Algorithmen und Datenstrukturen Hausarbeit: Vergleich von Bäumen

Hugo Protsch - 2510833

3. Februar 2021

Inhaltsverzeichnis

Einleitung

In der ersten Praktikumsaufgabe haben wir binäre Suchbäume betrachtet. Wir haben festgestellt, dass diese im Vergleich zu verketteten Listen eine erheblich bessere Laufzeit aufweisen, da beim Traversieren des Baumes nur über die Höhe des Baumes gelaufen wird, statt, wie bei der Liste, über jedes Element.

Es fiel jedoch auf, dass der Vorteil beim Einfügen von aufsteigenden und absteigenden Zahlen verloren geht. Hierbei entsteht ein Baum, der entartet, also nicht balanciert, ist. Im Extremfall entsteht somit eine einfach verkettete Liste. Die Laufzeit von Einfüge-, Lösch- und Suchoperationen ist in diesem Fall wieder linear. Um eine solche Entartung der Bäume und die daraus resultierende verminderte Performance zu vermeiden, muss also eine Strategie entwickeln werden, die den Baum balanciert hält. Ein Ansatz, der in Abschnitt 1 betrachtet wird, ist der AVL-Baum, der jeweils beim Einfügen und Löschen von Elementen die Balance des Baumes überprüft und im Falle der Entartung diese sofort ausgleicht.

Eine zweite Variation des binären Suchbaumes, die in Abschnitt 2 betrachtet wird, ist der Splay-Baum. Dieser hat das Ziel, Elemente, auf die häufig zugegriffen wird, im Baum möglichst weit oben zu halten und somit die Zugriffszeit auf diese zu verringern. Anders als beim AVL-Baum, wird ein Splay-Baum nicht balanciert, somit kann der Baum, wie ein normaler Binärbaum, entarten. Der Vorteil liegt also lediglich beim wiederholten Zugriff auf eine kleine Teilmenge der im Baum befindlichen Elementen.

In beiden Abschnitten wird zunächst der Algorithmus beschrieben, anschließend der Entwurf zur Implementation vorgestellt. Außerdem in Abschnitt 3 eine Laufzeitmessung durchgeführt, und mithilfe einer Trendlinie die Komplexität bestimmt. Des Weiteren werden die beiden Algorithmen miteinander und mit einem normalen Binärbaum verglichen.

1 AVL-Baum

1.1 Algorithmus

1.1.1 AVL-Bedingung

Ein AVL-Baum ist ein Binärbaum, der die zusätzliche Eigenschaft besitzt, dass die Balance, die als die Differenz der Höhe h der beiden Teilbäume definiert ist (siehe Formel 1), bei jedem Knoten mindestens 1 und maximal 1 beträgt. Diese Eigenschaft wird AVL-Bedingung genannt. Dabei ist die Höhe analog zum regulären Binärbaum definiert (siehe Formel 2).

$$bal(k) = h(T_r) - h(T_l) \in \{-1, 0, 1\} \tag{1}$$

$$h(k) = max(h(T_l), h(T_r)) \tag{2}$$

Durch diese Bedingung wird sichergestellt, dass der Baum zu jedem Zeitpunkt balanciert ist und somit das in der Einleitung beschriebene Problem der schlechten Laufzeit des Binärbaumes durch Entartung nicht auftreten kann.

1.1.2 Rebalancierung

Nach dem Einfügen und Löschen von Elementen kann es jeweils vorkommen, dass die Balance eines Konten -2 oder 2 beträgt. Somit muss der AVL-Baum nach diesen Operationen die AVL-Bedingung überprüfen, und eventuell eine Rebalancierung vornehmen. Dabei wird zwischen insgesamt vier Fällen unterschieden, die durch die Folge der Balancewerte definiert sind (siehe auch Abbildung 1a):

1. Left Left: -2/-1 oder -2/0 → Rechtsrotation
2. Right Right: +2/+1 oder +2/0 → Linksrotation
3. Left Right: -2/+1 → Doppelte Rechtsrotation
4. Right Left: +2/-1 → Doppelte Linksrotation

Die ersten beiden und letzten beiden Fälle sind dabei jeweils symmetrisch zueinander.

Rotation Die Rebalancierung wird mithilfe von Rotationen von Knoten vorgenommen. Bei einer Rotation wird immer ein Knoten als Wurzelknoten betrachtet. Alle Knoten, die über dem Wurzelknoten stehen, sind für die Rotation nicht von Relevanz. Der Wurzelknoten wird mit dem Kindknoten rotiert, auf dessen Seite die Unbalance vorliegt, im Folgenden wird dieses Kind Rotationsknoten genannt:

- Linksrotation: Rotieren vom Wurzelknoten mit rechtem Kindknoten
- Rechtsrotation: Rotieren vom Wurzelknoten mit linken Kindknoten

Betrachte Abbildung 1a (Left Left Case). Zum Rotieren der beiden Knoten nach rechts werden folgende Operationen ausgeführt:

1. Der Wurzelknoten (5) wird als rechtes Kind vom Rotationsknoten (4) gesetzt
2. Der eben ersetzte Knoten (C) (rechtes Kind vom Rotationsknoten) wird als linkes Kind des ursprünglichen Wurzelknotens (5) gesetzt
3. Der Rotationsknoten (4) wird als neue Wurzel gesetzt

Dabei muss die Höhe des Rotations- und Wurzelknotens angepasst werden. Dies lässt sich am leichtesten durch eine erneute Berechnung mit Formel 2 erzielen. Des Weiteren muss eventuell die Höhe der Knoten über dem Wurzelknoten angepasst werden.

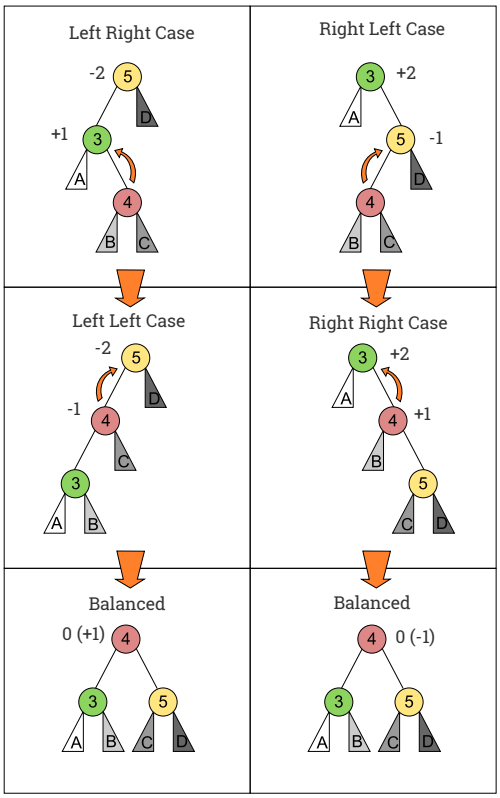
An Abbildung 1a (Balanced) wird deutlich, dass durch die Rechtsrotation beim Left Left Case die AVL-Bedingung wieder erfüllt ist.

Eine Linksrotation erfolgt symmetrisch.

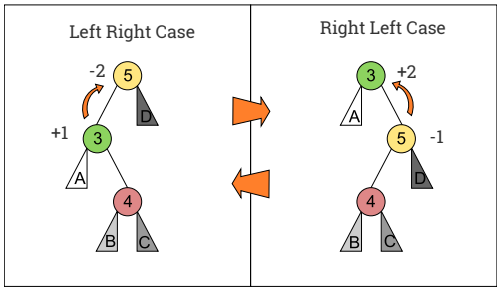
1.1.3 Doppelrotation

Bei den Fällen Left Left und Right Right konnte der Baum mit lediglich einer Rotation balanciert werden. Dies reicht bei den anderen Fällen nicht aus, wie wird aus Abbildung 1b ersichtlich wird.

Es müssen insgesamt zwei Rotationen durchgeführt werden: Mit der ersten Rotation wird der Left Left bzw. Right Right Case herbeigeführt, die zweite Rotation befriedigt anschließen die AVL-Bedingung. Betrachte Abbildung 1a (Left Right Case). Im Left Right Case wird zunächst eine Linksrotation zwischen dem linken Kindknoten des Wurzelelementes und des rechten Nachfolgers ausgeführt. Diese Rotation wird symmetrisch zur oben beschriebenen Rechtsrotation ausgeführt. Anschließend liegt der Left Left Case vor, eine einfache Rechtsrotation befriedigt nun die AVL-Bedingung.



(a) Korrekt



(b) Falsch

Abbildung 1: Rebalancierung

1.2 Entwurf

1.2.1 InitBT, IsEmptyBT, EqualBT, FindBT

Die Implementation dieser Methoden kann aus der Implementation des Binärbaumes übernommen werden. Bei der InitBT Methode müssen lediglich alle Counter der Rotationszählung zurückgesetzt werden.

