Quake Heaps — Erdbebenhaufen

Wolfgang Mulzer

1 Prioritätswarteschlangen

Sei X eine Menge von *Elementen* und K eine total geordnete Menge von *Schlüsseln*. Eine *Prioritätswarteschlange* speichert eine Menge $S \subseteq X \times K$ und unterstützt die folgenden Operationen:

- `insert`(x, k): Füge das Element x mit Schlüssel k zu S hinzu.
- `decrease-key`(x, k): Verringere den Schlüssel für x zu k. Dabei muss x in S vorhanden sein, und der alte Schlüssel für x muss größer als k sein.
- `delete-min`(): Finde das Element in S mit dem kleinsten Schlüssel, entferne es und gib es zurück.

Timothy Chans Quake Heaps implementieren den ADT Prioritätswarteschlange. Sie erreichen konstante amortisierte Laufzeit für `insert` und `decrease-key` sowie $O(\log n)$ amortisierte Laufzeit für `delete-min`.

2 Turnierbäume

2.1 Definition

Quake Heaps speichern die Elemente in *Turnierbäumen*. Ein Turnierbaum ist ein gewurzelter Baum mit den folgenden Eigenschaften (siehe auch Abbildung 1).

1. Alle Elemente und Schlüssel sind in den Blättern gespeichert.
2. Alle Blätter haben die gleiche Tiefe, d.h., alle Pfade von der Wurzel zu Blättern sind gleich lang.
3. Alle inneren Knoten haben 1 oder 2 Kinder.
4. Jeder innere Knoten verweist auf einen Schlüssel, das Minimum der Schlüssel seiner Kinder.
5. Jedes Blatt v enthält einen Verweis auf den höchsten Knoten, der auf den Schlüssel in v verweist.

2.2 `Link` und `Cut`

Turnierbäume unterstützen zwei Operationen: `link` und `cut` (siehe Abbildung 2).

`link`. Die Operation `link` erhält zwei Turnierbäume T_1 und T_2 gleicher Höhe h und vereinigt sie zu einen Turnierbaum der Höhe $h+1$. Sie erzeugt eine neue Wurzel r, die auf das Minimum der Schlüssel in den Wurzeln von T_1 und T_2 verweist, und macht T_1 und T_2 zu Kindern von r. Dies ergibt einen gültigen Turnierbaum, und alle Zeiger können in $O(1)$ Zeit aktualisiert werden. Folglich läuft `link` in konstanter Zeit.

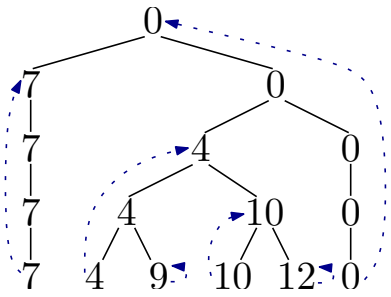

Fig. 1: Ein Turnierbaum. Alle Blätter haben die gleiche Tiefe. Jeder innere Knoten hat höchstens 2 Kinder und speichert das Minimum seiner Kinder.

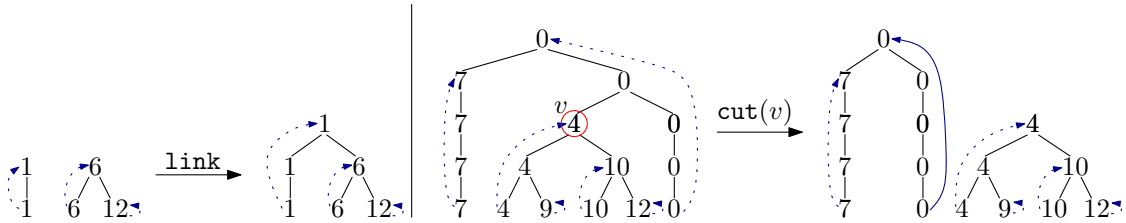

Fig. 2: Die Operationen `link` und `cut`. `link` vereinigt zwei Turnierbäume gleicher Höhe zu einem neuen Turnierbaum. `cut` schneidet einen Turnierbaumen in zwei Teile.

`cut`. Die Operation `cut` erhält einen Zeiger auf einen Knoten v, so dass sich der Schlüssel in v von dem Schlüssel in v's Elternknoten unterscheidet. Die Operation löscht die Kante zwischen v und seinem Elternknoten. Dadurch enstehen zwei gültige Turnierbäume, und die Laufzeit ist $O(1)$.

