Kurzskript Amortisierte Analyse / Splay-Bäume, Melanie Schmidt

1 Amortisierte Analyse

Wir beschäftigen uns in diesem Skriptauszug mit *amortisierter Analyse*. Diese bietet eine Möglichkeit, die Laufzeit von Operationen auf einer Datenstruktur genauer abzuschätzen als dies mit einer reinen Worst-Case-Analyse möglich ist. Wir betrachten als Einstiegsbeispiel zunächst einen *Binärzähler*. Wir folgen für dieses Beispiel dem Lehrbuch [1] und besprechen Auszüge der Seiten 454–462.

1.1 Amortisierte Analyse durch Bestimmung der Gesamtlaufzeit

Unser Binärzähler soll aus k Bits bestehen. Als Datenstruktur verwenden wir ein Array A, welches von 0 bis $k-1$ adressiert wird, wobei $A[0]$ das niederwertigste und $A[k-1]$ das höchstwertigste Bit speichert. Die durch A dargestellte Zahl x ist also $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$. Zu Beginn ist A mit 0en initialisiert. Der Zähler soll nun 2^k die dargestellte Zahl um eins inkrementieren, also von x auf $x+1$ erhöhen. Der letzte Schritt erzeugt einen Überlauf, so dass A erneut nur 0en enthält. Die folgende Funktion führt nun genau einen Erhöhungsschritt aus.

```
Increment(A)
1. i = 0
2. while i < k and A[i] == 1 do
3.         A[i] = 0
4.         i = i + 1
5. if i < k then
6.         A[i] = 1
```

Wir analysieren nun die Laufzeit eines solchen Erhöhungsschritts. Die Worst-Case-Laufzeit ist $\Theta(k)$: Wenn alle Bits vor Aufruf der Funktion auf 1 gesetzt sind, durchläuft die while-Schleife k Iterationen, um alle k Bits auf 0 zu setzen (die weiteren Schritte haben eine konstante Laufzeit). Mit einer *amortisierten Analyse* überlegen wir uns nun, dass ein Erhöhungsschritt *im Durchschnitt* sehr viel schneller ist. Noch etwas präziser ausgedrückt schauen wir uns an, was es kostet, beginnend bei einem mit Null initialisierten Zähler mehrere Erhöhungsschritte nacheinander auszuführen. Schätzt man die Laufzeit jeden Erhöhungsschritts mit $O(k)$ ab, so erhält man eine obere Schranke von $O(k \cdot m)$ für die Ausführung von m Erhöhungsschritten. Dies ist zwar als obere Schranke korrekt, aber viel zu groß.

Die wesentliche Überlegung ist die folgende. Wir stellen fest, dass nur das niederwertigste Bit tatsächlich bei jedem Erhöhungsschritt verändert wird. Das vorletzte Bit, gespeichert in $A[1]$, wird nur jedes zweite Mal verändert, und im Allgemeinen wird $A[i]$ nur in jedem 2^i. Schritt verändert. Man kann sich dies gut an einem Beispiel verdeutlichen, betrachten wir einen Zähler mit $k = 5$, den wir sechsmal erhöhen (die Bits sind nach abnehmender Wertigkeit sortiert).

```
00000
00001
00010
00011
00100
00101
00110
```

Wir bemerken: Damit $A[i]$ verändert wird, müssen die i Bits mit niedrigerer Wertigkeit zunächst alle 2^i möglichen Zahlen durchlaufen. Innerhalb von m Operationen wird $A[i]$ daher $\lfloor m/2^i \rfloor$ Mal verändert (wobei wir

davon ausgehen, dass wir bei 0 starten) . Wir erhalten insgesamt

$$\sum_{i=0}^{k-1} \left\lfloor \frac{m}{2^i} \right\rfloor < \sum_{i=0}^{k-1} \frac{m}{2^i} < m \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2m$$

Änderungen in A und eine Gesamtlaufzeit von $O(m)$ für m Operationen. Im Durschnitt benötigt eine Erhöhung also nur eine Laufzeit von $O(1)$. Das ändert natürlich nichts daran, dass die späteren und insbesondere die allerletzte Erhöhung eine Laufzeit von $\Theta(k)$ haben.

1.2 Potenzialmethode

Wir haben in obigem Beispiel die durchschnittliche Laufzeit bestimmt, indem wir die Gesamtlaufzeit von m Operationen ausgerechnet und durch m geteilt haben. Eine solche amortisierte Analyse wird auch als *Aggregat-Methode* bezeichnet. Eine weitere Methode zur amortisierten Analyse ist als *Account-* oder *Bankkonto-Methode* bekannt. Diese überspringen wir und besprechen als nächstes die Analyse mit der *Potenzialmethode*. Diese ist etwas komplizierter, bietet dafür aber auch mehr Anpassungsmöglichkeiten für komplexere Analyseaufgaben. Wir besprechen den Binärzähler erneut, um die Methode zu erklären.

Um die Laufzeit von m Operationen abzuschätzen, verwendet die Potenzialmethode eine *Potenzialfunktion*. Diese kann man sich als eine Art Konto vorstellen, auf das während günstiger Operationen etwas ‚eingezahlt' wird, was später für teure Operationen wieder ‚ausgegeben' werden kann. (Diese Vorstellung birgt allerdings etwas Verwechslungsgefahr mit der übersprungenen Bankkonto-Methode. Bei dieser werden mehrere Konten verwendet).