1.2.2 IsBT

Die Methode von dem Binärbaum wird um die Überprüfung der AVL-Bedingung erweitert. Dabei wird bei jedem Knoten zusätzlich überprüft, ob die Balance (siehe Formel 1) -1, 0 oder +1 beträgt. Der Ausdruck wird mit den bisherigen Ausdrücken und-verknüpft.

1.2.3 InsertBT, DeleteBT

Das Einfügen und Löschen wird wie bei einem Binärbaum realisiert. Die einzige Änderung, die vorgenommen werden muss, ist das Prüfen der AVL-Bedingung und eventuelles Rotieren, bottom-up nach dem Einfügen bzw. Löschen. Dies wird mithilfe der `Rebalance`-Methode vorgenommen, die in Abschnitt 1.2.5 weiter ausgeführt wird. Beim Löschen muss darauf geachtet werden, dass der Pfad

vom tatsächlich entfernten Knoten, also dem Substitution-Knoten, überprüft wird. In Abbildung 2 und 3 sind die notwendigen Änderungen jeweils blau markiert.

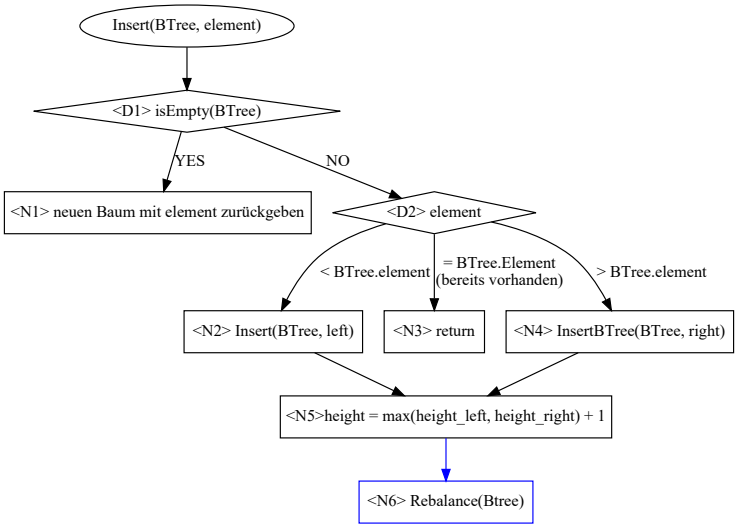


Abbildung 2: InsertBT

1.2.4 PrintBT

Diese Funktion speichert eine Representation des übergebenen Baumes als Datei im GraphViz Format. Dafür wird am Anfang einmal der Header (`digraph G{`), dann der Inhalt, am Ende eine schließende Klammer ausgegeben. Um den Inhalt auszugeben werden für jeden Knoten bis zu drei Zeilen ausgegeben:

1. `<Knoten> -> <Linkes Kind>;`
2. `<Knoten> -> <Rechtes Kind>;`
3. `<Knoten> [label = "<Knoten>\n(<Hoehe von Knoten>)" color = <Farbe>];`

Dabei entspricht `Knoten, Linkes Kind, Rechtes Kind` dem Wert des jeweiligen Knotens. Falls ein Kindknoten leer sein sollte, wird dafür nur die letzte Zeile ausgegeben. Diese formatiert den Knoten selber, dabei wird die Farbe abhängig der Balance gesetzt.

Knoten mit der Balance von -1 werden violett, mit der Balance von $+1$ blau, mit der Balance von 0 schwarz dargestellt. Falls die AVL-Bedingung nicht eingehalten wird, erscheint der Knoten rot.

1.2.5 Rebalance

Die `Rebalance`-Methode bekommt einen Knoten übergeben, überprüft die AVL-Bedingung und balanciert den Knoten durch Rotieren, falls die AVL-Bedingung nicht erfüllt ist. Dafür wird zunächst die Balance des übergebenen Knotens überprüft. Falls diese -1, 0, +1 beträgt, wird der Knoten nicht verändert. Andernfalls wird überprüft, welcher der in Abschnitt 1.1.2 aufgeführten Fälle vorliegt

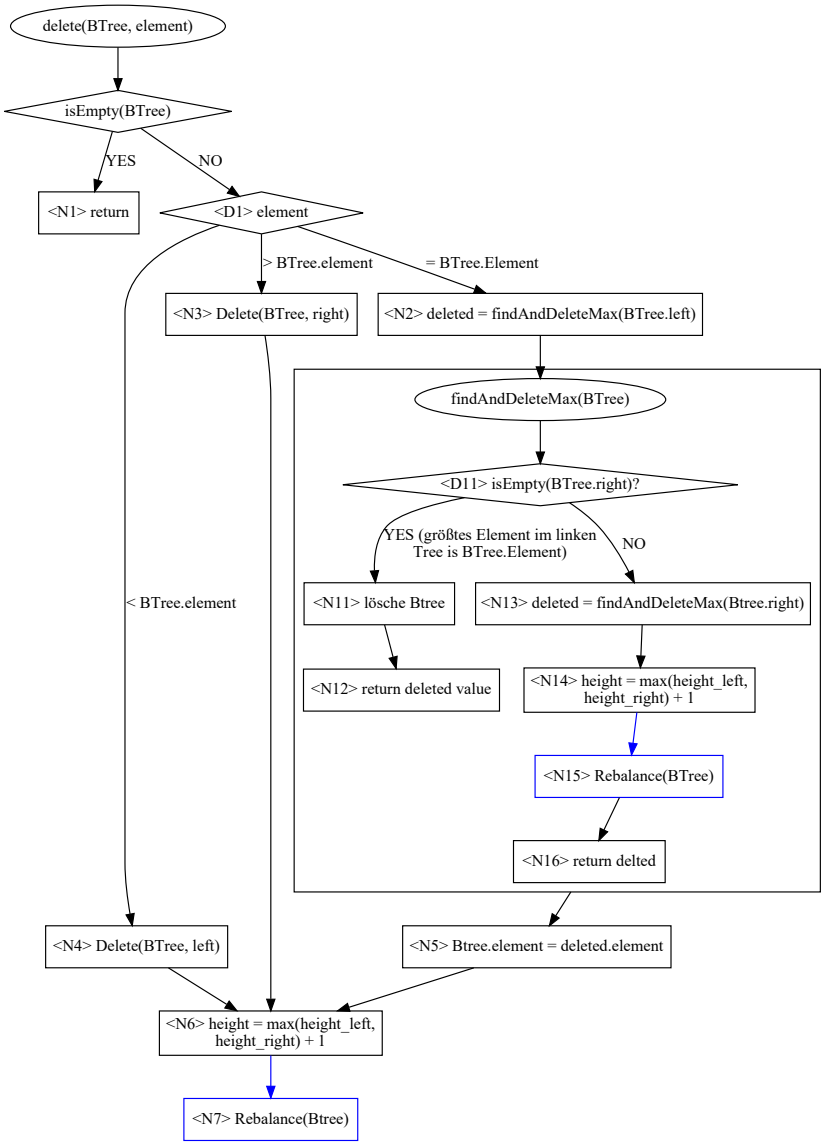


Abbildung 3: DeleteBT

und die nötigen Rotationen mithilfe der Rotate-Methode ausgeführt. Falls eine Doppelrotation ausgeführt wird, wird außerdem der entsprechende Counter erhöht. In Abbildung 4 wird die Methode in einem Flussdiagramm dargestellt.

