Bachelorarbeit

Andreas Windorfer

25. Juli 2020

Zusammenfassung

Inhaltsverzeichnis

1 Fazit

2 Einleitung

3 Binäre Suchbäume

Es gibt viele Varianten von binären Suchbäumen mit unterschiedlichen Eigenschaften und Leistungsdaten. In diesem Kapitel werden binäre Suchbäume im Allgemeinem beschrieben. Außerdem werden Begriffe definiert, die in den nachfolgenden Kapiteln verwendet werden.

3.1 Definition binärer Suchbaum

Ein **Baum** T ist ein minimal zusammenhängender, gerichteter Graph. Ein Baum ohne Knoten ist ein **leerer Baum**. In einem nicht leerem Baum gibt es genau einen Knoten ohne eingehende Kante, diesen bezeichnet man als **Wurzel**. Alle anderen Knoten haben genau eine eingehende Kante. Ein **Pfad** P_{jk} ist eine Folge von Knoten $(v_0,v_1,...,v_n)$, mit $v_0 = v_j$, $v_n = v_k$ und $\forall i \in \{1, 2, ..., n\}$: *$v_{i-1}$ ist der Elternknoten von v_i*. n ist die **Länge des Pfades**. Die Knoten v_0 bis v_{n-1} sind **Vorfahren** von v_n.Jeder Knoten v in T ist Wurzel eines **Teilbaumes** $T(v)$, der entsteht in dem alle Knoten u aus T entfernt werden, zu denen es keinen Pfad P_{vu} gibt. Knoten ohne ausgehende Kante nennt man **Blatt**, alle anderen Knoten werden als **innere Knoten** bezeichnet. Enthält der Baum eine Kante von Knoten v_1 zu Knoten v_2 so nennt man v_2 ein **Kind** von v_1 und v_1 bezeichnet man als den **Elternknoten** von v_2. Die Wurzel hat also keinen Elternknoten, alle anderen Knoten genau einen.
Bei einem **binärem Baum** kommt folgende Einschränkung hinzu:
Ein Knoten hat maximal zwei Kinder.
Entsprechend ihrer Zeichnung benennt man die Kinder in Binärbäumen als **linkes Kind** oder **rechtes Kind**. Sei w das linke bzw. rechte Kind von v, dann bezeichnet man den Teilbaum mit Wurzel w als **linken Teilbaum** bzw. **rechten Teilbaum** von v.
Bei einem **binären Suchbaum** ist jedem Knoten v ein innerhalb der Baumstruktur ein eindeutiger **Schlüssel** *key* (v) aus einem **Universum** zugeordnet werden, auf dem eine totale Ordnung definiert ist. Auf totale Ordnungen wird in diesen Kapitel noch eingegangen. Wenn nicht explizit anders angegeben wird hier und in den folgenden Kapiteln wird als Universum immer $\mathbb{N}$ verwendet, wobei die 0 enthalten ist. Die in einem binärem Suchbaum enthaltenen Schlüssel bezeichnen wir als seine **Schlüsselmenge**. Damit aus dem binären Baum ein binärer Suchbaum wird, benötigt man noch folgende Eigenschaft:
Für jeden Knoten im binären Suchbaum gilt, dass alle in seinem linken Teilbaum enthaltenen Schlüssel kleiner sind als der eigene Schlüssel. Alle im rechten Teilbaum enthaltenen Schlüssel sind größer als der eigene Schlüssel.

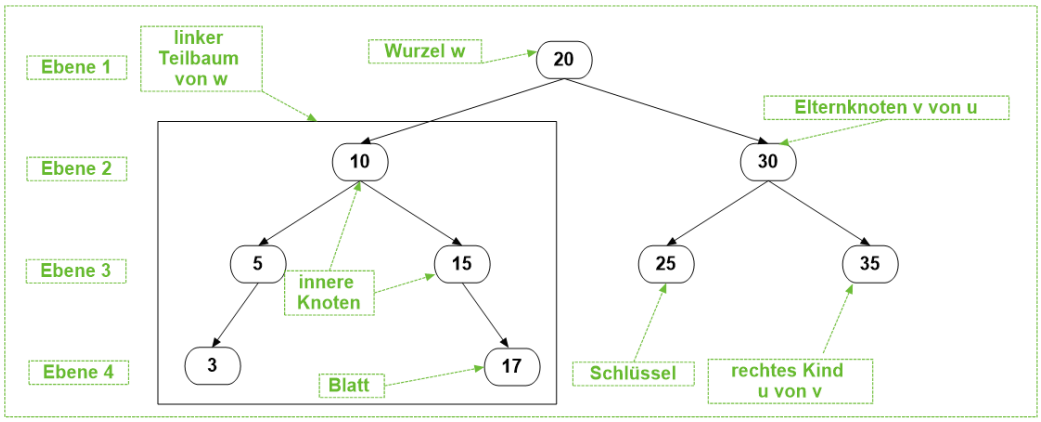

Abbildung 1: Ein binärer Suchbaum

Anstatt binärer Suchbaum schreibt man häufig **BST** für Binary Search Tree. Diese Abkürzung wird hier ab jetzt auch verwendet. In Implementierungen enthält jeder Knoten für das linke und rechte Kind jeweils einen Zeiger. Anstatt von entfernten oder hinzugefügten Kanten wird im folgenden häufig von umgesetzten Zeigern gesprochen.

3.2 Weitere Begriffe und Eigenschaften zum binären Suchbaum

Zwei verschiedene Knoten mit dem selben Elternknoten nennt man **Geschwister**. Den Knoten in einem BST wird auch eine **Tiefe** und eine **Höhe** zugeteilt. Für einen Knoten v gilt, dass die Länge des Pfades von der Wurzel zu ihm seiner Tiefe entspricht. Sei l die maximale Länge eines von v aus startenden Pfades. Die Höhe $h(v)$ von v ist dann $l+1$. Die Höhe der Wurzel entspricht der **Höhe des Baumes** $h(T)$, wobei ein leerer Baum Höhe 0 hat. Einen BST T mit Höhe h_T unterteilt man von oben nach unten in die **Ebenen** $1, 2..., h_T$. Die Wurzel liegt in der Ebene eins, deren Kinder in der Ebene zwei usw. Enthält eine Ebene ihre maximale Anzahl an Knoten nennt man sie **vollständig besetzt**.
Da im linken Teilbaum nur kleinere Schlüssel vorhanden sein dürfen und im rechten Teilbaum nur größere, kann man die Schlüsselmenge eines binären Suchbaumes, von links nach rechts, in aufsteigend sortierter Form ablesen. Denn angenommen es gibt zwei Knoten v_l, v_r mit den Schlüsseln k_l bzw. k_r, so dass $k_l > k_r$ gilt und v_l liegt weiter links im Baum als v_r. Ist v_l ein Vorfahre von v_r, so enthält der rechte Teilbaum von v_l einen Schlüssel der kleiner ist als k_l. Ist v_r Vorfahre von v_l, so enthält der linke Teilbaum von v_r einen Schlüssel