3 Operationen

Ein Quake Heap besteht aus einer Liste von Turnierbäumen. Die Operationen sind wie folgt implementiert.

3.1 `insert`

Die Operation `insert`(x, k) erzeugt einen neuen Turnierbaum, der nur aus einem Blatt mit Eintrag (x, k) besteht, und fügt ihn in die Liste ein. Dies benötigt $O(1)$ Zeit.

3.2 `decrease-key`

Die Operation `decrease-key`(x, k) erhält einen Verweis auf das Blatt, welches das Element x speichert. Dieser Verweis wird von `insert` zurückgeliefert, und er erspart uns die Suche nach x im Quake Heap.

Wir finden zuerst den höchsten Knoten v, der auf den Schlüssel für x verweist. Dies geht in konstanter Zeit mit Hilfe des Verweises im Blatt. Wenn v keine Wurzel ist, führen wir `cut`(v) aus. Dies ergibt einen Baum, in dem der Schlüssel für x minimal ist. Daher können wir den Schlüssel für x auf k verringern, ohne die Turnierbaum-Eigenschaften zu verletzen. Die Laufzeit ist konstant, da die inneren Knoten nur Verweise auf die Schlüssel speichern und wir daher den Schlüssel nur an einer Stelle ändern müssen.

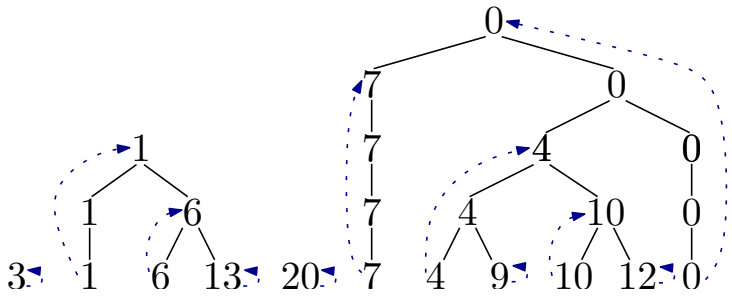

Fig. 3: Ein Quake Heap mit vier Turnierbäumen. Es gibt zwei Bäume der Höhe 0, einen Baum der Höhe 2 und einen Baum der Höhe 4. Ebene 0 hat 11 Knoten, Ebene 1 hat 6 Knoten, Ebene 2 hat 4 Knoten, Ebene 3 hat 2 Knoten, und Ebene 4 hat 1 Knoten. Die Invariante ist auf allen Ebenen erfüllt.

3.3 `delete-min`

Um `delete-min` effizient zu implementieren, benötigen wir einige zusätzliche Informationen in unserem Quake Heap. Neben den Turnierbäumen speichern wir zwei weitere Arrays.

- *Das Array* `T`. Der Eintrag `T[`i`]` speichert eine Liste von allen Bäumen mit Höhe i.
- *Das Array* `n`. Der Eintrag `n[`i`]` speichert die Anzahl der Knoten mit Höhe i.

Die Arrays `T` und `n` können mit konstantem Zusatzaufwand aktualisiert werden. Außerdem muss vor und nach jeder Operation die folgenden Invariante gelten:

Invariante 1. *Für alle* $i \geq 0$ *gilt* $\texttt{n}[i+1] \leq \frac{3}{4} \cdot \texttt{n}[i]$.

Die nächsten beiden Fakten folgen nun unmittelbar.

Fakt 2. *Invariante 1 impliziert, dass alle Knoten Höhe* $O(\log n)$ *haben.*

Fakt 3. *Die Operationen* `insert` *und* `decrease-key` *erhalten Invariante 1.*

Nun können wir `delete-min` beschreiben. Es besteht aus den folgenden vier Schritten:

1. **Minimum finden**. Gehe alle Wurzeln durch und finde das Element x mit minimalem Schlüssel.
2. **Minimum löschen**. Lösche den Pfad, der x entspricht. Dies erzeugt gegebenenfalls neue Bäume.
3. **Konsolidieren**. Stelle sicher, dass alle Bäume verschiedene Höhen haben. Dies geschieht wie folgt:
   (a) `for` i `:=` 0 `to` $O(\log n)$ `do`
   (b) `while T[`i`].size` ≥ 2 `do`
   (c) Lösche zwei Bäume T_1, T_2 aus `T[`i`]`.
   (d) $T^* \leftarrow \texttt{link}(T_1, T_2)$.
   (e) $\texttt{T}[i+1] \leftarrow \texttt{T}[i+1] \texttt{++} T^*$.
4. **Erdbeben**. Stelle sicher, dass Invariante 1 erfüllt ist. Gehe dazu alle Ebenen durch und finde die niedrigste Ebene i, für die $\texttt{n}[i+1] > \frac{3}{4} \cdot \texttt{n}[i]$ ist (d.h., Ebene i verletzt Invariante 1). Falls die Ebene i existiert, lösche alle Knoten mit Höhe größer als i. Ansonsten tue nichts.

Zur Analyse von `delete-min` benötigen wir zwei Lemmas.

Lemma 4. *Der Konsolidierungsschritt erzeugt höchstens $O(\log n)$ neue Ebenen.*

Beweis. Sei h die höchste nichtleere Ebene vor der Konsolidierung. Betrachte alle Knoten mit Höhe größer h, die beim Konsolidieren entstehen. Da `link` nur Knoten mit zwei Kindern erzeugt, ist jeder solche Knoten in einem perfekten binären Baum mit höchstens n Blättern enthalten (hierbei betrachten wir die Knoten mit Höhe h als die Blätter). Folglich gibt es höchstens $O(\log n)$ zusätzliche Ebenen. □

Lemma 5. *Sei T_0 die Anzahl der Bäume vor* `delete-min`*. Die Zeit für* `delete-min` *ist $O(\log n + T_0 + D)$, wobei D die Anzahl der gelöschten Knoten bezeichnet.*

Beweis. Wir brauchen $O(T_0)$ Zeit, um das Minimum zu finden. Wegen Fakt 2 benötigen wir $O(\log n)$ Schritte zum Löschen des Minimums. Der Konsolidierungsschritt benötigt $O(\log n)$ Zeit für die `for`-Schleife, plus die Zeit für die `link`-Operationen. Ein `link` braucht konstante Zeit, und bei jedem Link verringert sich die Anzahl der Bäume um eins. Da es vor dem Konsolidieren höchstens $T_0 + O(\log n)$ Bäume gibt (im zweiten Schritt entstehen ggf. $O(\log n)$ neue Bäume), ist die Zeit zum Konsolidieren $O(T_0 + \log n)$. Die Laufzeit für das Erdbeben ist $O(\log n)$, um die Ebene i zu suchen (wegen Lemma 4), plus $O(D)$, um die Knoten zu löschen. Insgesamt ergibt sich also eine Laufzeit von $O(\log n + T_0 + D)$. □

4 Analyse

Wir wollen nun die folgende Aussage beweisen:

Satz 6. *Jede Operationsfolge mit i* `insert`*s, d* `decrease-key`*s und e* `delete-min`*s benötigt $O(i+d+e\log n)$ Zeit. Dabei ist n die maximale Anzahl von Elementen, die im Quake Heap gespeichert sind. Das heißt,* `insert` *und* `decrease-key` *haben amortisierte Laufzeit $O(1)$,* `delete-min` *hat amortisierte Laufzeit $O(\log n)$.*

Die Idee ist, dass ein `delete-min` nur dann lange dauern kann, wenn viele `insert`s oder `decrease-key`s vorausgegangen sind. Wir können somit diese Zeit den vorherigen Operationen in Rechnung stellen. Es gibt verschiedene Methoden, um diese Umverteilung der Kosten zu formalisieren. Wir beschreiben die Analyse mit der *Buchhaltermethode*. In der Übung wird die *Potentialmethode* diskutiert.

Wir stellen uns vor, dass es für unseren Quake Heap die folgenden drei Konten gibt.

- Das *Knotenkonto* enthält 1 € für jeden Knoten im Quake Heap.
- Das *Baumkonto* enthält 2 € für jeden Baum im Quake Heap.
- Das *Einzelkinderkonto* enthält 4 € für jeden inneren Knoten, der nur ein Kind besitzt.

Die obigen Eigenschaften der Konten bezeichnen wir als *Konteninvariante*. Zu Beginn gibt es keine Knoten und alle Konten sind leer. Wir zeigen folgendes: Angenommen, wir zahlen $O(1)$ € für jedes `insert` und `decrease-key` und $O(\log n)$ € für jedes `delete-min`. Dann haben wir genügend Geld, um die tatsächlichen Kosten der Operationen zu bezahlen, ohne die Konteninvariante zu verletzen. Satz 6 folgt dann sofort.