Die Potenzialfunktion wird üblicherweise mit Φ bezeichnet und bildet jeden Zustand der analysierten Datenstruktur auf eine Zahl ab. Das klingt kompliziert, ist aber im Fall des Binärzählers ganz einfach: Als $\Phi(A)$ wählen wir die Anzahl der aktuell auf 1 gesetzten Stellen im Array A. Zu Beginn ist in unserem Beispiel also $\Phi(A) = 0$, da im Array noch keine 1en gespeichert sind.

Anstelle der tatsächlichen Laufzeit einer Operation analysieren wir nun die *amortisierte* Laufzeit einer Operation. Sei $A_0 = A$ und A_i der Zustand von A nach i Operationen, und sei t_i die tatsächliche Laufzeit der i. Operation. Dann ist die amortisierte Laufzeit der i. Operation definiert als

$$t_i + \Phi(A_i) - \Phi(A_{i-1}),$$

d.h. die Veränderung des Potenzials wird zur tatsächlichen Laufzeit hinzugezählt. Wenn die Potenzialfunktion durch die Operation steigt, dann ‚bezahlt' die Operation etwas zusätzlich, und wenn sie sinkt, dann wird eine eigentlich teure Operation durch das Potenzial finanziert. Zählt man die amortisierten Laufzeiten zusammen, so erhält man eine sogenannte Teleskopsumme, d.h. eine Summe, bei sehr viele Summanden wegfallen, weil sie sich gegenseitig ausgleichen:

$$\sum_{i=1}^{m} t_i + \Phi(A_i) - \Phi(A_{i-1}) = \sum_{i=1}^{m} t_i + \Phi(A_m) - \Phi(A_0) + \sum_{i=1}^{m-1} \Phi(A_i) - \Phi(A_i) = T + \Phi(A_m) - \Phi(A_0),$$

wobei wir mit T die tatsächliche Gesamtlaufzeit bezeichnen. Bisher haben wir nichts verwendet, was speziell für den Binärzähler gilt; genau so kann man amortisierte Laufzeiten auch allgemein für eine Datenstruktur D definieren, deren Zustände mit $D_0, \ldots, D_m$ bezeichnet werden, und für die dann ebenfalls gilt, dass die Summe der amortisierten Laufzeiten gerade die tatsächliche Gesamtlaufzeit plus $\Phi(D_m) - \Phi(D_0)$ ist. Wir erkennen nun, dass die Summe der amortisierten Laufzeiten eine obere Schranke für die tatsächliche Gesamtlaufzeit ist, wenn $\Phi(D_m) - \Phi(D_0) \geq 0$ gilt. Für unseren Binärzähler ist das der Fall: Es gilt $\Phi(A_0) = 0$ und $\Phi(A_m) \geq 0$, da die Anzahl der Einsen niemals negativ sein kann.

Es fehlt nun nur noch, dass wir die amortisierte Laufzeit eines Erhöhungsschritts des Binärzählers bestimmen. Wir zählen dazu wieder die Anzahl der Bitveränderungen, da die tatsächliche Laufzeit dazu proportional ist. Es gibt zwei Fälle für einen Erhöhungsschritt: Im ersten Fall sind nicht alle Bits auf 1 gesetzt. Wenn der Erhöhungsschritt t_i Bits verändert, dann sind in diesem Fall $t_i - 1$ dieser Bits vorher 1 und im Anschluss 0, und genau ein Bit wird von 0 auf 1 verändert. Die Anzahl der Einsen wird also um $t_i - 1$ verringert und um

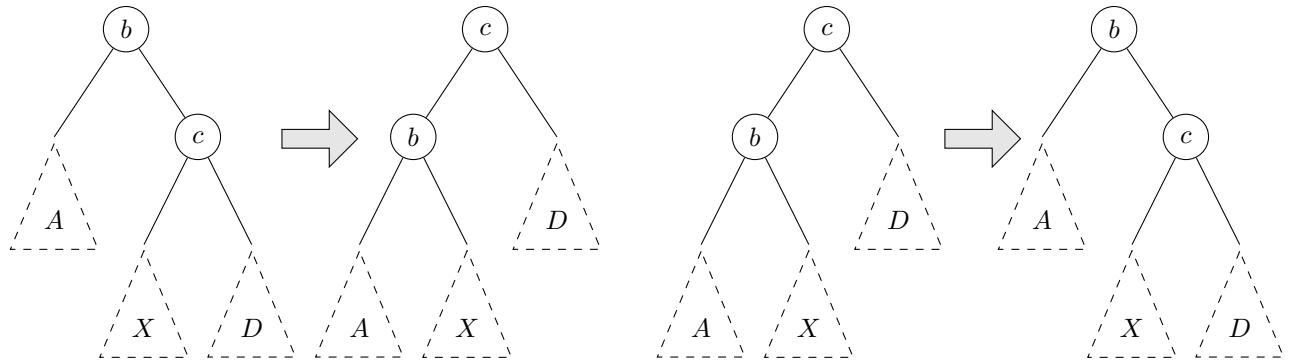

Abbildung 1: Links-Rotation an c und Rechts-Rotation an b.

1 erhöht, d.h. $\Phi(A_i) - \Phi(A_{i-1}) = -(t_i - 2)$. (Man beachte, das dies auch im allerersten Schritt stimmt und in diesem eine Erhöhung darstellt, denn es ist $t_1 = 1$ und somit $\Phi(A_i) - \Phi(A_{i-1}) = -(-1) = 1$, was ja auch stimmt.)