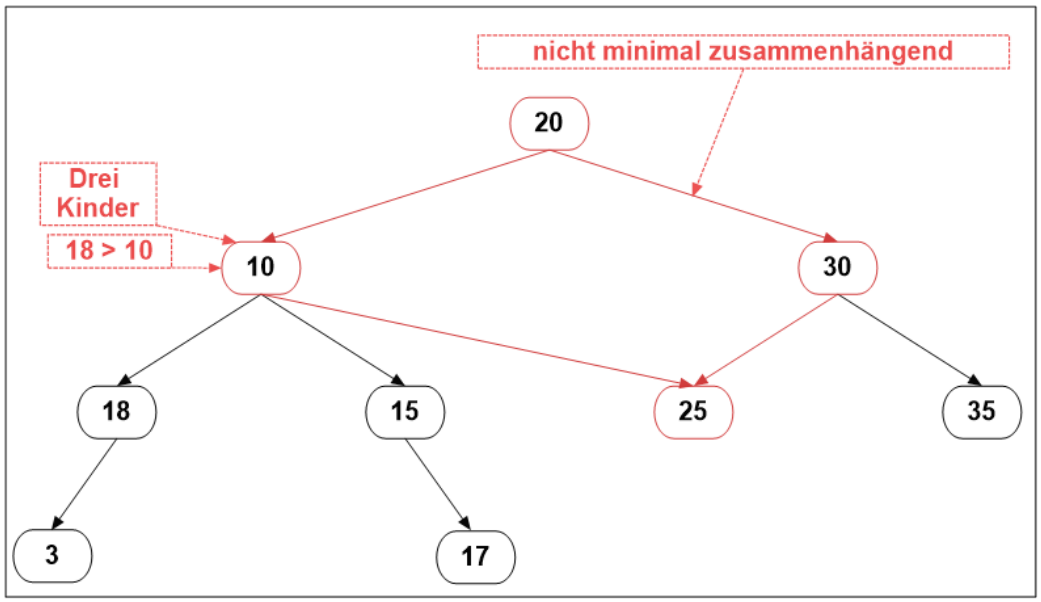

Abbildung 2: Kein binärer Suchbaum

der größer ist als k_r. Ist keiner der Knoten Vorfahre des anderen, muss es zumindest einen gemeinsamen Vorfahren geben, denn dann kann weder v_r noch v_l die Wurzel des BST sein. Sei v_v der gemeinsame Vorfahre mit der größten Tiefe. Der linke Teilbaum von v_v enthält dann einen größeren Schlüssel, als der rechte Teilbaum dieses Knotens. In jedem Fall erhält man einen Widerspruch zu der von BSTs geforderten Eigenschaft. Aus Platzgründen passiert es bei Zeichnungen von BSTs manchmal, dass ein Knoten in einem linken Teilbaum weiter rechts steht als die Wurzel des Teilbaumes, oder umgekehrt, weshalb man bei der Betrachtung solcher Zeichnungen etwas vorsichtig sein muss. Abbildung 4 enthält keine solche Konstellation.
Algorithmisch kann man sich die im BST enthaltenen Schlüssel aufsteigend sortiert durch eine **Inorder-Traversierung** ausgeben lassen. Es ist ein rekursives Verfahren, dass an der Wurzel startet und pro Aufruf drei Schritte ausführt.

Algorithmus *inorder* (**Knoten** v)

1. Existiert ein linkes Kind vl von v, rufe $inorder(vl)$ auf.
2. Gib den Schlüssel von v aus.
3. Existiert ein rechtes Kind vr von v, rufe $inorder(vr)$ auf.

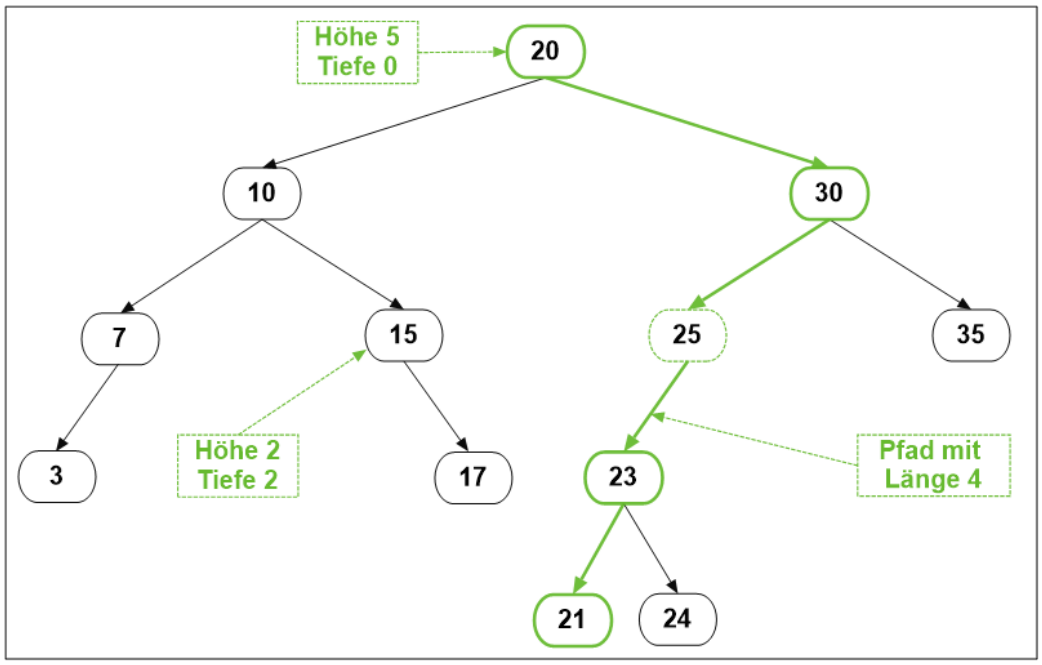

Abbildung 3: Ein weiterer binärer Suchbaum

Dass das Verfahren funktioniert sieht man leicht, durch Induktion über die Anzahl der Knoten n. Für $n = 1$ funktioniert es, da der einzige im BST enthaltene Schlüssel ausgegeben wird. Wir nehmen nun an, dass die Ausgabe für BSTs mit Knotenzahl $\leq n$ korrekt ist. Sei T_1 ein BST mit Knotenanzahl $n+1$ und Wurzel w. Sowohl für den linken als auch für den rechten Teilbaum von w gilt, dass die Anzahl enthaltener Knoten $\leq n$ ist. Als erstes wird der linke Teilbaum von w korrekt ausgegeben, dann der Schlüssel von w selbst und zuletzt der rechte Teilbaum von w. Damit wurde auch für den Gesamtbaum die richtige Ausgabe erzeugt. Als **Vorgänger** eines Knoten v, mit Schlüssel k_v bezeichnet man den Knoten mit dem größten im BST enthaltenem Schlüssel k für den gilt $k < k_v$. Aus der Inorder-Traversierung kann man eine Anleitung zum Finden des Vorgängers ableiten. Wenn ein linker Teilbaum vorhanden ist, wird der größte Schlüssel in diesem, also der am weitesten rechts liegende, direkt vor k ausgegeben. Anderenfalls wird der Schlüssel des tiefsten Knotens, auf dem Pfad von der Wurzel zu v ausgegeben, bei dem v im rechten Teilbaum liegt. Als **Nachfolger** von v, bezeichnet man den Knoten mit dem kleinsten im BST enthaltenem Schlüssel k für den gilt $k > k_v$. Da dieser Schlüssel bei der Inorder-Traversierung direkt nach v ausgegeben wird, findet man den zugehörigen Knoten ganz links im rechten Teilbaum von v, falls ein solcher vorhanden ist. Ansonsten ist es der