`insert`. Die realen Kosten sind $O(1)$. Wir erzeugen einen neuen Knoten und einen neuen Baum, müssen also 1 € in das Knotenkonto und 2 € in das Baumkonto zahlen. Insgesamt sind $O(1)$ € zu bezahlen.

`decrease-key`. Die realen Kosten sind $O(1)$. Ggf. entstehen ein neuer Baum und ein neues Einzelkind, also sind bis zu 2 € auf das Baumkonto und 4 € auf das Einzelkinderkonto zu zahlen. Insgesamt also $O(1)$ €.

`delete-min`. Laut Lemma 5 sind die realen Kosten $O(\log n + T_0 + D)$, wobei T_0 die anfängliche Anzahl der Bäume ist und D die Anzahl der gelöschten Knoten. Wir beschreiben nun die notwendige Buchführung.

1. Hebe T_0 € vom Baumkonto ab, um die $O(T_0)$ realen Kosten zu bezahlen.
2. Zahle 1 € auf das Baumkonto für jeden neuen Baum beim Löschen des Minimums; $O(\log n)$ € total.
3. Überweise 1 € vom Baumkonto auf das Knotenkonto für jeden neuen Knoten, der durch ein `link` entsteht. Da jedes `link` die Anzahl der Bäume verringert, reicht das Geld aus.
4. Zahle 2 € auf das Baumkonto pro Baum nach dem Konsolidieren; $O(\log n)$ € total (wegen Lemma 4).
5. Hebe D € vom Knotenkonto ab, um die $O(D)$ realen Kosten zu bezahlen. Da D Knoten gelöscht werden, bleibt genug Geld auf dem Knotenkonto.
6. Bei einem Erdbeben, überweise bis zu $2\mathtt{n}[i]$ € vom Einzelkinderkonto auf das Baumkonto, um die evtl. neu entstandenen Bäume abzudecken. Hierbei ist i die Ebene, auf der sich das Erdbeben ereignet.

Die Buchhaltung deckt fast alle Kosten über die Konten. Es verbleiben $O(\log n)$ € zu entrichten. Wir müssen aber noch überprüfen, dass die Konteninvariante erhalten bleibt. Für das Baumkonto und das Knotenkonto ist dies leicht zu sehen. Für das Einzelkinderkonto brauchen wir das folgende Lemma. Es besagt, dass bei einem Erdbeben mindestens $\mathtt{n}[i]/2$ innere Knoten mit nur einem Kind gelöscht werden

Lemma 7. *Sei $\mathtt{n}[i+1] > \frac{3}{4}\mathtt{n}[i]$. Dann enthält Ebene $i+1$ mindestens $\mathtt{n}[i]/2$ Knoten mit nur einem Kind.*

Beweis. Sei n_1 die Anzahl der Knoten auf Ebene $i+1$ mit einem Kind, n_2 die Anzahl der Knoten auf Ebene $i+1$ mit zwei Kindern und n_0 die Anzahl der Wurzeln auf Ebene i. Es gilt $\mathtt{n}[i+1] = n_1 + n_2$ und $\mathtt{n}[i] = n_0 + n_1 + 2n_2$. Laut Annahme ist

$$n_1 + n_2 = \mathtt{n}[i+1] > \frac{3}{4}\mathtt{n}[i] = \frac{3}{4}n_0 + \frac{3}{4}n_1 + \frac{3}{2}n_2.$$

Daraus folgt $n_1 > 3n_0 + 2n_2 \geq n_0 + 2n_2$, und es gilt

$$n_1 = \frac{n_1 + n_1}{2} > \frac{n_1 + n_0 + 2n_2}{2} = \frac{\mathtt{n}[i]}{2},$$

was zu zeigen war. □

Aus Lemma 7 folgt, dass bei einem Erdbeben mindestens $4 \cdot \mathtt{n}[i]/2 = 2\mathtt{n}[i]$ € im Einzelkinderkonto frei werden, so dass die Überweisung im letzten Schritt die Konteninvariante erhält (`delete-min` erzeugt keine neuen Einzelkinder). Damit ist die Analyse beendet.