Wir schlussfolgern, dass die tatsächliche Laufzeit t_i ist, aber die amortisierte Laufzeit $t_i - (t_i - 2) = 2$ beträgt. Wir müssen noch prüfen, ob dies auch im zweiten Fall gilt, wenn alle Bits vor der Operation auf 1 gesetzt waren. Dann sinkt das Potenzial aber sogar um t_i, so dass die amortisierte Laufzeit nur kleiner ist.

Insgesamt stellen wir auch hier fest, dass die Summe der amortisierten Laufzeiten in $O(m)$ liegt.

2 Splay-Bäume

Wir knüpfen nun an ein früheres Thema der Vorlesung an, nämlich die Verwendung binärer Suchbäume als dynamische Wörterbücher. Wir haben bereits AVL-Bäume kennengelernt, welche die Operationen `Insert`, `Delete` und `Find` in Worst-Case-Laufzeit $O(\log n)$ durchführen können, wobei n die Anzahl der aktuell gespeicherten Schlüssel ist. Wie zuvor werden wir auch hier nicht zwischen einem Knoten und dem gespeicherten Schlüssel unterscheiden und gehen davon aus, dass die zu dem Schlüssel gehörige Information immer korrekt mitverwaltet wird und dass die Schlüssel eindeutig sind.

Jetzt lernen wir eine Datenstruktur kennen, die im Worst-Case eine sehr viel höhere Laufzeit für eine Operation aufweisen kann, nämlich $\Theta(n)$. Dafür ist sie aber einfacher zu implementieren und ist sehr gut geeignet, wenn die Schlüssel nicht gleichmäßig auftreten, sondern einige sehr viel häufiger angefragt werden als andere. Vor allem aber handelt es sich um eine verblüffende Datenstruktur, die für alle Operationen eine *amortisierte* Laufzeit von $O(\log n)$ bietet. Es handelt sich um *Splay*-Bäume. Diese Datenstruktur geht auf Sleator und Tarjan zurück [4] und die folgende Beschreibung orientiert sich lose an Skripten von Dinitz [2] und Erickson [3].

Um Splay-Bäume zu erklären, benötigen wir drei Rotationen. Einige davon kennen wir bereits: Links-Rotationen, Rechts-Rotationen, Links-Rechts-Rotationen und Rechts-Links-Rotationen haben wir bereits bei AVL-Bäumen verwendet.

Abbildung 1 zeigt die beiden einfachen Rotationen nach links und rechts. Wir beachten, dass die eingezeichneten Teilbäume A, X und D alle gleichhoch eingezeichnet sind. Dies soll nicht bedeuten, dass sie wirklich alle gleich hoch sind, wenn wir diese Operation anwenden, sondern dass wir nichts über ihre Höhe wissen. In Splay-Bäumen wird die Wahl von Rotationen davon nämlich nicht abhängen. Alle eingezeichneten Teilbäume können also leer oder von irgendeiner Höhe sein.

Dasselbe gilt auch für Abbildung 2 und Abbildung 3, die die ebenfalls von AVL-Bäumen bekannten Doppelrotationen zeigen, also die Links-Rechts-Rotation und die Rechts-Links-Rotation. Neu ist nun, dass in Splay-Bäumen auch Rotationen benötigt werden, die zweimal in dieselbe Richtung rotieren, also eine Links-Links-Rotation sowie eine Rechts-Rechts-Rotation. Wie auch die einfache Links- und Rechts-Rotation sind auch diese symmetrisch und in Abbildung 4 kombiniert abgebildet.

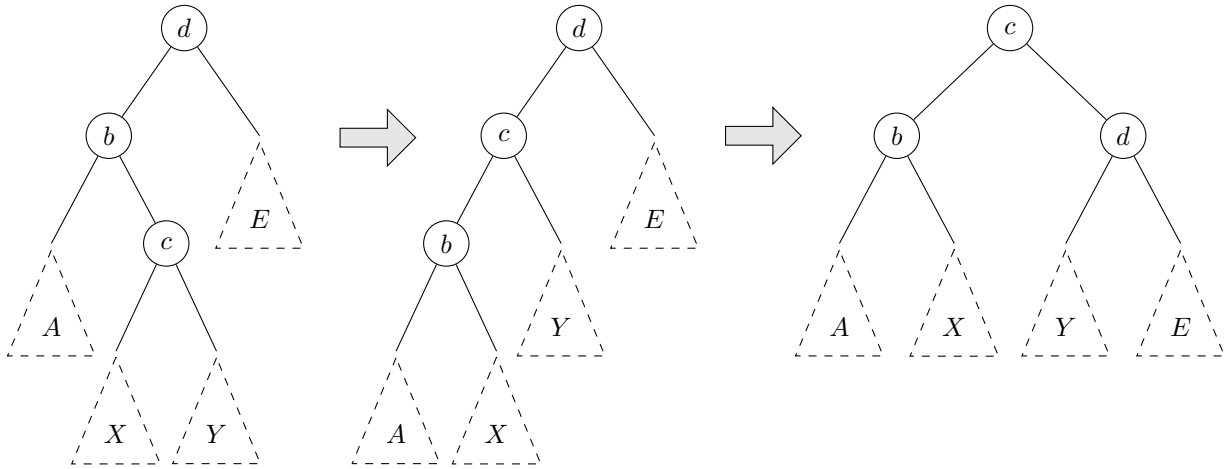

Abbildung 2: Links-Rechts-Rotation an c.

