Max Wisniewski , Alexander Steen

Tutor: Lena Schlipf

Aufgabe 1 Das Offline-Minimum-Problem

Wir verwalten eine Menge $T \subseteq \{1, ..., n\}$, 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 $\{1, ..., n\}$ *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

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

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

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

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

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

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

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

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

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

i=8 $\rightarrow$ $j = 5$: 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 Induktionsvorraussetzung wurden die Zahlen $1, .., i - 1$ schon ihren Extract Operationen zugewiesen. Sie $R = \{E_k | E_k = \{1, ..., i - 1\}\}$. 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 ken 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 Induktionsvorraussetzung wurden alle Werte von x $in \leq x < k$ schon eine Wert besitzen. Der Algorithmus wird eine Menge, immer mit der nächst groößeren 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 Represantanten 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 beliebeigen $x \in I_j$, $find(x)$ auf und packen es mit dem j in die Map:
Dazu bauen wir das Programm, dass danach folgendes erfüllt ist:
$foral 1 \leq j \leq m+1 \ : \ x \in I_j \Rightarrow (x,j) \in SM \ \wedge \ (j,x) \in HM$

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 un möglich ein neues Element (Baum mit Rank 0) mit dem Represtanten 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 Representaten 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 Begin jedes Durchlaufens brauchen müssen wir find auf einem beliebigen Element ausführen. Dies kann im schlimmsten Fall der Schlechtkondizioniertest 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 Representaten der gerade vereinigten Mengen ausgeführt, diese stehen auf maximal 1 unter der neuen Wurzel. Die Operation kostet daher $O(1)$.

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

Aufgabe 2 Amortisierte Analyse

a) Gegeben sei ein elektirsches 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, wir 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 kten 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:

Wir müssen für die dynamische Größe 2 Dinge tun. Wen das Array voll oder über eine bestimmte Füllmenge geht, dann müssen wir das Array vergrößern und alles was bisher darin war kopieren. Wenn es zu leer wird, müssen wir es wieder verkleinern und auch wieder alles kopieren.

Damit wir einfügen und löschen konstant hinbekommen, kann das kopieren nur um einen konstanten Faktor der Größe sein. So können wir beim einfügen und löschen die Kosten darauf aufteilen. Wir wollen die Kosten nun so aufteilen, so dass wir mit dem Einfügen das Vergrößern bezahlen und mit dem Löschen das Verkleinern.

Wir müssen also nur 2 Arten von Folgen analysieren. Die erste ist eine Folge von add und enlarge Operatioenen (die wir durch reine adds darstellen können) und die andere eine Folge von remove und reduce Operatioenen (die wir wieder durch reine removes darstellen können).

Betrachten wir uns diese Folgen einmal genauer, sehen wir, dass wir einen Worst-Case so aufbauen können, das wir die amortisierte Laufzeit gegen diesen Fall abschätzen können. Dieser Fall ist genau ein flackern zwischen vergrößern und verkleinern. Hätten wir noch mehr Operationen würden wir zwar eine leicht bessere Gesamtlaufzeit bekommen, aber wenn wir die Laufzeit des Flackerns gut beschreiben können, haben wir auch die besseren Laufzeiten damit abgeschätzt.

Als nächstes Überlegen wir uns, dass es keinen Sinn macht das Array zu vergrößern, bevor es vollständig gefüllt ist, da wir sonst immer unnötig Speicher verbrauchen, da der letzte Teil nie genutzt wird.

Damit das Array gleichmäßgt wächst sagen wir nun, dass wir es, wenn es voll ist, k mal so groß machen wie zuvor. Ist es zu leer so teilen wir die Länge durch k. Damit ist die Länge des Array ein Exponent zur Basis k. Eine leichte erste Überlegung ist, dass wir als Grenze für das verkleinern einen Wert wählen müssen, der kleiner als $\frac{1}{k}$ ist, da wir sonst das Array nicht komplett rein kopieren könnten.

Da wir durch einfügen das Array quasi unendlich oft erweitern können, können wir aussagen, dass die Kosten vom erweitern von allen add Operationen getragen werden müssen, seit dem letzten Kopiervorgang. Sonst müsste ein add, die Kosten von m kopiertvorgängen mittragen und wär somit nicht mehr konstant.

Sei $l > k$ nun die Zahl, so dass wir das Array bei einer Füllmenge von $\frac{1}{l}$ verkleinern. D.h. die Operatioen im Worst-Case zum einfügen bei einem Array der Größe n sind genau $n - \frac{k \cdot n}{l} = \frac{l-k}{l} n$. Nun ist die Folge so aufgebaut, dass wir nach diesen Operationen noch n Elemente kopieren müssen. Sollten wir nun noch dazwischen Remove ausführen oder eine Längere Liste von adds zur Verfügung haben, würde die relativen Kosten nur sinken.
Wann wir für eine add Operation also $\frac{l}{l-k}$ annehmen (WIr brauchen n Operationen für das umkopieren und die Folge ist so Lang wie eben geant. Also brauchen wir das reziproke des Koeffizienten), so können wir das Array auf jedenfall Erweitern und haben dabei amoritisiert nur konstante Kosten gehabt.