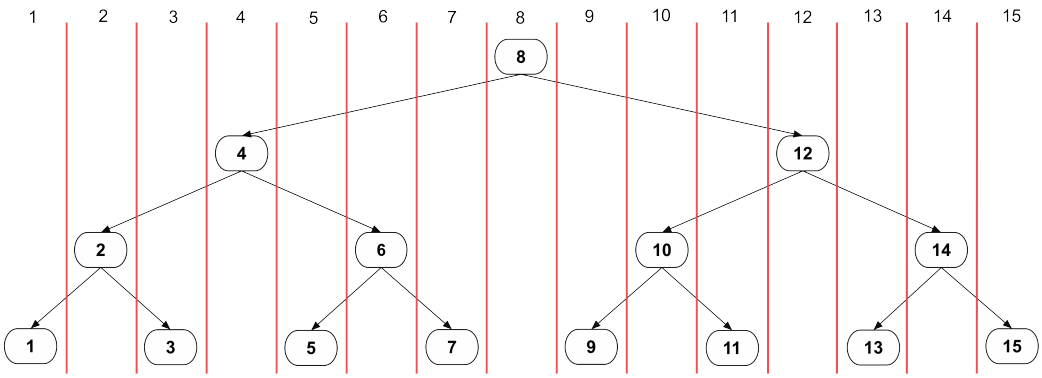

Abbildung 4: Schlüssel sind aufsteigend sortiert ablesbar.

tiefste Knoten, auf dem Pfad von der Wurzel zu v, bei dem v im linkem Teilbaum liegt. Abbildung 5 zeigt Vorgänger und Nachfolger eines Knotens. Als **Vorfahre** eines Knotens v bezeichnet man alle Knoten auf dem Pfad von der Wurzel zu v, inklusive v selbst.

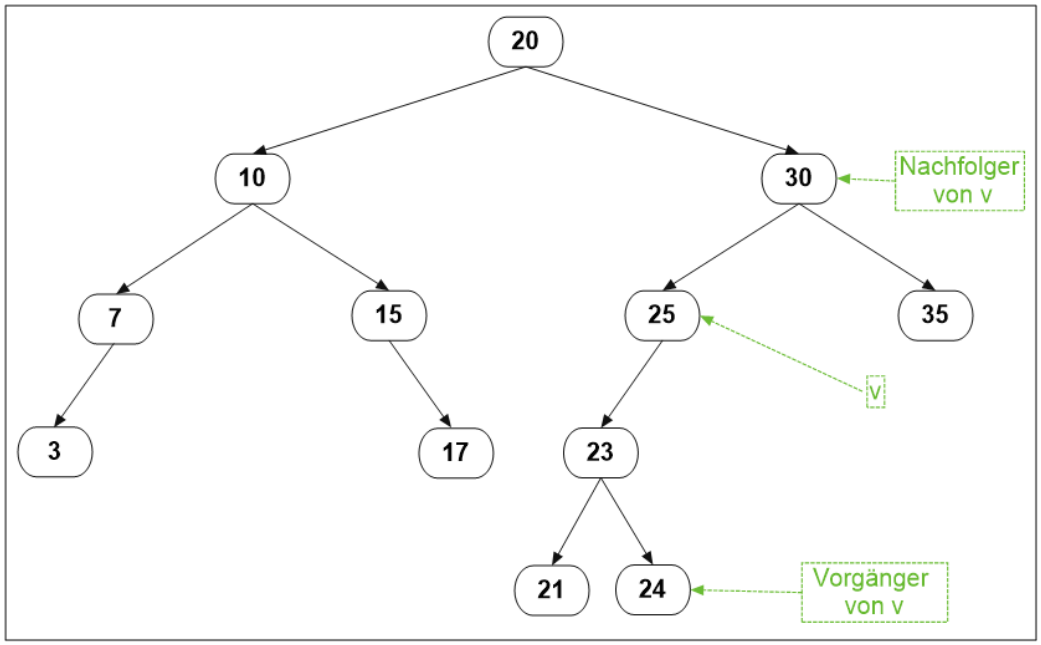

Abbildung 5: Darstellung von Vorgänger und Nachfolger.

Total geordnete Menge Eine Menge M wird als **total geordnet** bezeichnet wenn auf ihr eine zweistellige Relation $\leq$ definiert ist, die folgende Eigenschaften erfüllt.

Für alle a,b,$c \in M$ gilt:

1. $(a, a) \in R$ (reflexiv)
2. $(a, b) \in R \wedge (a, b) \in R \Rightarrow a = b$ (antisymmetrisch)
3. $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$ (transitiv)
4. $(a, b) \notin R \Rightarrow (b, a) \in R$ (total)

Die Eigenschaften 1,2 und 4 werden benötigt um für zwei beliebige Elemente aus der Menge feststellen zu können ob sie gleich sind, oder bei Ungleichheit, welches Element weiter Links bzw. Rechts im BST liegen muss. Dafür wird z.B getestet ob die Elemente (a, b) und (b, a) in der Relation liegen. Eigenschaft 3 ist notwendig, denn liegt b weiter rechts im BST als a und c liegt weiter rechts als b, dann liegt c natürlich auch weiter rechts als a.
Die von uns verwendete „Kleiner-Gleich-Beziehung“ auf den natürlichen Zahlen erfüllt alle Eigenschaften.

Verändern eines BST durch Rotationen. Wird ein BST durch eine Veränderung in einen anderen BST überführt, kann es passieren dass sich die Eigenschaften eines Knoten ändern. Um nicht immer erwähnen zu müssen auf welchen BST sich eine Aussage bezieht, wird es ab jetzt durchgängig so sein, dass sich ein Variablenname ohne angefügten Hochstrich auf den BST vor der Änderung bezieht. Der gleiche Variablenname mit angefügtem Apostroph bezieht sich dann auf den selben Knoten nach der Änderung. Z.B. bezieht sich x auf den Knoten mit Schlüssel k, in der Ausgangssituation, dann bezieht sich x' auf den Knoten mit Schlüssel k nach dem Ausführen der Änderung.