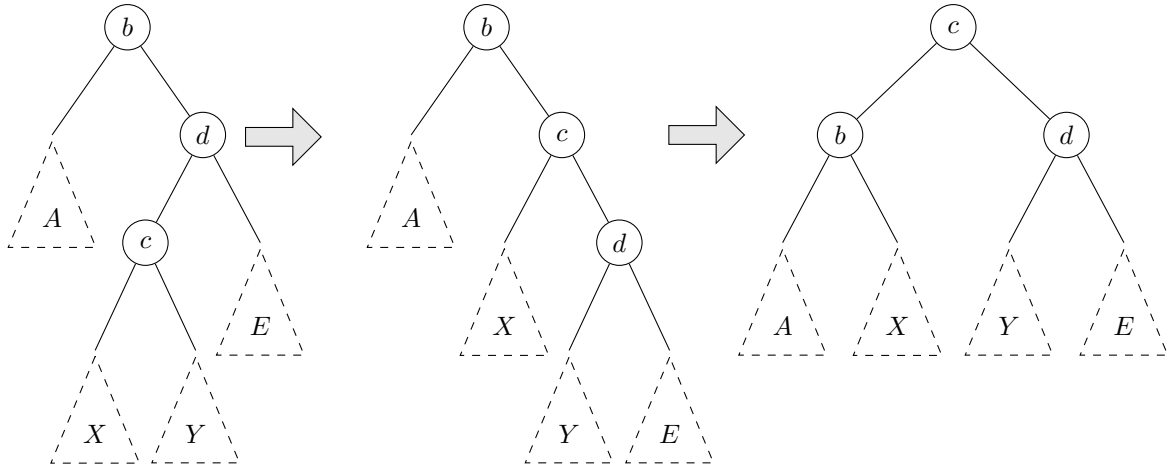

Abbildung 3: Rechts-Links-Rotation an c.

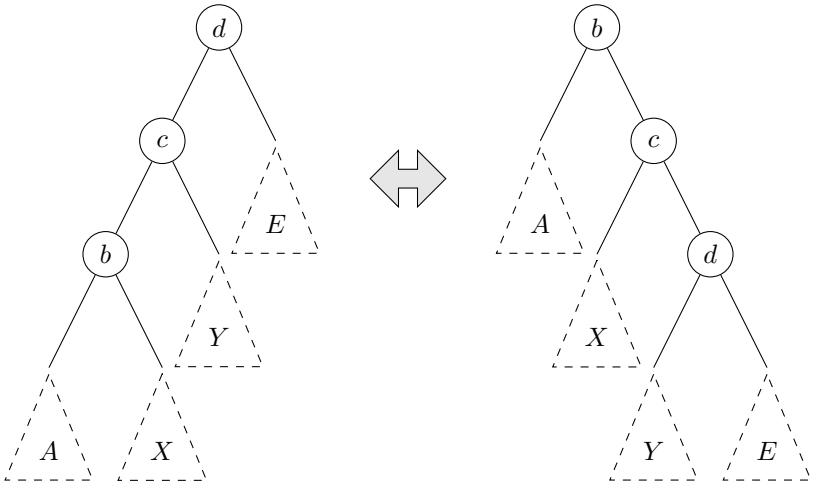

Abbildung 4: Hier sind zwei Doppelrotationen abgebildet: Von links nach rechts gelesen eine Rechts-Rechts-Rotation an b, von rechts nach links gelesen eine Links-Links-Rotation an d.

2.1 Operationen in Splay-Bäumen

Wir benötigen nur eine neue Operation, die `Splay` Operation, die auf einen Knoten angewendet wird. Wir beschreiben diese Operation im nächsten Abschnitt, wichtig ist jetzt zunächst, dass `Splay(`u`)` den Baum so umstellt, dass u im Anschluss die Wurzel des Baumes ist. Diese neue Operation wird in die bereits von (unbalancierten) binären Suchbäumen bekannten Operationen wie folgt eingefügt:

- `Find(`x`)`: Sei u der Knoten, bei dem die Suche nach x endet (also entweder x oder der Vorgänger bzw. Nachfolger von x). Rufe dann `Splay(`u`)` auf.
- `Insert(`u`)`: Füge u wie üblich in den Baum ein. Rufe dann `Splay(`u`)` auf.
- `Delete(`x`)`: Führe eine Suche nach x aus. Bei erfolgloser Suche rufe `Splay(`u`)` auf letztem besuchten Knoten u auf. Bei erfolgreicher Suche nach x rufe `Splay(`x`)` auf. x ist nun der Wurzelknoten. Wir löschen diesen, so dass der Baum in zwei Teilbäume zerfällt. Einer enthält kleinere, der andere größere Schlüssel als x. In dem Baum, der die kleineren Schlüssel enthält, suche den größten Schlüssel w. Rufe Rufe `Splay(`w`)` auf und hänge den zweiten Baum als rechtes Kind an w an.

Wir bleiben an dieser Stelle auf einem etwas höheren Beschreibungsniveau und verschieben die vollständige Realisierung dieser Methoden als Pseudocode in die Übungen. Auch wollen wir sofort argumentieren, dass wir uns im Folgenden auf die Analyse der `Splay`-Operation zurückziehen können. Dazu beobachten wir, dass `Find` und `Insert` dieselben Schritte ausführen wie in einem normalen balancierten Suchbaum plus jeweils eine Splay-Operation. Ihre Laufzeit ist also proportional zur Höhe des Baums plus der Laufzeit der Splay-Operation. `Delete` ist etwas umfangreicher. Hier wird die Laufzeit für eine `Find`-Operation, zwei Splay-Operationen und dem Finden des größten Schlüssels benötigt. Wir wissen bereits, dass die Laufzeit für letztgenanntes auch proportional zur Höhe des Baums ist, so dass wir auch hier vor allem die Splay-Operation untersuchen müssen.