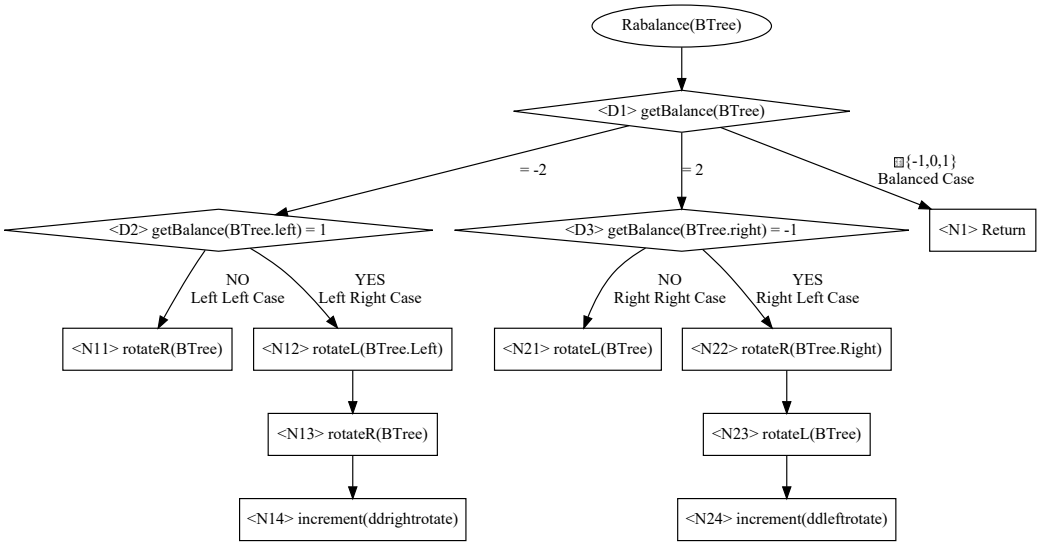
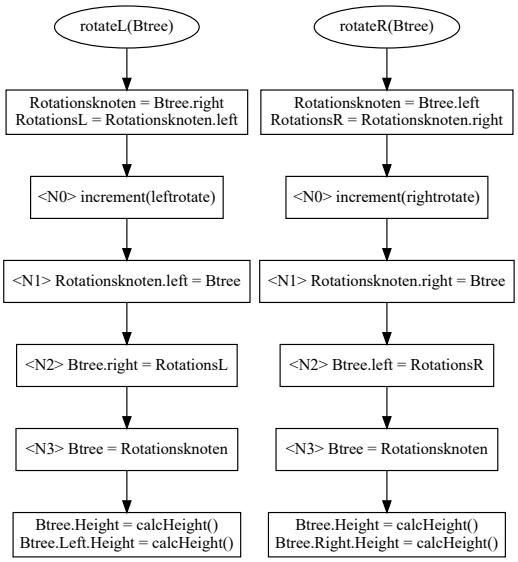


Abbildung 4: Rebalancierung

1.2.6 Rotate

Das Rotieren wird wie in Abschnitt 1.1.2 beschrieben implementiert. In Abbildung 5 wird die Rechts- und Linksrotation zusätzlich in jeweils einem Flussdiagramm dargestellt.



(a) Links (b) Rechts

Abbildung 5: Rotieren

1.3 Aufgaben

1.3.1 Aufgabe 1.4 - Analysieren beispielhafter Bäume

Beispielhafte Bäume werden mit der zur verfügung gestellten Methode `analyseBT:test(<Anzahl Elemente>)` analysiert.

100 Elemente Ein Aufruf mit 100 Elementen ergibt folgende Ausgabe:

```
>analyseBT:test(100).
Bei gegebener Höhe 8:
        minimale Anzahl Knoten: 54.
        maximale Anzahl Knoten: 255
Bei gegebener Anzahl an Knoten 100:
        minimale Höhe: 6.
        maximale Höhe: 9
Durchschnittliche Balancefehler: 0.28
        maximaler Balancefehler: 1.
```

An der Ausgabe ist zu erkennen, dass der erzeugte Baum mit 100 Elementen eine Höhe von 8 aufweist. Dies liegt innerhalb des theoretischen Rahmens für AVL Bäume der Elementanzahl 100 von mindestens 6 und maximal 9. Die Ausgabe zeigt außerdem die minimale und maximale Elementanzahl für einen Baum der vorgegebenen Höhe an, die hier bei 54 und 255 liegt. Der maximale Balancefehler beträgt 1, somit wurde die AVL-Bedingung bei jedem Knoten eingehalten. Der durchschnittliche Balancefehler beträgt 0.28, somit hatten 28 der 100 Knoten eine Balance von -1 oder 1.

1000 Elemente Ein Aufruf mit 1000 ergibt folgende Ausgabe:

```
>analyseBT:test(1000).
Bei gegebener Höhe 12:
         minimale Anzahl Knoten: 376.
         maximale Anzahl Knoten: 4095
Bei gegebener Anzahl an Knoten 1000:
         minimale Höhe: 9.
         maximale Höhe: 14
Durchschnittliche Balancefehler: 0.321
         maximaler Balancefehler: 1.
```

Aus der Ausgabe wird ersichtlich, dass die AVL-Bedingung auch bei 1000 Elementen eingehalten wird. Des Weiteren wird die logarithmische Natur des AVL-Baumes ersichtlich.

Fehlerhafter Baum Ein Aufruf mit einem Baum, der an einem Knoten eine Balance von 2 besitzt, ergibt folgende Ausgabe:

```
>analyseBT:analyseBT({3, 3, {}, {4, 2, {}, {5, 1, {}, {}}}}).
Bei gegebener Höhe 3:
         minimale Anzahl Knoten: 4.
         maximale Anzahl Knoten: 7
Bei gegebener Anzahl an Knoten 3:
         minimale Höhe: 2.
         maximale Höhe: 1
Durchschnittliche Balancefehler: 1.0
         maximaler Balancefehler: 2.
```

Der maximale Balancefehler beträgt wie erwartet 2, außerdem liegt die Höhe des Baumes über dem theoretischen Maximum. Der Baum ist somit kein AVL-Baum.

1.3.2 Aufgabe 1.5 - Löschen von 88 Prozent der Zahlen

Es wird ein Baum aus 100 Zufallszahlen erstellt, anschließen werden zufällig 88 davon wieder gelöscht. In Abbildung 6 ist der Baum vor dem Löschen, in Abbildung 7 nach dem Löschen zu sehen. In beiden Abbildungen wird die AVL-Bedingung erfüllt.

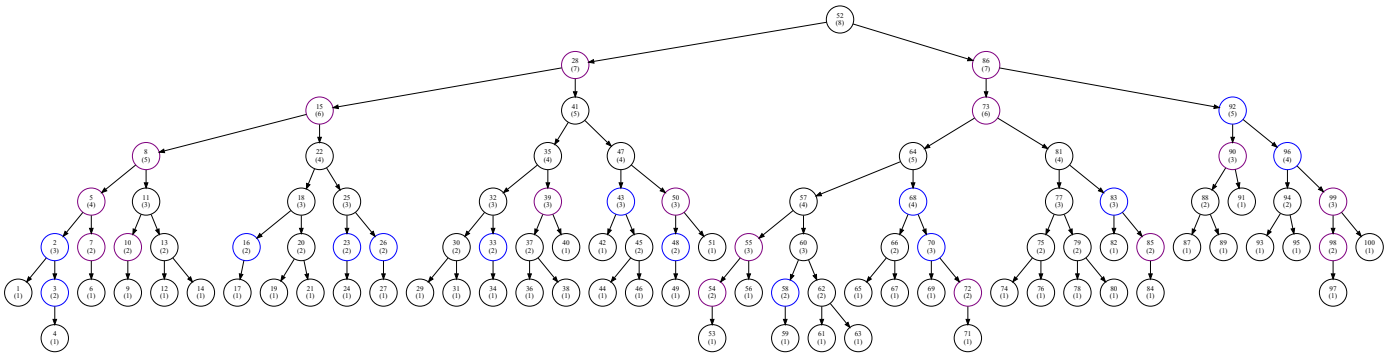


Abbildung 6: Vor dem Löschen

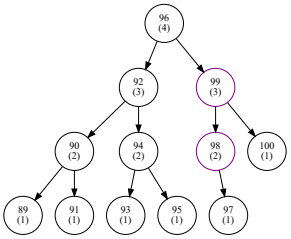


Abbildung 7: Nach dem Löschen

2 Splay-Baum

Das Ziel eines Splay-Baumes ist es, die Zugriffszeit auf häufig verwendete Elemente zu verringern, in dem diese beim Zugriff nach oben rotiert werden. Dies könnte vor allem Anwendungen, in denen nur auf eine kleine Menge aller Elemente zugegriffen wird, beschleunigen.

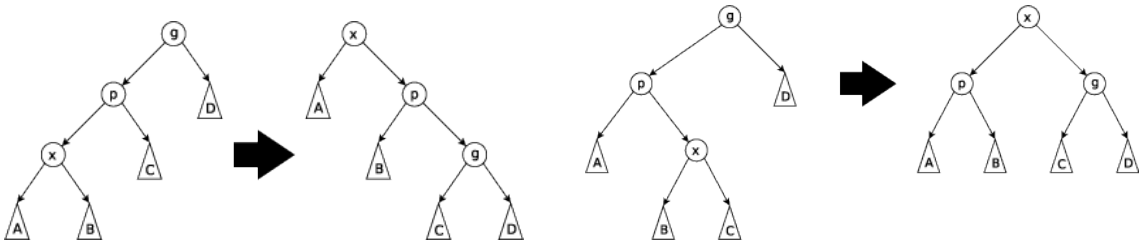
2.1 Algorithmus

Ein Splay-Baum ist ein Binärbaum, der beim Einfügen und Suchen von Elementen, diese jeweils an die Wurzel befördert. Dieser Vorgang wird Splaying genannt.