Rotationen können verwendet werden um lokale Änderungen an der Struktur eines BST durchzuführen, ohne eine der geforderten Eigenschaften zu verletzen. Es wird zwischen der Linksrotation und der Rechtsrotation unterschieden. Hier wird zunächst auf die in Abbildung 6 dargestellte Linksrotation eingegangen. Sei x der Knoten auf dem eine Linksrotation durchgeführt wird. Sei z der Elternknoten von x. z muss existieren, ansonsten darf auf x keine Rotation durchgeführt werden. Sei B der linke Teilbaum von x. Nach der Rotation ist x' linkes bzw. rechtes Kind von dem Knoten, an dem z linkes bzw. rechtes Kind war. z' ist linkes Kind von x'. B' ist rechtes Kind von z'. Unabhängig von der Anzahl der im BST enthaltenen Knoten und der

Ausführungsstelle im BST ist eine Linksrotation also mit dem Aufwand verbunden drei Zeiger umzusetzen. Zu beachten ist, dass die Höhen von x' und der Knoten in dessen, ansonsten unverändertem, rechtem Teilbaum jeweils um eins größer sind als die von x und den Knoten in dessen rechtem Teilbaum. Die Höhe der Knoten im Teilbaum mit Wurzel z' sind jeweils um eins kleiner als vor der Rotation. Abbildung 7 zeigt die symmetrische Rechtsrotation. Man muss im obigen Beschreibung lediglich links durch rechts ersetzen und umgekehrt. Dass es durch eine Rotation zu keiner Verletzung der BST Eigenschaften kommt, sieht man den Abbildungen direkt an. In Abbildung 8 erkennt man, dass sich die Wirkung einer Rotation auf x durch eine gegenläufige Rotation auf z' aufheben lässt.

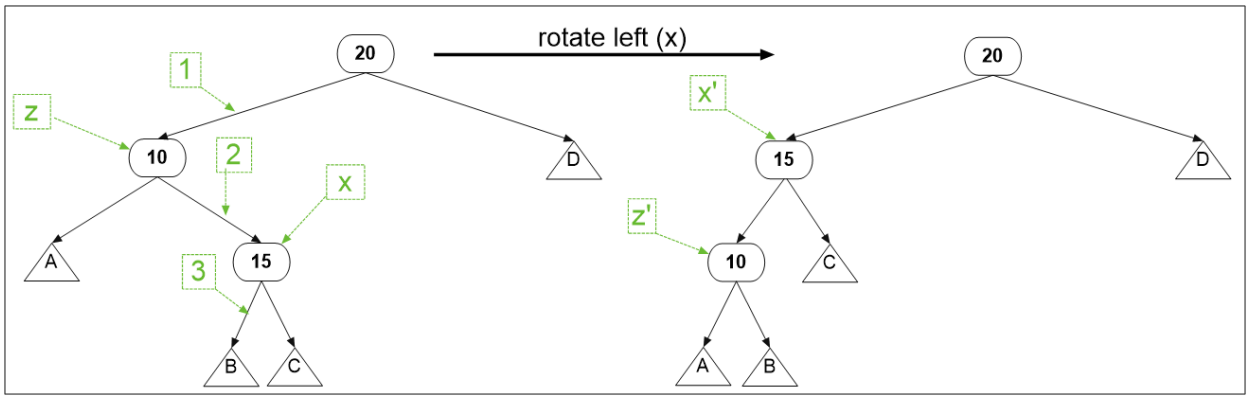

Abbildung 6: Linksrotation auf Knoten x.

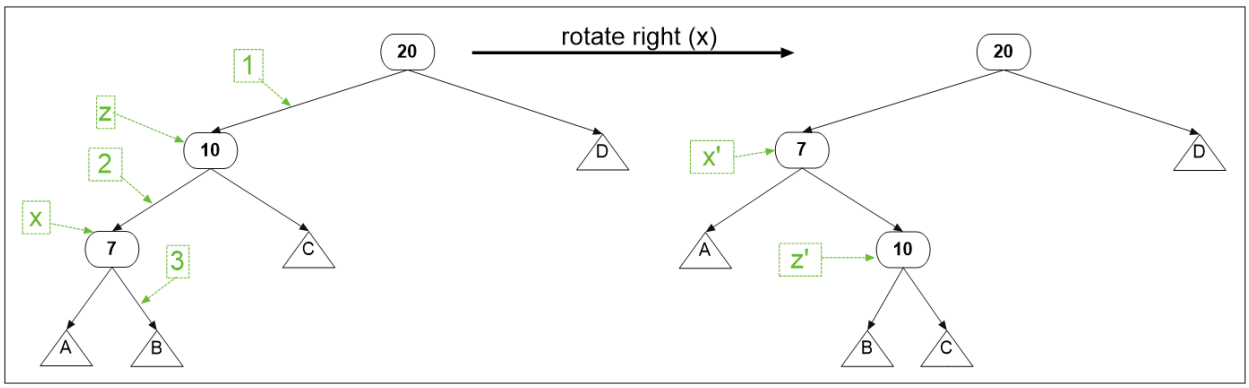

Abbildung 7: Rechtsrotation auf Knoten x.

Grundoperationen *search*, *insert* und *delete* Hier geht es nur um die Standardvarianten eines BST. Später werden Varianten gezeigt die von