Im Splay-Baum werden keinerlei explizite Maßnahmen getroffen, damit der Baum balanciert ist. Es ist auch tatsächlich möglich, dass der Baum vollkommen unbalanciert ist und eine Höhe von n hat. Die Worst-Case-Laufzeit aller Operationen ist daher $\Theta(n)$.

2.2 Die Splay-Operation

`Splay(`u`)` rotiert u in die Wurzel des Baumes. Dies geschieht iterativ abhängig davon, wo sich der Elter- und Großelterknoten von u befindet. Ist u bereits die Wurzel, geschieht nichts. Ist u nicht die Wurzel, es gibt aber keinen Großelterknoten, so wird eine einfache Links- oder Rechts-Rotation an u ausgeführt, je nachdem, ob u rechts oder links an der Wurzel angeschlossen ist. Im dritten Fall bezeichnen wir den Elterknoten von u mit e und den Großelterknoten mit g. Es gibt nun vier Möglichkeiten, wie diese zu u liegen können: u ist linkes Kind von e und e ist linkes Kind von g, u ist linkes Kind von e und e ist rechtes Kind von g, u ist rechtes Kind von e und e ist linkes Kind von g, oder u ist rechtes Kind von e und e ist rechtes Kind von g. Abhängig davon wird (in passender Reihenfolge aufgezählt) eine Rechts-Rechts-Rotation, eine Rechts-Links-Rotation, eine Links-Rechts-Rotation oder eine Links-Links-Rotation ausgeführt.

Durch die Ausführung der einfachen oder doppelten Rotation steigt u um ein oder zwei Level auf. Solange u nicht die Wurzel ist, wird der Schritt wiederholt. Wir beobachten, dass Splay(u) entweder keine oder genau eine einfache Rotation verwendet, und ansonsten Doppelrotationen benutzt.

Jede Rotation benötigt eine konstante Anzahl an Schritten und hat somit eine konstante Laufzeit. Außerdem ist die Laufzeit von `Splay(`x`)` asymptotisch mindestens so groß wie die Suche nach x vor der `Splay`-Operation, da wir den Suchpfad zurücklaufen, während wir die Rotationen durchführen. Da alle im letzten Abschnitt beschriebenen Operationen `Splay` nur auf Knoten aufrufen, die zuvor gesucht wurden, und weil jede Operation nur eine konstante Anzahl von `Splay`-Operationen verwendet, entspricht die Laufzeit aller beschriebenen Operationen asymptotisch der Laufzeit einer `Splay`-Operation.

2.3 Ein Beispiel

Bevor wir uns der amortisierten Analyse zuwenden, wollen wir uns ein einfaches Beispiel ansehen. Dabei werden wir zwei Dinge beobachten: Erstens ist es tatsächlich möglich, dass eine Operation Laufzeit $\Omega(n)$ hat, weil der