2.1.1 Splaying

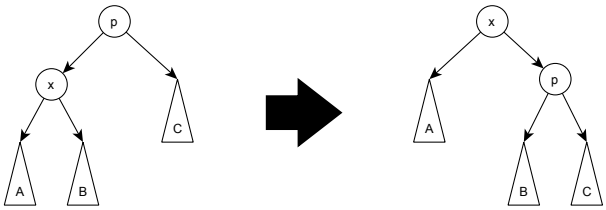
Ähnlich wie beim AVL-Baum wird wieder ein Knoten als Wurzel betrachtet. Zunächst wird das gesuchte Element an diese Wurzel gebracht. Dieser Prozess wird anschließend rekursiv auf die verbleibenden Knoten über der Wurzel angewendet, bis das Element an der Wurzel des kompletten Baumes steht.

Es wird zwischen 3 Fällen unterschieden, die durch den Pfad von der Wurzel zu dem nach oben zu befördernden Elementes definiert sind. Alle Fälle haben jeweils ein symmetrisches Paar. In Abbildung 8 ist jeweils eines der Paare dargestellt. In Klammern ist der symmetrische Fall angegeben. Die Wurzel `x` ist dabei jeweils die, die nach oben befördert werden soll. Der Vorgang wird im Entwurf der Methode Splay (Abschnitt 2.2.4) weiter ausgeführt.



(a) Zig Zig: L/L (R/R)

(b) Zig Zag: L/R (R/L)



(c) Zig: L (R)

Abbildung 8: Splaying Vorgang

2.2 Entwurf

2.2.1 InitBT, IsEmptyBT, EqualBT, PrintBT

Wie beim AVL-Baum kann auch hier die Implementation des Binärbaumes verwendet werden. Lediglich InitBT wird wieder um das Zurücksetzen der Counter erweitert. PrintBT ist identisch zu der in Abschnitt 1.2.4 beschrieben Methode.

2.2.2 IsBT

Bei dem AVL-Baum wurde IsBT um das Überprüfen der AVL-Bedingung erweitert. So konnte die korrekte Arbeitsweise des AVL-Baumes sichergestellt werden. Bei dem Splay-Baum lässt sich die Korrektheit vom Splaying jedoch nicht überprüfen, da die Datenstruktur selber keine Informationen über die Zugriffe auf Elemente speichert. In `IsBT(BTree)` werden also lediglich die Voraussetzungen des Binärbaumes überprüft, somit kann die Implementation vom Binärbaum übernommen werden.

2.2.3 FindBT

Beim Finden eines Elementes wird dieses an die Wurzel des Baumes befördert. Dafür wird zunächst top-down das gesuchte Element gefunden, wobei der Pfad bottom-up hoch gegeben wird. Als Erstes wird `HERE`, danach `L` oder `R` zurückgegeben. Beim zweiten Schritt ist das gesuchte Element zwei Elemente entfernt, kann also über zwei Kanten erreicht werden. Nun wird die entsprechende Doppelrotation durchgeführt. Dies wird in die Splay Methode ausgelagert. Das gesuchte Element steht jetzt an dem Knoten, der gerade als Wurzel betrachtet wird, somit wird als Pfad wieder `HERE` zurückgegeben. Dies wird rekursiv fortgeführt, bis die Wurzel des Baumes erreicht wird. Der Vorgang ist in Abbildung 9 dargestellt.

In der Abbildung ist der Fall, dass das gesuchte Element nicht im Baum vorhanden ist, präteriert. Hierfür soll der Pfad als `NOTFOUND` zurückgegeben werden. Falls bei der Überprüfung des Pfades

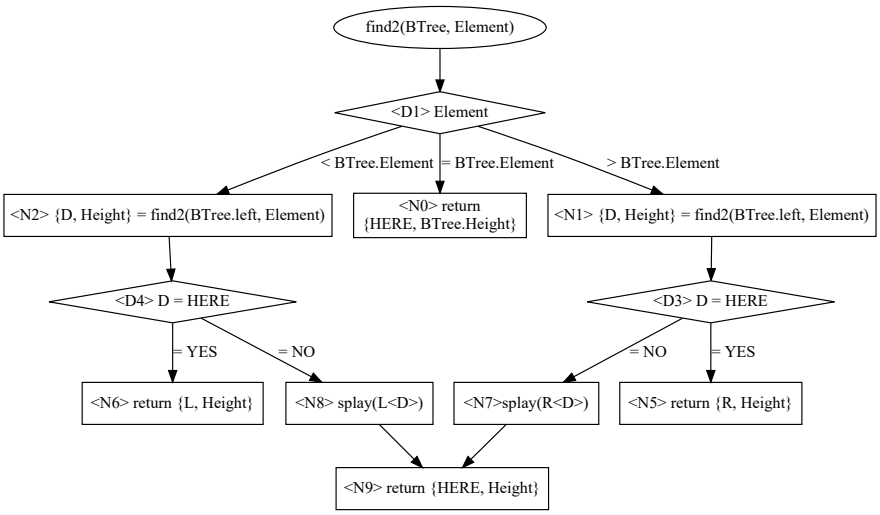


Abbildung 9: FindBT

`NOTFOUND` vorliegt, wird `HERE` zurückgegeben, was dazu führt, dass das zuletzt aufgerufene Element gesplayt wird, also zur neuen Wurzel wird `<R1>`.

Da immer zwei Nachfolger betrachtet werden, ist es möglich, dass am Ende das gesuchte Element am linken oder rechtem Kind vom Wurzelknoten steht. Um dies zu überprüfen, wird die eben beschriebene Methode in eine Wrapper-Methode umhüllt. Dort kann getestet werden, ob am Ende `HERE` zurückgegeben wurde. Falls dies der Fall ist, wird ein Zig (Einfachrotation) ausgeführt. Anschließend steht das gesuchte Element an der Wurzel. Dies ist in Abbildung 10 dargestellt. Diese

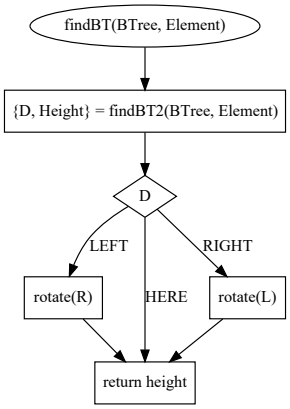


Abbildung 10: FindBT Wrapper

Operation ist auch in *FindBT*, *FindTP* und *DeleteBT* zu implementieren und wird in den Entwurf dieser Methoden nicht weiter explizit betrachtet.

2.2.4 Splay

Die Splay Methode bekommt einen Baum und einen Pfad $P \in \{LL, RR, LR, RL\}$ übergeben, welcher angibt, welcher Splaying Fall vorliegt. Es wird eine Fallunterscheidung über den Pfad durchgeführt, welche die Abfolge der Rotationen bestimmt. Die Rotationen sind wie beim AVL Baum zu implementieren.

- LL → Rechtsrotation auf Wurzel → Rechtsrotation auf neue Wurzel. `<R1>`
- RR → Linksrotation auf Wurzel → Linksrotation auf neue Wurzel. `<R2>`
- RL → Rechtsrotation auf rechtes Kind → Linksrotation auf Wurzel. `<R3>`
- LR → Linksrotation auf linkes Kind → Rechtsrotation auf Wurzel. `<R4>`

2.2.5 FindTP

Alternativ kann beim Finden eines Elementes dieses nur um einen Knoten nach oben bewegt werden. Zunächst wird das Element gefunden, anschließend eine Zig-Operation durchgeführt. Um dies zu realisieren, wird, sobald das gesuchte Element gefunden wurde, ein Flag zurückgegeben, welches signalisiert, dass rotiert werden soll `<R1>`. Wenn beim Überprüfen des Rückgabewertes das Flag gesetzt ist, wird in die entsprechende Richtung rotiert `<R2>`. Somit wird das Element nur um eine Ebene nach oben rotiert, da das Flag anschließend nicht mehr zurückgegeben wird `<R3>`. Falls ein Element nicht gefunden wird, wird `NOTFOUND` zurückgegeben `<R4>`. Beim Überprüfen des Pfades wird daraufhin `HERE` zurückgeben `<R5>`, was dazu führt, dass das zuletzt zugegriffene Elemente genau eine Ebene nach oben rotiert wird.

2.2.6 InsertBT

Beim Einfügen eines Elementes wird dieses an die Wurzel befördert. Dies kann dadurch realisiert werden, dass das Element zunächst wie bei einem Binärbaum eingefügt, anschließend mithilfe von FindBT an die Wurzel befördert wird. Diese Implementation hat den Nachteil, dass der Baum insgesamt zweimal durchlaufen wird. Zur Optimierung werden, beide Schritte in einem Durchlauf durchgeführt: Zunächst top-down das Element einfügen `<R1>`, dann, mithilfe von Rotationen, bottom-up das Element zur Wurzel bringen `<R2>`. Dabei ist letzteres wie bei FindBT zu realisieren, die Höhe muss dabei nicht zurückgegeben werden. Außerdem muss das `NOTFOUND` nicht zurückgegeben werden, da dieser Fall hier nicht auftritt. Stattdessen wird nach dem Einfügen `HERE` zurückgegeben, damit dieses Element gesplayt wird `<R3>`.