Gleiche Überlegungen führen dazu, dass man beim Löschen in einem Array der Länge n $\frac{n}{k} - \frac{n}{l} = \frac{l-k}{l \cdot k} n$ Operationen hat. Für eine Operation können wir amortisiert die Kosten $\frac{l \cdot k}{l-k}$ annehmen, so sind die Kosten für das Verkleinern amortisiert wieder konstant.

Nun haben wir gezeigt, dass wir auch mit erweitern und verkleinern eine amorisiert Konstante Laufzeit hinbekommt und zwar mit $T_{add}(n) = \frac{l}{l-k}$ und $T_{rem}(n) = \frac{l \cdot k}{l-k}$. Nun müssen wir nur noch k, l so bestimmen, dass wir einen möglichst guten Tradeoff zwischen Laufzeit und der Speicherplatznutuzng finden. Der bei dieser aufgestellten Eigenschaft mindestens benötigte Platz ist $\frac{k}{l} n$ (nach dem verkleinern) oder $\frac{1}{k} n$ (nach dem Vergrößern)

Nun haben wir 4 Funktionen, von denen wir die Laufzeit möglichst klein und den Platz möglichst groß halten wollen. Da wir immer mindestens ene Formel finden, bei der min eine der Variablen im Nenner steht, können wir darauf schließen, dass der Ideale Punkt in einem recht niedrigen Bereich liegt, da der Graph eine Hyperbel beschreibt und dieser gegen 0 sehr schnell steigt, was uns für die Laufzeit nicht gefällt

und bei großen Werten recht schnell gegen die 0 geht.

Wir schauen uns also einfach einmal eine Tabelle über ein paar exemplarische Werte an. Wir haben uns dafür entschieden nur natürliche Ergebnisse zu nehmen, da sich es gerade für k günstiger macht, wenn die Länge des Arrays eine natürliche Zahl ist.

(k,l)	T_{add}	T_{rem}	Sp_{gr}	Sp_{kl}
(2,3)	3	6	$\frac{1}{2}n$	$\frac{2}{3}n$
(2,4)	2	4	$\frac{1}{2}n$	$\frac{1}{2}n$
(3,4)	4	12	$\frac{1}{3}n$	$\frac{3}{4}n$
(2,5)	$\frac{5}{3}$	$\frac{10}{3}$	$\frac{1}{2}n$	$\frac{2}{5}n$
(3,5)	$\frac{5}{2}$	$\frac{15}{2}$	$\frac{1}{3}n$	$\frac{3}{5}n$
(4,5)	5	20	$\frac{1}{4}n$	$\frac{4}{5}n$

Nun halten wir den Tradeoff bei den ersten beiden Einträgen der Tabelle für am akzeptabelsten. Weder die Laufzeiten sind zu schlecht, noch der Platz wird zu wenig genutzt. Nun kann man sich noch dafür entscheiden, ob es einem wichtiger ist, weniger Platz zu verbrauchen und dafür geringfügig länger bei den Operationen zu brauchen oder lieber den Platz optimaler Nutzen.

Wenn wir viel aus Datenstruckturen löschen möchten, macht sich der (2,3) Fall ganz gut, wenn wir eher viel einfügen der (2,4) Fall, da bei beiden ver erweiterungsspeicher gleich ist, aber die Laufzeit besser.
Java implementiert z.B. eher den ersten Fall.

Wir konnten also ermitteln, dass wir ein Array dynamisch verwalten können und das mit konstanten Kosten. Dabei ergab sich für der beste Tradeoff zwischen Laufzeit und Speicherausnutzung, wenn wir die Arrays immer verdoppeln oder halbieren. Halbieren können wir dabei entweder bei $\frac{1}{4}$ Füllmenge, wen wir eine schnellere laufzeit der Operationen wollen, oder bei $\frac{1}{3}$, wenn wir den Platz besser nutzen wollen.

Aufgabe 3 Wahrscheinlichkeitsrechnung und Hashing

a) Auf der ausgelassenen Weihanchtsfeier der Teilnehmer der HA-Vorlesung gibt es einen Julklapp. Die Teilnehmer bringen jeweils ein Geschenk mit und alle Geschenke werdenin 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

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

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

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

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

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

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 $h : K \to \{0, ..., N-1\}$ 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 $f : A \to B$, dann

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

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

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

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

Diese $S_t \subseteq K$ können wir nun als unser S wählen, so dass wir die Vorraussetzung 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 verkette Listen einfügen kann.