2$$

Bits ändern müssen. Dies hat zur Folge, da jedes ändern 1ne Stromeinheit kostet, das wir beim Hochzählen höchstens $2n$ Stromeinheiten brauchen werden.

Amortisierte Kosten: Wir haben eine Folge der Länge n und die Kosten dieser Folge betragen $2n$. Das ergibt amortisierte Laufzeit von $T_{amo} = \frac{2n}{n} = 2 = O(1)$.

Nach der Analyse könnten wir nun auch aussagen, dass wenn wir für jedes Zählen 2 Stromeinheiten zahlen, immer über 0 bleiben würden.

- b) Entwicklen und analysieren Sie eine Methode um ein Array zu verwalten, welche seine Größe dynamisch ändern kann.

Lösung:

voll -> verdoppeln

1/3 -> halbieren

Aufgabe 3 Wahrscheinlichkeitsrechnung und Hashing

- a) Auf der ausgelassenen Weihachtsfeier der Teilnehmer der HA-Vorlesung gibt es einen Julklapp. Die Teilnehmer bringen jeweils ein Geschenk mit und alle Geschenke werden in einen großen Sack getan. Danach zieht jeder der Partygäste zufällig ein Geschenk aus dem Sack. Was ist die erwartete Anzahl der Leute, die ihr eigenes Geschenk ziehen?

Lösung:

Wir konstruieren uns eine Indikatorvariable, ob die k -te Person ihr eigenes Geschenk zieht. Die Wahrscheinlichkeit bestimmen wir unabhängig von den $k - 1$ vorrigen Personen, so dass wir über alle Indikatorvariablen eine Summer bilden können und diese immer noch mit der Linearität des Erwartungswertes nach aussen ziehen können.

Sei

$$X_k = \begin{cases} 1 & , k\text{-te Person zieht ihr geschenk} \\ 0 & , \text{sonst} \end{cases}$$

Die Wahrscheinlichkeit $Pr(X = X_k)$ muss nun mit einberechnen, dass die $k - 1$ Gäste zuvor das Geschenk nicht gezogen haben. Da bei jedem Zug die Anzahl der Geschenke sich verringert haben wir die Wahrscheinlichkeit:

$$Pr(X_k = 1) = \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdot \dots \cdot \frac{n-k}{n-k+1} \cdot \frac{1}{n-k} = \frac{(n-1)! \cdot (n-k)!}{(n-k-1)!n!(n-k)} = \frac{1}{n}$$

Wir sehen, dass die Wahrscheinlichkeit nicht mehr von k sondern nur noch von n abhängt. Der Erwartungswert ist somit:

$$E \left[\sum_{i=1}^n X_i \right] = \sum_{i=1}^n E[X_i] \stackrel{\text{Indikator}}{=} \sum_{i=1}^n Pr(X_i = 1) = \sum_{i=1}^n \frac{1}{n} = \frac{n}{n} = 1$$

Wir erwarten also, dass 1ne Person ihr eigenes Geschenk zieht.

- b) Sei N die Größe einer gegebenen Hashtabelle und n beliebig. Zeigen Sie, dass für jede Schlüsselmenge K mit $|K| \geq (n-1)N + 1$ und jede Hashfunktion $h : K \rightarrow \{0, \dots, N-1\}$ eine Menge $S \subseteq K$ mit $|S| \geq n$ existiert, so dass alle Elemente von S auf denselben Eintrag der Hashtabelle abgebildet werden.
Was bedeutet das für die worst-case Laufzeit von Hashing mit Verkettung.

Lösung:

Diese Aufgabe lösen wir dem Taubenschlagprinzip. Bei diesem gilt:

Sei $f : A \rightarrow B$, dann

$$\exists y \in B : \#\{x \in A | f(x) = y\} = \left\lceil \frac{\#A}{\#f(A)} \right\rceil.$$

Nun haben wir $f = h$, $A = K$ $f(A) \subseteq \{0, \dots, N\}$ mit $\#A = \#K \geq (n-1)N + 1$ und $\#f(A) \leq \#\{0, \dots, N-1\} = N$.

Nach dem Taubenschlagprinzip existiert nun $t \in \{0, \dots, N-1\}$, so dass für $S_t = \{x \in K \mid h(x) = t\} \subseteq K$ gilt:

$$\#S_t \geq \left\lceil \frac{\#K}{\#\{0, \dots, N-1\}} \right\rceil = \left\lceil \frac{(n-1)N + 1}{N} \right\rceil = \left\lceil n - 1 + \frac{1}{N} \right\rceil = n.$$

Diese $S_t \subseteq K$ können wir nun als unser S wählen, so dass wir die Voraussetzung der Aussage erfüllen.

Für die Laufzeit bedeutet das, wenn wir gerade einen Wert $x \in K$ suchen, so dass $h(x) = t$ gilt, dann müssen wir im schlimmsten Fall n Schritt machen, da wir die Position im Array in $O(1)$ finden, aber danach eine Liste der Länge n durchsuchen müssen.

Dies gilt auch für Löschen, da wir ebenfalls das Element suchen müssen.

Einfügen geht weiterhin in $O(1)$, da man in $O(1)$ in verkettete Listen einfügen kann.