2.2.7 DeleteBT

Bei deleteBT wird ein Element als Erstes mit FindBT an die Wurzel befördert `<R1>`. Dieses wird gelöscht, nun bleiben zwei Teilbäume, übrig, welche mit JoinBT zu einem zusammengeführt werden `<R2>`. Wenn ein Element nicht gefunden wird, wird das zuletzt besuchte Element gesplayt, dies wird wie in FindBT (Abschnit 2.2.3) realisiert.

2.2.8 JoinBT

Um zwei Bäume zu einem zusammenzuführen wird zunächst das größte Element des linken Teilbaumes gesplayt `<R1>`, somit hat die neue Wurzel kein rechtes Kind, da es das größte Element des Baumes ist. Der rechte Teilbaum wird jetzt als rechtes Kind der neuen Wurzel gesetzt `<R2>`.

2.3 Aufgaben

2.3.1 Aufgabe 2.5 - Analysieren beispielhafter Bäume

Beispielhafte Bäume werden mit der zur verfügung gestellten Methode `analysesBT:test(<Anzahl Elemente>)` analysiert.

100 Elemente Ein Aufruf mit 100 Elementen ergibt folgende Ausgabe:

```
> analysesBT:test(100).
Ausgaben beziehen sich auf die Rahmenbedingungen von AVL-Bäumen.
Bei gegebener Höhe 17:
         minimale Anzahl Knoten: 4180.
         maximale Anzahl Knoten: 131071
Bei gegebener Anzahl an Knoten 100:
         minimale Höhe: 6.
         maximale Höhe: 9
Durchschnittliche Balancefehler: 2.19
         maximaler Balancefehler: 10.
```

1000 Elemente Ein Aufruf mit 1000 Elementen ergibt folgende Ausgabe:

```
> analysesBT:test(1000).
Ausgaben beziehen sich auf die Rahmenbedingungen von AVL-Bäumen.
Bei gegebener Höhe 25:
         minimale Anzahl Knoten: 196417.
         maximale Anzahl Knoten: 33554431
Bei gegebener Anzahl an Knoten 1000:
         minimale Höhe: 9.
         maximale Höhe: 14
Durchschnittliche Balancefehler: 1.97
         maximaler Balancefehler: 18.
```

An den Ausgaben ist zu erkennen, dass die Bäume im Vergleich zu AVL-Bäumen deutlich schlechter balanciert sind. Bei beiden Ausgaben ist der durchschnittliche Balancefehler ca. 2, wobei dieser bei den AVL-Bäumen nur bei ca. 0.3 lag. Außerdem überschreitet die Höhe vom Splay-Baum im ersten Beispiel die maximale Höhe eines AVL-Baumes um 100 Prozent. Hierdurch wird ersichtlich, dass ein Splay-Baum für zufällige Lese- und Schreiboperationen eine schlechtere Charakteristik aufzeigt.

2.3.2 Aufgabe 2.6 - Prüfen der Strategie anhand von Bildern

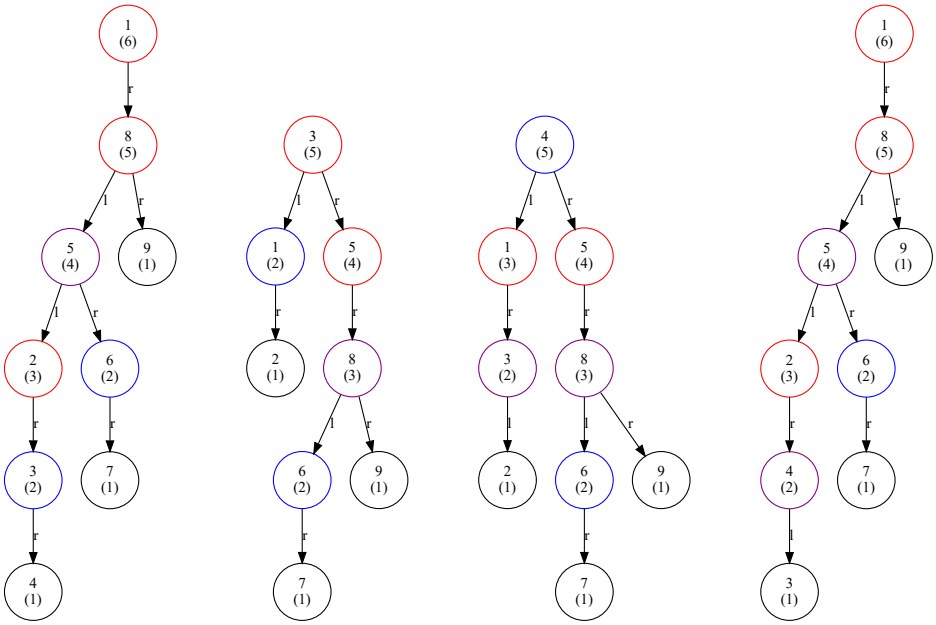
Zum Prüfen der Korrektheit der Strategie werden kleine Bäume mithilfe von printBT ausgegeben. In Abbildung 11a ist der Ausgangsbaum dargestellt. Alle folgenden Operationen werden auf diesem Baum ausgeführt.

Im Baum vorhandene Elemente In Abbildung 11b wurde die 4 gelöscht. Der Vaterknoten der 4 ist die 3 gewesen, somit wurde diese gesplayt und steht nun an der Wurzel.

In Abbildung 11c wurde die `FindBT(4)` auf den Ausgangsbaum angewandt, die 4 wurde gesplayt und steht jetzt an der Wurzel.

In Abbildung 11d wurde die `FindTP(4)` auf den Ausgangsbaum angewandt. Die 4 wurde um eine Position gesplayt.

In Abbildung 12a wurde die 10 eingefügt, diese wurde an die Wurzel gesplayt.



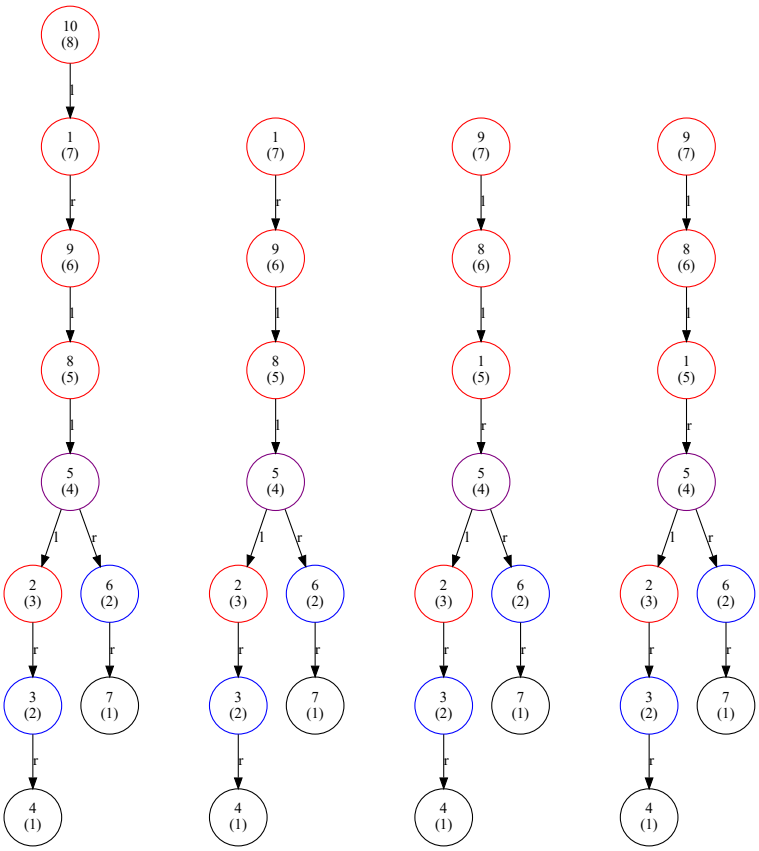
(a) Ausgangsbaum (b) Delete 4 (c) FindBT 4 (d) FindTP 4

Abbildung 11: Bilder der Splaying-Strategie - vorhandene Elemente

Im Baum nicht vorhandene Elemente In Abbildung 12d und Abbildung 12c wurde jeweils die 99 gesucht bzw gelöscht. Da die 99 nicht existiert wurde der Vaterknoten (9) gesplayt und steht anschließend an der Wurzel.

In Abbildung 12b wurde die `FindTP(99)` auf den Ausgangsbaum angewandt. Hier wurde die 9 um eine Position gesplayt.

Alle Funktionen funktionieren somit wie erwartet.