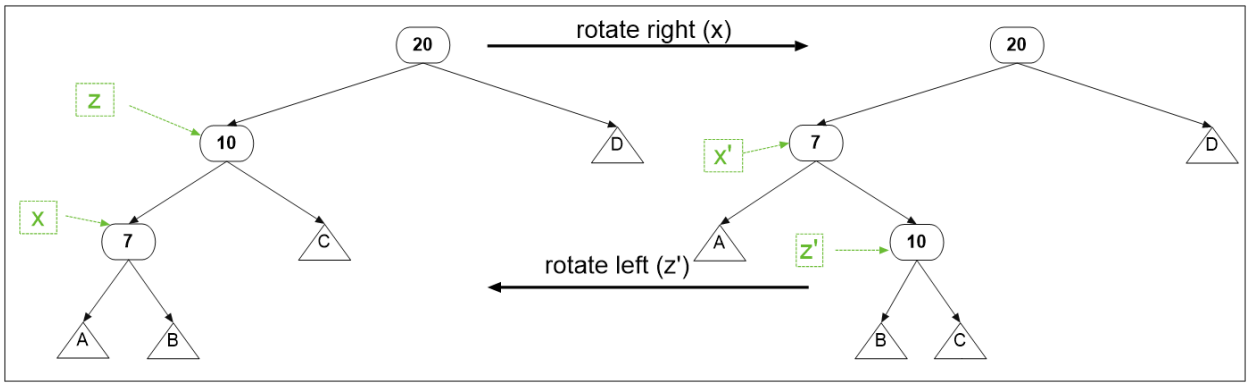

Abbildung 8: Gegenseitiges aufheben von Rotationen

diesem Verhalten zum Teil deutlich abweichen. Innerhalb Operationen wird häufig von einem Knoten aus direkt auf dessen Elternknoten zugegriffen, so dass man sich im Baum auch nach oben hin bewegen kann. In Implementierungen wird das so umgesetzt, dass es zusätzlich zu den beiden Zeigern auf die Kinder noch einen zum Elternknoten gibt.Es sei ein BST T gegeben. Innerhalb eines Pfades werden hier jedoch entweder nur Zeiger auf Kinder oder nur auf Elternknoten verwendet.
Die Operation *search*(Key k) gibt eine Referenz auf den Knoten im BST zurück, dessen Schlüssel mit k übereinstimmt. Die Operation startet an der Wurzel und vergleicht den darin enthaltenen Schlüssel mit dem Gesuchten. Ist der gesuchte Schlüssel kleiner, muss er sich im linken Teilbaum des betrachtetem Knoten befinden und die Suche wird bei dessen Wurzel fortgesetzt. Ist der Schlüssel größer, muss er sich im rechten Teilbaum befinden und die Suche wird bei dessen Wurzel fortgesetzt. Dieses Verhalten iteriert solange bis der gesuchte Schlüssel gefunden ist, oder der Teilbaum bei dem die Suche fortgesetzt werden müsste, leer ist. Ist das Letztere der Fall, ist der gesuchte Schlüssel im Baum nicht vorhanden und es wird eine leere Referenz zurückgegeben. In keinem Fall kommt es zu einer Veränderung des BST.
Bei *insert(Key k)* wird zunächst wie bei *search*(Key k) verhalten. Wird k gefunden, wird das Einfügen abgebrochen und der BST bleibt unverändert. Wird ein leerer Teilbaum T_2 erreicht, wird ein neu erzeugter Knoten mit Schlüssel k an der Position von T_2 eingefügt. Durch den neuen Knoten wird keine BST Eigenschaft verletzt. Durch ersetzen eines leeren Teilbaumes, durch einen Knoten bleibt es bei einem binären Baum. Das Verhalten von *insert* stellt sicher, dass k nur in linken Teilbäumen von Knoten mit Schlüssel $> k$ bzw. in rechten Teilbäumen von Knoten mit Schlüssel $< k$ liegt.
Auch bei *löschen(Schlüssel k)* wird sich zunächst wie beim *search*(Key k) verhalten. Ist k im BST nicht vorhanden wird abgebrochen und der BST

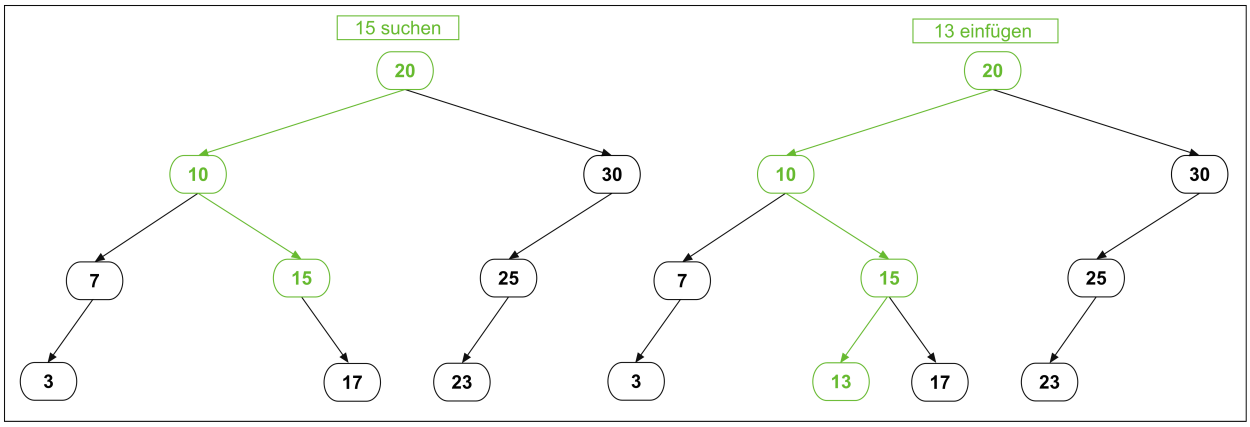

Abbildung 9: Links zeigt eine Suche nach dem Schlüssel 15. Rechts das Einfügen des Schlüssels 13

bleibt unverändert. Ansonsten werden drei Fälle unterschieden. Sei v der Knoten mit Schlüssel k.

1. v ist ein Blatt:
   v kann ohne weiteres aus dem BST entfernt werden.

2. v hat genau ein Kind c:
   Ist v die Wurzel kann er entfernt werden und c wird zur neuen Wurzel. Ansonsten ist v entweder ein linkes oder ein rechtes Kind eines Knoten w. c nimmt nun den Platz von v im BST ein. Das bedeutet, dass die Kanten von w nach v und von v nach c entfernt werden. Außerdem wird eine Kante von w nach c so eingefügt, dass c wie zuvor v das linke bzw. rechte Kind von w wird.

3. v hat zwei Kinder:
   Sei T_l der linke Teilbaum von v und T_r der Rechte. Sei z der Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von v. Als Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von v, kann z kein linkes Kind haben. Ist z ein Blatt wird seine eingehende Kante entfernt. Hat z ein rechtes Kind z_r, so nimmt dieses, analog zur Beschreibung im Fall 2, den Platz von z ein. In beiden Fällen ist z nun ein Knoten ohne Kante. Im nächsten Schritt nimmt nun z den Platz von v ein, T_l wird links an z angefügt und T_r rechts. War v zu Beginn die Wurzel, so wird z' zur neuen Wurzel.
   In keinen Teilbäumen eines Knotens außer denen von z kommen Schlüssel hinzu. Um eventuelle Verletzungen von Eigenschaften festzustellen, kann sich also auf z' beschränkt werden. Der linke Teilbaum von z'

war der linke Teilbaum von v und der Schlüssel von v ist kleiner als der von z. Der rechte Teilbaum von z enthält die Schlüssel des rechten Teilbaumes von v mit Ausnahme des Schlüssels von z selbst. z wurde gerade ausgewählt weil sein Schlüssel der Kleinste in diesem Teilbaum ist.