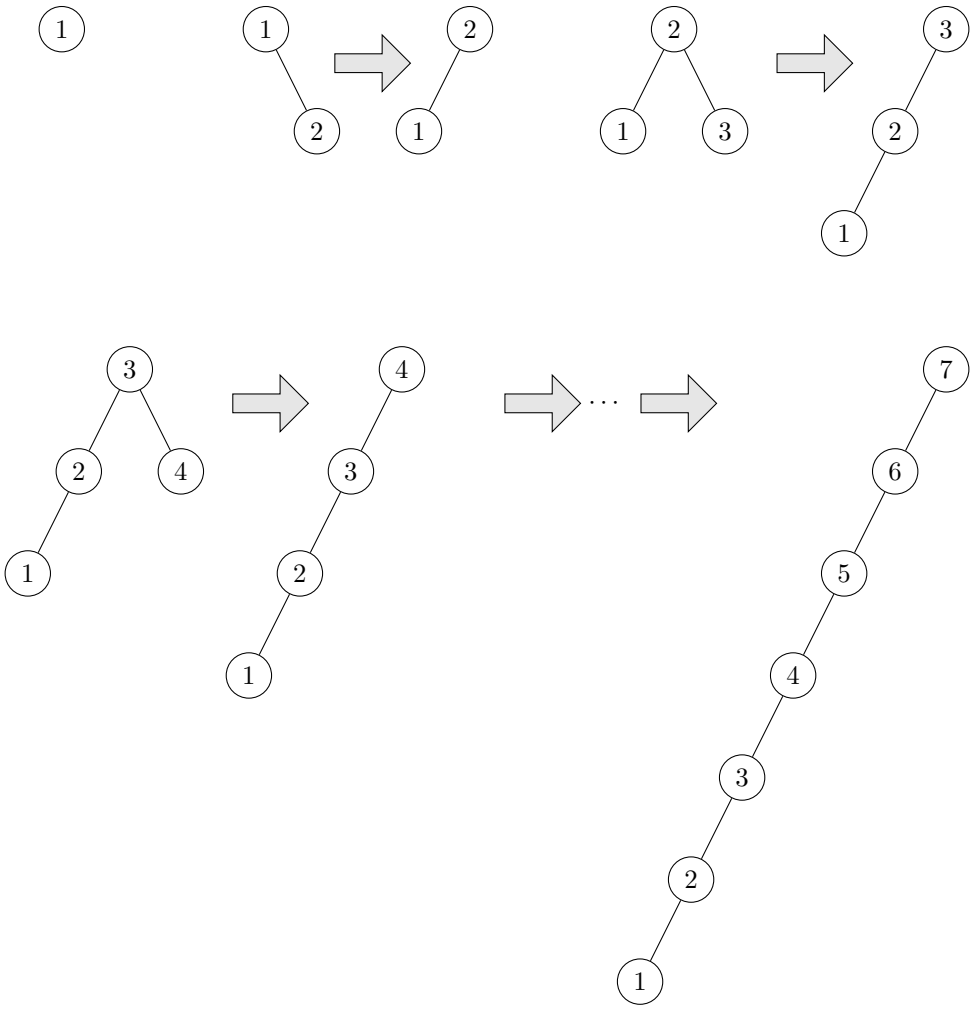

Abbildung 5: Eine Einfügesequenz, die einen Splay-Baum erzeugt, der ein Pfad ist.

Baum unbalanciert ist. Zweitens werden wir beobachten, dass –in diesem Beispiel– die `Splay`-Operation diese Unbalanciertheit sehr effizient behebt. Wir fügen in einen zunächst leeren Suchbaum die Schlüssel 1, 2, 3, 4, 5, 6 und 7 in dieser Reihenfolge ein. Dies ist in Abbildung 5 abgebildet. Nach dem Einfügen jedes Schlüssels u wird `Splay(`u`)` ausgeführt. In unserem Beispiel führt dies jedes Mal zu einer Links-Rotation, so dass der Splay-Baum immer ein Pfad bleibt. Statt 7 Schlüssel einzufügen, kann man im Allgemeinen m aufsteigende Schlüssel einfügen und erhält einen Pfad der Länge m.

Die Laufzeit der Einfügeoperationen ist allerdings nicht lang: Für jeden Schlüssel werden konstant viele Schritte benötigt, um ihn in den Baum einzuhängen und dann eine Links-Rotation durchzuführen. Ein Problem (für die amortisierte Laufzeit) wäre nun, wenn wir in diesem Pfad sehr häufig teure Operationen ausführen würden. In einem normalen binären Suchbaum könnten wir zum Beispiel immer wieder nach dem kleinsten Schlüssel suchen, was immer wieder Zeit $\Theta(m)$ in Anspruch nehmen würde. Dies geht in einem Splay-Baum aber nicht, denn hier wird ja auch bei einer Suchoperation der Baum verändert. Tatsächlich sehen wir in Abbildung 6, dass das Suchen nach 1 die Höhe des Splay-Baums bereits merklich verkleinert.

Rufen wir nun `Find(`3 auf, um erneut eine möglichst große Suchzeit auszulösen, können wir eine Links-Rechts-Rotation und eine Rechts-Links-Rotation beobachten (siehe Abbildung 7). Die Höhe des Baumes verringert sich nochmal um eins, so dass er nun fast vollständig balanciert ist.

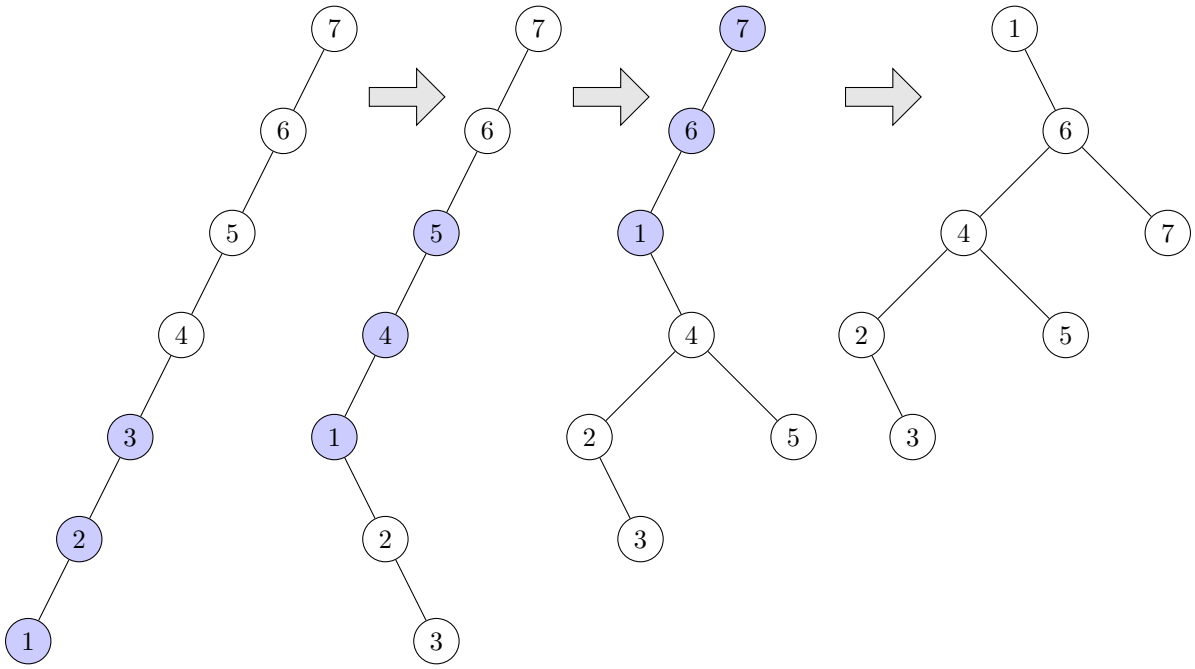

Abbildung 6: Zum Abschluss von `Find(1)` wird `Splay(1)` ausgeführt. Dies löst drei Rechts-Rechts-Rotationen aus, jeweils an 1. Die beteiligten Knoten sind in der Abbildung eingefärbt.