(a) Insert 10 (b) FindTP 99 (c) Delete 99 (d) FindBT 99

Abbildung 12: Bilder der Splaying-Strategie - nicht vorhandene Elemente

3 Laufzeitmessung

Für die Laufzeitmessungen wird die zur Verfügung gestellte Datei *zeitAVLBT* verwendet. *InsertBT*, *DeleteBT*, *FindBT*, *FindTP*, *isBT* und *EqualBT* werden hier jeweils für aufsteigende, absteigende und zufällige Zahlen getestet. Es wurde über 20 Messungen gemittelt, die Schrittgröße beträgt jeweils 500 Elemente.

Die Testsystemspezifikationen sehen folgt aus:

- CPU: Intel Core i7-8700K
- RAM: 32GB 3000MHz
- Windows 10 Build 19041.746
- Erlang/OTP 23.2

3.1 AVL - Baum

In Abbildung 13a ist jeweils die Laufzeit von *FindBT* und *DeleteBT* kumuliert dargestellt. Die anderen Methoden wurden hier nicht dargestellt, da deren Laufzeit jeweils nur bis zu 10ms bei 50 Tausend Elementen beträgt, und diese bei einer Messauflösung von 1ms nicht aussagekräftig sind. An den Messwerten lässt sich zunächst nicht die Komplexität des Algorithmus festellen, allerdings lassen sich Aussagen über die relative Laufzeit der jeweiligen Kombinationen festhalten.

In Abbildung 13b, 13c und 13d sind jeweils die Messungen pro Element dargestellt. An diesen lässt sich unter anderem die Komplexität des Algorithmus bestimmen.

3.1.1 Löschen vs. Einfügen

Die erste Auffälligkeit ist, dass das Löschen in allen drei Fällen schneller als das Einfügen ist. Dies Entspricht nicht den Erwartungen. Nach diesen müsste das Einfügen schneller als das Löschen sein, da beim Einfügen jeweils maximal einmal rotiert wird. Dies wird z.B. an Abbildung 1 deutlich: In beiden Fällen – dem Left-Right Case und Left-Left Case (und deren gespiegelten Varianten) – ändert sich die Höhe des Wurzelelementes nach dem Einfügen nicht, diese beträgt weiterhin 1.

- Left Right Case: Nach dem Einfügen der 4 beträgt die Höhe 1
- Left Left Case: Nach dem Einfügen der 3 beträgt die Höhe 1

Beim Löschen ist es hingegen möglich, dass bei jedem Knoten rotiert werden muss. Somit ist die erwartete Laufzeit vom Löschen besser als die vom Einfügen.

3.1.2 Zufällig vs. Auf- / Absteigend

Des Weiteren ist sowohl beim Löschen als auch beim Einfügen die Laufzeit bei aufsteigenden Zahlen ähnlich zu der bei absteigenden Zahlen. Bei zufälligen Zahlen zeigt der Algorithms jedoch eine bemerkbar schlechtere Laufzeit auf. Dies ist auch in Abbildung 13d bei FindBT zu sehen. Da der Algorithmus bei zufälligen Zahlen nicht langsamer sein sollte als bei sortieren Zahlen, ist dies wahrscheinlich auf das Generieren der Zufallszahlen zurückzuführen.

Aus Abbildung 13c lässt sich des Weiteren erkennen, dass die Laufzeit bei *isBT* und *EqualBT* bei zufälligen und sortierten Zahlen ähnlich ist. Lediglich *equalBT* zeigt bei zufälligen Zahlen eine etwas höhere Laufzeit auf, dies liegt jedoch im Rahmen des Messfehlers. Dies Entspricht den Erwartungen: Da die Bäume in allen Fällen balanciert sind, sollte ein Aufruf dieser Methoden kein unterschiedliches Verhalten aufzeigen.

3.1.3 Komplexität

In Abbildung 13b wurden für die Bestimmung der Komplexität bei *InsertBT* und *deleteBT*, jeweils mit zufälligen Zahlen, eine logarithmische Regression eingezeichnet. Daran ist zu erkennen, dass die Komplexität des Einfügens und Löschens logaritmisch ist. Zum Verbessern der Aussagekraft der Messung müsste der Bereich unter 500 Elementen genauer betrachtet werden, dies ist allerdings durch die hohe Streuung bei kleineren Elementzahlen nicht möglich gewesen.

In Abbildung 13d ist *FindBT* dargestellt. In dem Bereich von ca. 500 - 5000 Elementen streuen die Werte stark, weswegen sich eine sinnvolle Regression nicht einzeichnen lässt. Mit dem bloßem Auge lässt sich allerdings erkennen, dass die Laufzeit von *FindBT* bei zufälligen Zahlen einen logaritmischen Trend aufzeigt. Bei sortierten Zahlen lässt sich im Gegensatz nicht sagen, ob der Graph logarithmisch oder konstant ist. Letzteres lässt sich allerdings ausschließen, da aus dem Entwurf

bekannt ist, dass bei *FindBT* über den Baum gelaufen wird und somit keine konstante Komplexität vorliegen kann. Um dies zu bestätigen, müsste auch hier die Laufzeit für Elementanzahlen unter 500 durchgeführt werden.

3.2 Splay-Baum

Beim Splay-Baum werden bei der zu verfügung gestellten Datei jeweils 8 Mal 12,5 Prozent der Zahlen gesucht. In Abbildung 14a und 14b sind die Laufzeiten kumuliert, in Abbildung 14c und 14d elementweise dargestellt. Hier wurden, anders als bei der Zeitmessung vom AVL-Baum in Abschnitt 3.1, Balkendiagramme als Darstellungsart gewählt, da viele Messwerte nahezu identisch sind und somit besser zu sehen sind.

Bei den Laufzeitmessungen ist zu beachten, dass die diese bei auf- und absteigenden Zahlen nur bis zu 10500 Elementen durchgeführt wurden, da die Laufzeit dieser deutlich länger als bei zufälligen ist. Außerdem ist bei den kumulierten Messungen in 14a *FindTP* und in 14b *FindBT* auf der x-Achse abgeschnitten.

3.2.1 Zufällig vs. Auf- / Absteigend

Bei auf- und absteigenden Zahlen fällt an Abbildung 14a auf, dass *InsertBT* eine sehr schnelle Laufzeit aufweist. Diese ist konstant. Das liegt daran, dass bei sortierten Zahlen das neue Element jeweils an der Wurzel angefügt wird, und anschließend einmal rotiert wird, sodass dieses Element die neue Wurzel darstellt. Somit ist der Aufwand bei jedem Einfügen gleich hoch.

Des Weiteren ist die Laufzeit von *EqualBT* bei aufsteigenden Zahlen sehr schnell, wobei diese bei absteigenden Zahlen deutlich langsamer ist. Dies lässt sich durch die Implementation der *EqualBT*-Methode erklären: Der Baum wird zunächst in aufsteigend sortierter Reihenfolge als Liste ausgegeben, anschließend wird diese Liste auf Gleichheit überprüft. Da beim Einfügen von sortieren Elementen diese an der Wurzel eingefügt werden, wird die Reihenfolge des Baumes umgekehrt. Somit ist ein Baum nach dem Einfügen von aufsteigend sortierten Elementen absteigend sortiert und muss von der *EqualBT*-Methode umgekehrt werden. Dies ist in Erlang besonders aufwendig, da beim Anfügen an eine Liste jedes Mal bis ans Ende der Liste gelaufen werden muss. Der Baum, in den die Elemente absteigend sortiert eingefügt wurden, ist hingegen bereits aufsteigend sortiert. Der Aufwand für das Umkehren der Liste entfällt somit. In Abbildung 14d ist zu erkennen, dass die Laufzeit von *EqualBT* bei zufälligen Zahlen niedriger ist, als bei aufsteigenden, jedoch höher als bei absteigendenden Zahlen. So beträgt die Laufzeit bei 10500 Elementen bei zufälligen Zahlen beispielsweise ca. 250ms, bei aufsteigenden Zahlen ca. 1100ms.

3.2.2 FindBT vs. FindTP

In Abbildung 15a ist *FindTP* auch bei auf- und absteigenden Zahlen dargestellt. Der Graph von *FindTP* scheint bis zu ca 5500 Elementen quadratisch zu verlaufen, danach führt dieser linear fort. Es wird deutlich, dass im Fall von sortieren Zahlen nicht häufig genug rotiert, um die zugegriffenen Elemente nach oben zu rotieren. Somit ist die Komplexität von *FindTP* in diesem Fall nicht mehr logarithmisch. Falls auf ein Element, welches sich weit unten im Baum befindet, häufig zugegriffen wird, wird dieses jeweils nur eine Position gesplayt, dies ist bei einem großem Baum sehr aufwendig.