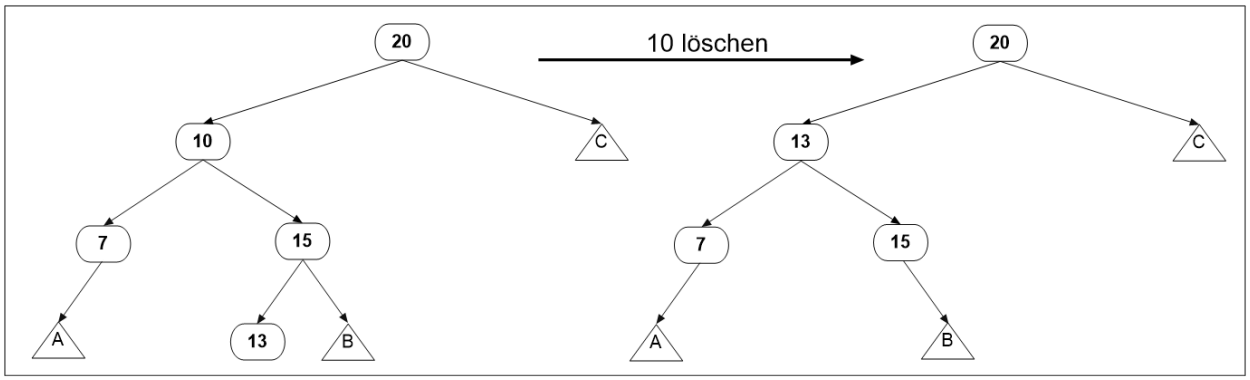

Abbildung 10: Löschen des Schlüssels 10

Laufzeit Die worst-case-Laufzeit der drei Operationen ist jeweils $O(h)$, wobei h die Höhe von T ist. Bei *search* werden maximal h Knoten aus T betrachtet. Beim Einfügen überlagern die Kosten der Suche, die konstanten Kosten für das Anhängen des neuen Knotens. Bei *delete* wird in Fall eins und zwei nach dem Suchen ebenfalls nur noch lokal beim gesuchten Knoten gearbeitet. Bei *delete* mit Fall drei muss zunächst zum Knoten z erreicht werden, dafür sind maximal h Schritte notwendig. Danach muss v erreicht werden, wozu ebenfalls maximal h Schritte notwendig sind. Die Kosten für das Entfernen und Hinzufügen von Kanten sind an beiden Stellen konstant.

Unterschiedliche Baumhöhen Da die Höhe h eines BST T mit n Knoten entscheidend für die Laufzeit der vorgestellten Operationen ist, wird hier auf diese eingegangen. Die maximale Höhe n erreicht ein BST wenn es ein Blatt im BST gibt und jeder andere Knoten ein Halbblatt ist. Die Baumstruktur geht in diesem Fall über zu einer Listenstruktur über, dies wird als **entarten** bezeichnet. Minimal wird h wenn T **vollständig balanciert** ist. Das ist der Fall wenn alle Ebenen über der Untersten vollständig besetzt sind. Sind zusätzlich in der untersten Ebene, links von jedem Knoten, alle Knoten enthalten, wird der BST als **komplett** bezeichnet, siehe Abbildung 11.

Lemma 3.1. *Die Höhe eines vollständig balancierten BST T mit n Knoten ist* $\lfloor \log_2(n) \rfloor + 1$.

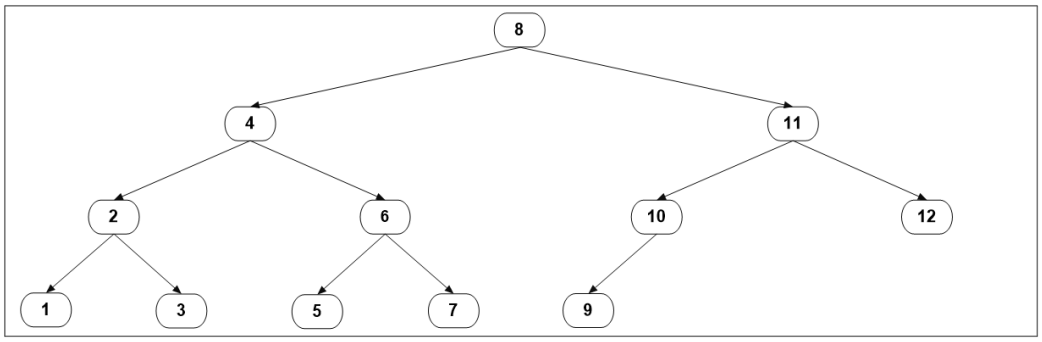

Abbildung 11: Kompletter BST mit 12 Knoten

Beweis. Es sei $N(h)$ die maximale Anzahl an Knoten in einem vollständig balancierten BST mit Höhe h. $N(h)$ berechnet sich indem die maximale Anzahl an Knoten jeder Ebene aufaddiert wird.

$$N(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

h ist minimal wenn gilt:

$$\begin{aligned} &N(h-1) < n \leq N(h) \\ \Leftrightarrow\ &N(h-1) + 1 \leq n < N(h) + 1 \end{aligned}$$

Einsetzen:

$$\begin{aligned} &2^{h-1} \leq n < 2^h \\ \Rightarrow &h = \lfloor \log_2(n) \rfloor + 1 \end{aligned}$$

□

4 Dynamische Optimalität

Dieses Kapitel beschäftigt sich vor allem mit der Laufzeit von Folgen von *access* Operationen, eine speziellere Form der *search* Operation.

4.1 BST Zugriffsfolgen

Sei T ein BST mit der Schlüsselmenge K. Beschränkt man den Parameter von *search* auf $k \in K$, wird die Operation als *access* bezeichnet. In diesem Kapitel werden Folgen solcher *access* Operationen auf einem BST mit unveränderlicher Schlüsselmenge betrachtet. Notiert wird eine solche **Zugriffsfolge** durch Angabe der Parameter. Bei der Zugriffsfolge $x_1, x_2, ...x_m$ wird also zunächst *access(x_1)* ausgeführt, dann *access(x_2)* usw. Bei BST wird bezüglich Zugriffssequenzen zwischen online und offline Varianten unterschieden. Bei **offline BST** ist die Zugriffsfolge zu Beginn bereits bekannt, somit kann ein Startzustand gewählt werden, der die Kosten minimiert. Beim **online BST** ist die Zugriffsfolge zu Beginn nicht bekannt. Bei einer worst case Laufzeit-Analyse muss somit von dem Startzustand ausgegangen werden bei dem die Kosten am höchsten sind. In dieser Arbeit werden *access* Operation betrachtet die folgende Eigenschaften einhalten:

1. Die Operation verfügt über genau einen Zeiger p in den BST. Dieser wird zu Beginn so initialisiert, dass er auf die Wurzel zeigt. Terminiert der Algorithmus muss p auf den Knoten mit Schlüssel k zeigen.

2. Der Algorithmus führt eine Folge dieser Einzelschritte durch:

   - Setze p auf das linke Kind von p.
   - Setze p auf das rechte Kind von p.
   - Setze p auf den Elternknoten von p.
   - Führe eine Rotation auf p aus.