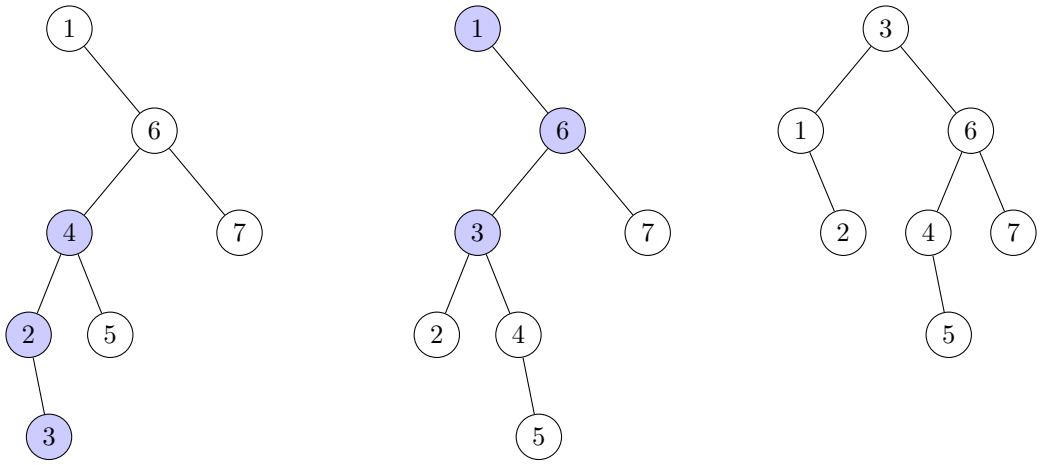

Abbildung 7: Zum Abschluss von `Find(3)` wird `Splay(3)` ausgeführt. Dies löst eine Links-Rechts-Rotation und eine Rechts-Links-Rotation aus, jeweils an 3. Die beteiligten Knoten sind in der Abbildung eingefärbt.

2.4 Die amortisierte Laufzeit einer `Splay`-Operation

Intuitiv können wir uns die Funktionsweise eines Splay-Baums so vorstellen: Wenn es einen langen Pfad im Baum gibt, so hat es viele Operationen benötigt, diesen zu erstellen. Dieses spiegelt sich im Potenzial wieder. Wird nun eine Operation aufgerufen, die diesen langen Pfad verwendet, dann ist das in der amortisierten Laufzeit zunächst nicht schlimm, da die vielen vorangegangenen Operationen die Kosten eines Durchlaufs dieses Pfades abdecken. Durch die `Splay`-Operation wird die Länge ausreichend verringert, und das Potenzial sinkt wieder.

Diese Intuition ist natürlich zu ungenau, um wirklich etwas darüber auszusagen, was im Splay-Baum passiert. Wir wenden nun die amortisierte Analyse nach der Potenzialmethode an.

Wir analysieren wir eine Abfolge von m Operationen, die auf einen anfangs leeren Baum angewendet werden, der während des gesamten Ablaufs maximal n Knoten enthält. Zur Vereinfachung analysieren wir nur die Anzahl der ausgeführten Rotationen, wobei wir einfache Rotationen und Doppelrotationen gleich behandeln.

Definition 1. *Sei $T = (V(T), E(T))$ ein binärer Suchbaum. Für ein $u \in V(T)$ bezeichnen wir mit*

- *`size(u)` $= s(u)$ die Anzahl der Knoten in dem Teilbaum, dessen Wurzel u ist und setzen*
- *`rank(u)` $= r(u) = \lfloor \log(s(u)) \rfloor$.*

Das Potenzial von T definieren wir durch

$$\Phi(T) = \sum_{u \in V(T)} \texttt{rank(u)}.$$

Ein vollständiger Binärbaum der Höhe i enthält $2^{i+1} - 1$ Knoten. Die Wurzel hat daher Rang i, ihre Kinder haben Rang $i - 1$, und so weiter. Die Blätter haben Rang 0. In einem Pfad mit $2^{i+1} - 1$ Knoten hingegen hat nicht nur die Wurzel Rang i, sondern insgesamt 2^i der Knoten. Die Summe der Ränge sagt also etwas darüber aus, wie (un)balanciert der Baum ist.

Lemma 2. *Durch eine Links- oder Rechts-Rotation am Knoten u erhöht sich $\Phi(T)$ um maximal $r'(u) - r(u)$, wobei $r'(u)$ der Rang von u nach der Rotation ist und $r(u)$ der Rang vor der Rotation. Durch eine Links-Rechts-, Rechts-Links-, Links-Links- oder Rechts-Rechts-Rotation am Knoten u erhöht sich $\Phi(T)$ um maximal $2(r'(u) - r(u)) - 1$.*

Beweis. Wir beobachten zuerst, dass bei einer Rotation nur die Ränge der beteiligten zwei bzw. drei Knoten verändert werden können. Für alle anderen Knoten ändert sich nichts an der Anzahl der Knoten im Teilbaum.