Im Falle von zufälligen Zahlen (Abbildung 14d) erweist sich *FindTP* als das schnellere der beiden Verfahren. *FindBT* ist bis zu 8 Mal langsamer, was sich dadurch erklären lässt, dass der Overhead der Rotationen zu hoch ist. Dadurch, dass die Elemente zufällig eingefügt wurden, ist der Baum am Anfang einigermaßen ausgeglichen. Die Zugriffszeit auf ein Element ist somit nicht hoch genug, dass es sich lohnen würde, dieses Element komplett nach oben zu rotieren. Dabei ist zu bedenken,

dass, durch die Funktionsweise von Erlang, Rotationen sehr teuer sind, da die Knoten jedes Mal neu erstellt werden müssen, also Speicherplatz alloziert wird. In einer Sprache, in der mit veränderlichen Variablen gearbeitet wird, könnte *FindBT* eine bessere Laufzeit aufweisen.

Zusammenfassen lässt sich sagen, dass *FindBT* eine konstantere Laufzeit bei unterschiedlichen Bedingungen aufweist, wobei *FindTP* bei zufälligen Zahlen schneller, jedoch bei sortierten Zahlen deutlich langsamer ist. Da die Komplexität von dem letzteren Verfahren bei sortierten Zahlen nicht mehr logarithmisch ist, ist *FindBT* vorzuziehen, wenn sortierte Zahlen nicht ausgeschlossen sind.

3.2.3 Komplexität

In Abbildung 14c und 14d sind jeweils logarithmische Regressionen für *DeleteBT*, *FindBT* und *FindTP* eingezeichnet. Alle Methoden zeigen ein logarithmisches Verhalten. Lediglich *FindTP* scheint bei zufälligen Zahlen besser als logarithmisch zu sein. Dies könnte jedoch auch daran liegen, dass der erste Messwert einen Ausreißer nach oben darstellen kann. *FindTP* ist bei sortierten Zahlen (Abbildung 15a), wie oben beschrieben, nicht mehr logarithmisch.

3.3 Vergleich

3.3.1 AVL vs. Binärbaum

Beim Binärbaum haben wir festgestellt, dass bei sortierten Zahlen die Komplexität des Algorithms nicht mehr logarithmisch ist. Der AVL-Baum hat dieses Problem behoben, wie beim Betrachten der Messwerte zum AVL-Baum ersichtlich wurde. In Abbildung 15c und Abbildung 15d ist außerdem zur Veranschaulichung ein direkter Vergleich von *FindBT*, *InsertBT* und *DeleteBT* dargestellt. Hier wird der AVL-Baum mit dem Binärbaum bei aufsteigenden und zufälligen Zahlen verglichen. Für den Vergleich werden die kumulierten statt der elementweisen Messwerte herangezogen, da beim Binärbaum die Messergebnisse fehlerhaft waren.

Sortierte Zahlen An den sortieren Zahlen ist zu erkennen, dass der AVL Baum bei allen Operationen dramatisch viel schneller als der normale Binärbaum ist. Dies liegt daran, dass der normale Binärbaum im Fall von sortierten Zahlen eine einfach verkette Liste wird, und somit eine lineare Komplexität aufweist. Der AVL-Baum hingegen balanciert sich beim Einfügen und Löschen, wodurch eine logarithmische Laufzeit beibehalten wird.

Zufällige Zahlen Da der normale Binärbaum mit zufälligen Zahlen bereits relativ balanciert ist, ist an den Messungen mit zufälligen Zahlen der Overhead der Rotationen zu erkennen. Bei *FindBT* verhalten sich beide Bäume ähnlich, da hier keine Rotationen beim AVL-Baum ausgeführt werden. Bei *DeleteBT* und *InsertBT* ist zu erkennen, dass der normale Binärbaum ca. 2 bis 3 Mal schneller ist.

3.3.2 AVL vs. Splay-Baum

In Abbildung 15b wird der AVL-Baum mit dem Splay-Baum bei zufälligen Zahlen verglichen. Dafür ist jeweils der Quotient zwischen dem Splay und AVL-Baum dargestellt. Beim Vergleich fällt auf, dass der AVL-Baum ca. 10 bis 1000 mal schneller als der Splay-Baum ist. Dies lässt sich an den in Erlang teuren Rotationen erklären und ist in beim Vergleichen von *FindBT* und *FindTP* bereits aufgefallen. Beim Suchen mit zufälligen Zahlen beträgt der Faktor im Vergleich mit *FindTP* ca. 100, mit *FindBT* sogar nahezu 1000. Dies lässt sich dadurch erklären, dass beim AVL-Baum lediglich lesend auf den Arbeitsspeicher zugegriffen wird, wohingegen beim Splay-Baum pro Rotation mehrere Knoten neu

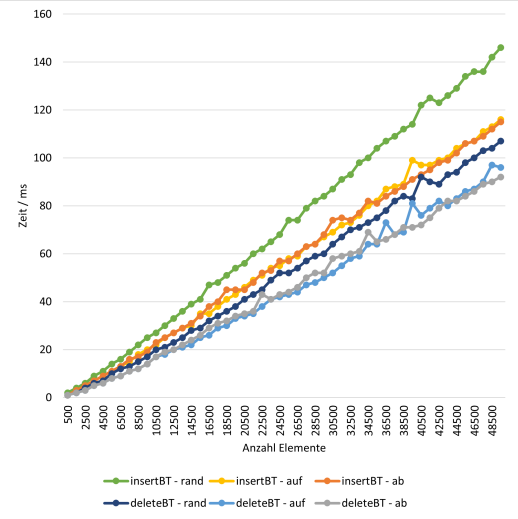
geschrieben werden müssen. Da bei *FindBT* besonders viele Rotationen ausgeführt werden, ist der Faktor dementsprechend höher.

Bei einer Implementation in Erlang ist der AVL-Baum einem Splay-Baum somit vorzuziehen.

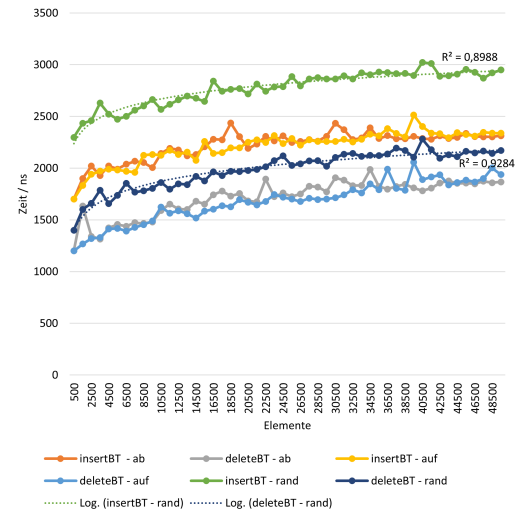
4 Fazit

Zusammenfassend lässt sich festhalten, dass der AVL-Baum die Schwächen eines normalen Binärbaums behebt, dabei jedoch die Laufzeit bei zufälligen Zahlen bis zu 3 Mal verschlechtert. Falls sortierte Zahlen nicht ausgeschlossen sind, ist der AVL-Baum trotzdem eine bessere Wahl, da die lineare Komplexität des normalen Binärbaums im Falle von sortierten Zahlen die Laufzeit dramatisch verschlechtert.

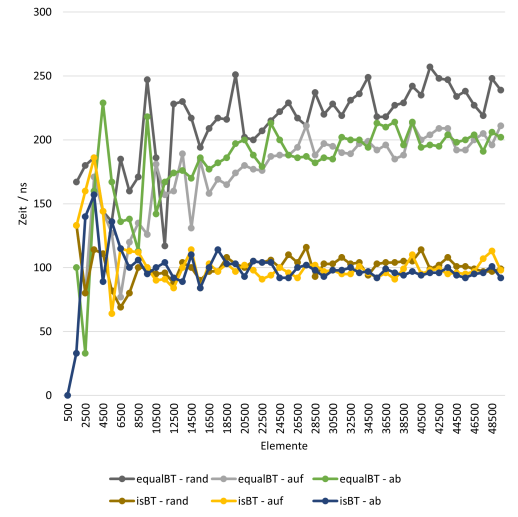
Der Splay-Baum ist hingegen, zumindest bei der Erlang Implementation, nicht zu empfehlen. Beim Zugriff auf 12,5 Prozent der Zahlen erweist sich dieser in den Laufzeittests in jedem Fall deutlich langsamer als die Alternative. Dies ist hauptsächlich der Erlang Implementation geschuldet und könnte bei einer Implementation in einer Sprache mit mutablen Variablen deutlich performanter sein.