Zur Auswahl des nächsten Einzelschrittes können zusätzliche in p gespeicherte Hilfsdaten verwendet werden. Es darf nur auf die Daten des Knotens zugegriffen (lesend oder schreibend) werden, auf den p zeigt. Es wird $n = |K|$ gesetzt. Außerdem werden hier pro Knoten als Hilfsdaten nur konstant viele Konstanten und Variablen zugelassen, die jeweils eine Größenordnung von $\log(n)$ haben dürfen.
Die Initialisierung und die Ausführung jedes Einzelschrittes aus Punkt 2 kann in konstanter Zeit durchgeführt werden. Es werden jeweils Einheitskosten von

1 verwendet. Höhere angenommene Kosten würden die Gesamtkosten lediglich um einen konstanten Faktor erhöhen. Es sei a die Anzahl der insgesamt durchgeführten Einzelschritte während einer Zugriffsfolge X mit Länge m. Dann berechnen sich die Gesamtkosten zum Ausführen von X mit $a+m$. Es muss zu jeder Schlüsselmenge und jeder Zugriffsfolge zumindest einen offline BST geben, so dass die Gesamtkosten keines anderen niedriger sind. Diese Kosten werden als $\mathbf{OPT}(\mathbf{X})$ bezeichnet.
In [1] wurde gezeigt, dass der Zustand eines BST mit maximal $2n-2$ Rotationen in jeden anderen BST mit der gleichen Schlüsselmenge überführt werden kann. Da bei der Berechnung der Kosten für $OPT(X)$, m ebenfalls als Summand vorkommt, können die zusätzlichen Kosten der online Varianten, für $m > n$ asymptotisch betrachtet vernachlässigt werden.
Als **dynamisch optimal** wird ein BST bezeichnet wenn er eine beliebige Zugriffssequenz X in $O(OPT(X))$ Zeit ausführen kann. Ein BST der jede Zugriffssequenz in $O(c \cdot OPT(X))$ Zeit ausführt, wird als **c-competitive** bezeichnet. Es konnte bis heute für keinen BST bewiesen werden, dass er dynamisch optimal ist. Es wurden aber mehrere untere Schranken für $OPT(X)$ gefunden. Eine davon wird nun vorgestellt.

4.2 Erste untere Schranke von Wilber

Robert Wilber hat in [2] zwei Methoden zur Berechnung unterer Schranken für die Laufzeit von Zugriffsfolgen bei BST vorgestellt. Hier wird auf die Erste davon eingegangen. Im folgenden werden offline BST betrachtet, bei denen während einer *access(k)* Operation, der Knoten mit Schlüssel k, durch Rotationen zur Wurzel des BST gemacht wird. Ein solcher BST wird als **standad offline BST** bezeichnet. Asymptotisch betrachtet entsteht hierdurch kein Verlust der Allgemeinheit. Sei v_p der Knoten auf den p zum Zeitpunkt t direkt vor der Terminierung von *access* zeigt. Sei d die Tiefe von v_p . Dann sind mindestens Kosten $d+1$ entstanden. Mit d Rotationen kann v_p zur Wurzel gemacht werden und mit d weiteren Rotationen kann der Zustand zum Zeitpunkt t wieder hergestellt werden. Für einen BST T mit Schlüsselmenge K_T und einer Zugriffsfolge X notieren wir die minimalen Kosten eines wie eben vorgestellt arbeitenden BST mit $W(X,T)$. Im folgenden wird angenommen, dass
$K = \{i \in \mathbb{N} | i \in [j,k] \ mit \ j,k \in \mathbb{N}\}$ gilt. Dadurch entsteht kein Verlust der Allgemeinheit, denn anderenfalls könnte man die Schlüsselmenge einfach aufsteigend sortiert mit j startend durchnummerieren. Eine Rotation wird innerhalb dieses Kapitels mit (i,j) notiert. i ist dabei der Schlüssel des Knotens v auf dem die Rotation ausgeführt wird. j ist der Schlüssel des Elternknoten von v, vor Ausführung der Rotation. Aus einer Fol-

ge von Rotationen $r = (i_1, j_1), (i_2, j_2), .., (i_m, j_m)$ erhält man die Folge $r_x^y = (i_{1'}, j_{1'}), (i_{2'}, j_{2'}), .., (i_{m'}, j_{m'})$ in dem man aus r jede Rotation entfernt bei der $i \notin [l, r] \vee j \notin [l, r]$ gilt. Ähnlich erhält man aus X die Zugriffsfolge X_x^y in dem aus X alle Schlüssel k entfernt werden, für die $k < x \vee k > y$ gilt.

lower bound tree Ein lower bound tree Y zu T ist ein BST, der genau $2|K| - 1$ Knoten enthält. Seine $|K|$ Blätter enthalten die Schlüssel aus K. Die $|K| - 1$ internen Knoten enthalten die Schlüssel aus der Menge $\{r \in R | \exists i, j \in K \colon (i + 1 = j \wedge r = i + 0,5)\}$. Y kann immer erstellt werden indem zunächst ein BST Y_i mit den internen Knoten von Y erzeugt wird. Ein Blatt wird dann an der Position angefügt, an der die Standardvariante von *insert* angewendet auf Y_i ihren Schlüssel einfügen würde. Dass hierbei für zwei Blätter mit Schlüssel k_1, k_2 die gleiche Position gewählt wird ist ausgeschlossen, da es einen internen Knoten mit Schlüssel k_i so geben muss dass $(k_1 < k_i < k_2) \vee (k_1 > k_i > k_2)$ gilt. An der Konstruktionsanleitung ist zu erkennen, dass zu den meisten BST mehrere mögliche lower bound trees existieren. Abbildung 12 zeigt eine beispielhafte Konstellation.

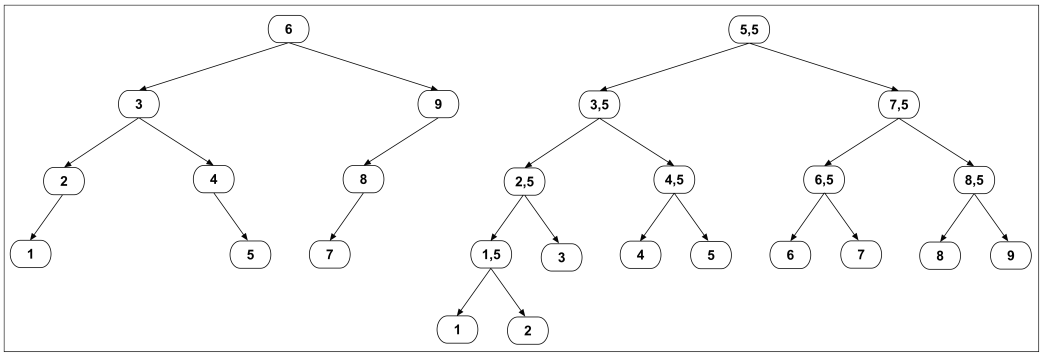

Abbildung 12: Rechts ist ein möglicher lower bound tree zum linken BST dargestellt.