- Sei e der Elter von u bei einer Links- oder Rechts-Rotation. Dann ist die Potenzialänderung $r'(u) + r'(e) - r(u) - r(e)$. Nach der Rotation ist u die Wurzel des Teilbaums, dessen Wurzel e vorher war, daher gilt $r'(u) = r(e)$. Außerdem gilt $r'(e) \leq r'(u)$, da e nach der Rotation Kind von u ist. Also ändert sich das Potenzial um maximal $r'(u) - r(u)$.
- Sei e der Elter und g der Großelter bei einer Doppelrotation. Wir machen eine Fallunterscheidung. Sei zuerst $r(g) = r(e) = r(u) = r$, der Baum ist also lokal unbalanciert und alle drei beteiligten Knoten haben zu Beginn denselben Rang. Nach der Rotation ist u die Wurzel, also gilt $r'(u) = r(g)$. Außerdem muss der Rang entweder für g oder für e gesunken sein, da beide nun Kinder von u sind: Hätten sowohl g als auch e Rang r, so hätten die beiden Teilbäume jeweils mindestens 2^r Knoten, so dass der Teilbaum, dessen Wurzel u ist, mindestens $2^r + 1$ Knoten hätte. Dies kann nicht sein. Da beide Ränge auch weiterhin maximal r sind und einer strikt kleiner ist, gilt also $r'(g) + r'(e) \leq 2r - 1$. Dementsprechend ist die Potenzialänderung begrenzt durch

  $$r'(u) + r'(e) + r'(g) - r(u) - r(e) - r(g) = r'(e) + r'(g) - r(u) - r(e) \leq 2r - 1 - 2r = -1.$$

  Dies ist durch $2(r'(u) - r(u)) - 1$ beschränkt, da $r'(u) - r(u) \geq 0$ ist (tatsächlich ist ja $r'(u) = r = r(u)$, also ist $r'(u) - r(u) = 0$).

  Sei nun $r(g) > r(u)$. Die Potenzialänderung ist wieder $r'(e) + r'(g) - r(u) - r(e)$, und es gilt $r(e) \geq r(u)$, da e der Elter von u ist. Wie zuvor argumentieren wir, dass u, e und g nach der Rotation nicht alle denselben

Rang haben können, da u der Elter von e und g ist. Daher gilt $r'(e) + r'(g) \leq 2r'(u) - 1$. Wir erhalten also eine Potenzialänderung von maximal

$$2r'(u) - 1 - 2r(u) = 2(r'(u) - r(u)) - 1.$$

□

Theorem 3. *`Splay(u)` hat eine amortisierte Laufzeit von $O(\log n)$, wobei n die Anzahl der Knoten im Suchbaum ist.*

Beweis. Eine `Splay`-Operation führt maximal eine einfache Rotation aus und davor ausschließlich Doppelrotationen. Die Laufzeit der Operation ist proportional zur Anzahl der Rotationen, deshalb zählen wir jetzt nur Rotationen und setzen für eine Rotation Laufzeit 1 an. Dann ist die amortisierte Laufzeit einer Doppelrotation am Knoten u nach Lemma 2 höchstens

$$1 + 2(r'(u) - r(u)) - 1 \leq 2(r'(u) - r(u)),$$

die einfache Rotation hat eine amortisierte Laufzeit von $2(r'(u) - r(u)) + 1$, wobei r' die Rangfunktion nach der Doppelrotation ist und r die davor. Nehmen wir an, dass k Doppelrotation und 1 einfache Rotation ausgeführt werden (ohne die abschließende einfache Rotation ist es nur billiger), und dass die Rangfunktion vor der `Splay`-Operation r_0 heißt und die nach der i. Doppelrotation r_i. Dann erhalten wir eine Teleskopsumme für die amortisierte Laufzeit der k Doppelrotationen, die alle am Knoten u stattfinden, der langsam im Baum hinaufwandert:

$$\sum_{i=1}^{k} 2(r_i(u) - r_{i-1}(u)) = 2(r_k(u) - r_0(u)).$$

Durch die abschließende einfache Rotation erhalten wir die Rangfunktion r_{k+1}. Addieren wir die amortisierte Laufzeit dieser Rotation noch hinzu, erhalten wir als obere Schranke $2(r_{k+1}(u) - r_0(u))$.

Nun sind wir fertig: Der Rang eines Knotens ist immer beschränkt durch $O(\log n)$, und $r_0(u)$ ist nicht-negativ. Also ist die amortisierte Laufzeit der `Splay`-Operation durch $O(\log n)$ beschränkt. □

Da wir mit einem leeren Baum starten, ist das Potenzial zu Beginn Null. Das Potenzial wird auch nie negativ. Die Gesamtlaufzeit von m Operationen auf einem Baum, der leer beginnt und in dem während der Operationen immer maximal n Schlüssel gespeichert sind, ist also $O(m \log n)$.

Literatur

[1] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms, 3rd Edition.* MIT Press, 2009.

[2] Dinitz, Michael: *Lecture notes on Introduction to Algorithms, Lecture 8: Splay Trees.* `http://www.cs.jhu.edu/~mdinitz/classes/IntroAlgorithms/Fall2016/Lectures/lecture8.pdf`, 2016.

[3] Erickson, Jeff: *Lecture notes on Algorithms, Lecture 16: Scapegoat and Splay Trees.* `https://jeffe.cs.illinois.edu/teaching/algorithms/notes/10-scapegoat-splay.pdf`, 2013.

[4] Sleator, Daniel D. und Robert E. Tarjan: *Self-Adjusting Binary Search Trees.* Journal of the ACM, 32(3):652–686, 1985.