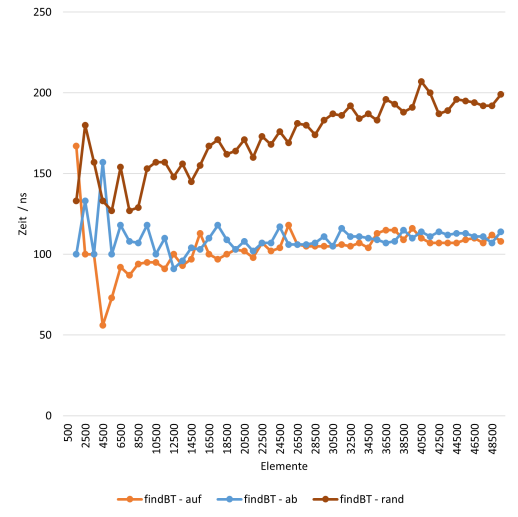
(a) Kumuliert



(b) Pro Element - DeleteBT und InsertBT

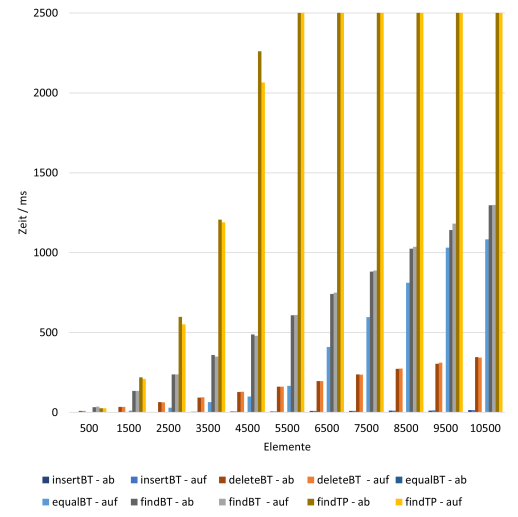


(c) Pro Element - EqualBT und IsBT

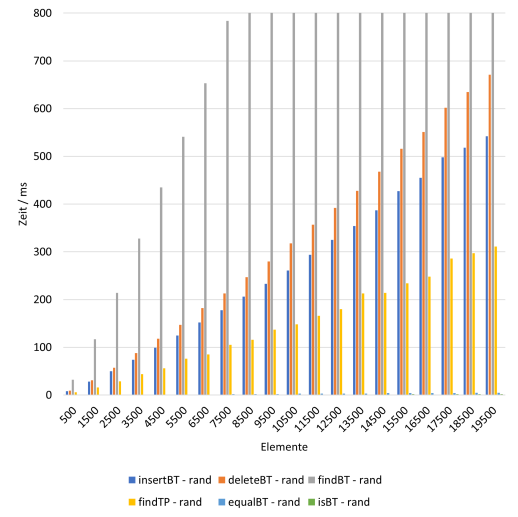


(d) Pro Element - FindBT

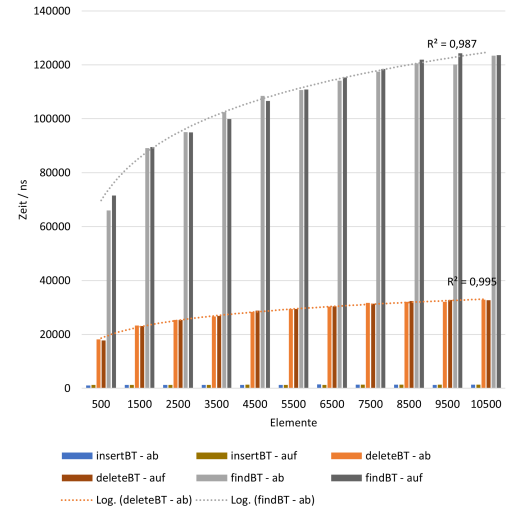
Abbildung 13: Zeitmessungen AVL-Baum



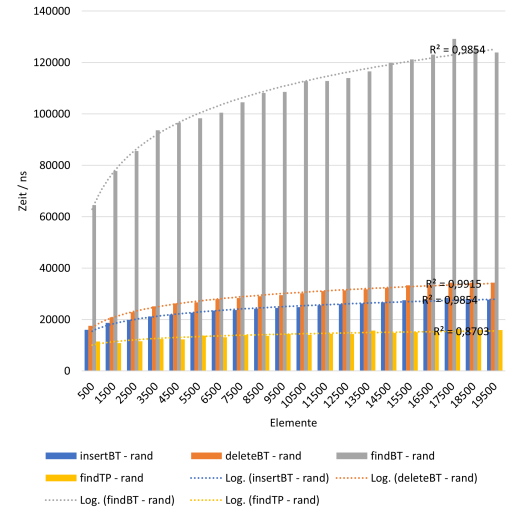
(a) Kumuliert - Auf- / Absteigend



(b) Kumuliert - Zufällig

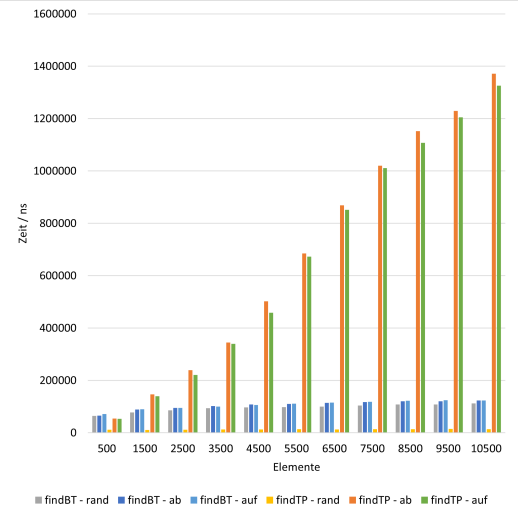


(c) Pro Element - Auf- / Absteigend

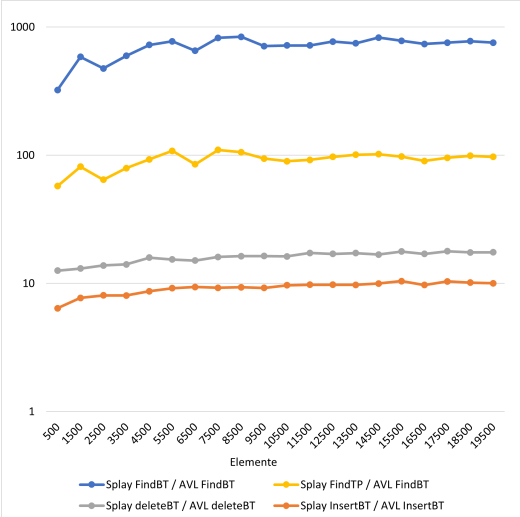


(d) Pro Element - Zufällig

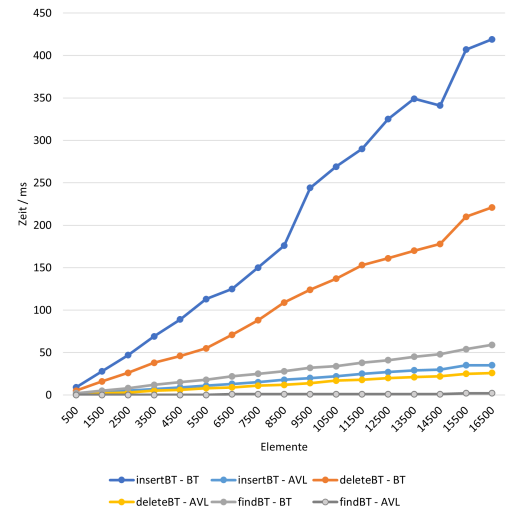
Abbildung 14: Zeitmessungen Splay-Baum



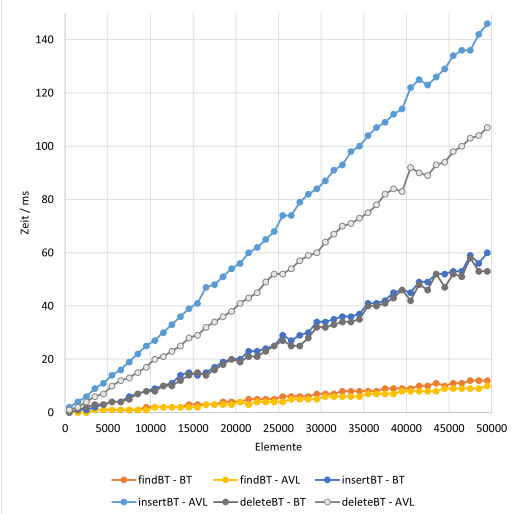
(a) Splay FindTP vs. FindBT - pro Element



(b) AVL vs. Splay - zufällig, pro Element



(c) AVL vs. BT - aufsteigend



(d) AVL vs. BT - zufällig

Abbildung 15: Vergleich

A Bildquellenverzeichnis

Abbildung 1

Basierend auf Grundlage von:
https://commons.wikimedia.org/wiki/File:AVL_Tree_Rebalancing_he.svg

Abbildung 8c

https://en.wikipedia.org/wiki/File:Splay_tree_zig.svg

Abbildung 8b

https://en.wikipedia.org/wiki/File:Zigzag.gif

Abbildung 8a

https://en.wikipedia.org/wiki/File:Zigzig.gif

B Eigenständigkeitserklärung