Nun wird die Funkion ${}_X(T, Y, X)$ vorgestellt. Ihre Parameter sind ein BST T, ein lower bound tree Y und eine Zugriffsfolge X. Y und X müssen passend für T erstellt sein, ansonsten ist ${}_X(T, Y, X)$ undefiniert . Die Auswertung erfolgt zu einer natürlichen Zahl. Sei U die Menge der internen Knoten von Y und m die Länge von X. Sei $u \in U$ und l der kleinste Schlüssel eines Blattes im Teilbaum mit Wurzel u, sowie r der größte Schlüssel eines solchen Blattes. Sei v der tiefste gemeinsame Vorfahre der Knoten mit Schlüssel aus $[l, r]$ in T. Sei o die Folge $o_0, o_1, .., o_m = key(v) \circ X_l^r$. $i \in [1, m]$ ist eine *u-Transition* wenn gilt $(o_{i-1} < u \wedge o_i > u) \vee (o_{i-1} > u \wedge o_i < u)$.Die Funktion $score\,(u) : U \to \mathbb{N}$

ist definiert durch $score\,(u) = |\{i \in \mathbb{N} \mid i \textit{ ist eine } u\textit{-Transition}\}|$. Mit Hilfe von *score* kann nun ${}_X(T, Y, X)$ definiert werden.

$$ {}_X(T, Y, X) = m + \sum_{u \in U} score\,(u) $$

Im eigentlichen Satz wird $W\,(X, T) \geq {}_X(T, Y, X)$ gezeigt werden. Dafür werden aber noch ein Lemma und einige Begriffe benötigt. Der **linke innere Pfad** $(v_0, v_1, .., v_n)$ eines Knotens v ist der längst mögliche Pfad für den gilt, v_0 ist das linke Kind von v und für $i \in \{1, .., n\}$,v_i ist das rechte Kind von v_{i-1}. Der **rechte innere Pfad** $(v_0, v_1, .., v_n)$ eines Knotens v ist der längst mögliche Pfad für den gilt, v_0 ist das rechte Kind von u und v_i ist das linke Kind von v_{i-1}.
T_l^r ist ein mit $[l, r]$ von T abgeleiteter BST, so dass er genau die Schlüssel aus T enthält, die in $[l, r]$ liegen. Sei v_d der tiefste gemeinsame Vorfahre der Knoten mit Schlüssel aus $[l, r]$ in T. (Existiert ein solcher nicht ist T_l^r der leere Baum). Es muss $key(v_d) \in [l, r]$ gelten. Denn hat v_d keine Kinder ist sein Schlüssel der Einzige aus $[l, r]$. Hat v_d ein Kind v_c und $key(v_d) \notin [l, r]$, dann wäre v_c ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten. Hat v_d zwei Kinder gibt es drei Fälle:

- Im linken und rechten Teilbaum von v_d sind Schlüssel aus $[l, r]$ enthalten. Dann muss aufgrund der Links-Rechts-Beziehung $key(v_d)$ auch in $[l, r]$ enthalten sein.
- In genau einem Teilbaum von v_d sind Schlüssel aus $[l, r]$ enthalten. Sei v_c die Wurzel dieses Teilbaumes. Gilt zusätzlich $key(v) \notin K_l^r$, dann wäre v_c ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten.
- In den beiden Teilbäumen sind keine Schlüssel aus $[l, r]$ enthalten. Dann muss $key(v_d)$ der Einzige in T_l^r enthaltene Schlüssel sein.

Ein Knoten u_d mit Schlüssel $key(v_d)$ bildet die Wurzel von T_l^r. Nun wird beschrieben wie Knoten zu T_l^r hinzugefügt werden. Dazu werden zwei Mengen verwendet. U ist eine zu Beginn leere Menge, W enthält zu Beginn u_d.

1. Gilt $U = W$, beende das Verfahren.
2. Sei $w \in W$ ein Knoten mit $w \notin U$. Sei v der Knoten in T mit $key(w) = key(v)$. Sei P_l der linke innere Pfad von v und P_r der rechte innere Pfad von v.
3. Ist P_l der leere Pfad weiter mit 5.

4. Sei k_l der Schlüssel des Knoten mit der kleinsten Tiefe in P_l, für den gilt $k \geq l$. Erzeuge einen Knoten w_l mit Schlüssel k_l und füge ihn als linkes Kind an w an. Füge w_l zu W hinzu.

5. Ist P_r der leere Pfad weiter mit 7.

6. Sei k_r der Schlüssel des Knoten mit der kleinsten Tiefe in P_r, für den gilt $k \leq r$. Erzeuge einen Knoten w_r mit Schlüssel k_r und füge ihn als rechtes Kind an w an. Füge w_r zu W hinzu.

7. Füge w zu U hinzu, weiter mit 1

Das Verfahren muss terminieren da die Anzahl der Knoten von T endlich ist. So konstruiert muss T_l^r ein BST sein. Ein Beispiel stellt Abbildung 13 dar.

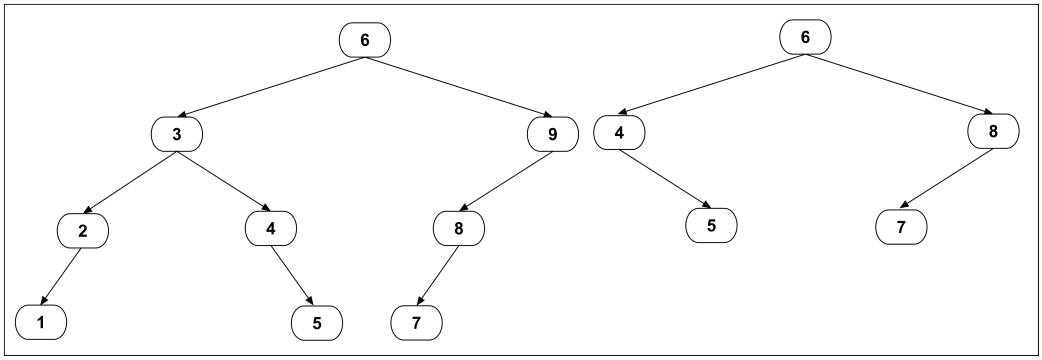

Abbildung 13: Links ein BST T. Rechts ein davon abgeleiteter BST T_4^8 .

Sei K_1 die Schlüsselmenge von T und K_2 die von T_l^r. Sei $K_l^r = K_1 \cap \{i \in \mathbb{N} | i \in [l, r]\}$. Jetzt wird noch darauf eingegangen warum $K_2 = K_l^r$ gilt

$K_2 \subseteqq K_l^r$ ergibt sich direkt aus dem Verfahren zur Konstruktion von T_l^r.

$K_l^r \subseteqq K_2$:
Sei $k \in K_l^r$ und v_k der Knoten in T mit $key(v_k) = k$. Es muss einen Pfad $P_T = v_0, .., v_n$ in T geben, mit $v_0 = v_d$, $v_n = v_k$. Sei m die Anzahl der Knoten in P_T, mit einem Schlüssel in $[l, r]$. Nun folgt Induktion über m.
Für $m = 1$ gilt $k = key\,(v_d)$ und $k \in K_2$.
Induktionsschritt:
Sei v_w der Knoten mit der größten Tiefe in $v_0, .., v_{n-1}$ mit $key(v_w) \in K_2$. Nach Induktionsvoraussetzung gibt es einen Knoten u_w mit $key(u_w) = key(v_w)$ in T_l^r. Es sei $key(v_w) > key(v_k)$, der andere Fall ist symmetrisch. Ist v_k das linke Kind von v_w, dann enthält das linke Kind von u_w den Schlüssel $key(v_k)$.

Anderenfalls gilt für alle v_j mit $w < j < k$, $key(v_j) < l < key(v_k)$. Somit muss v_{w+1} ein linkes Kind sein und die Knoten in P_T mit größerer Tiefe als der von v_{w+1} müssen rechte Kinder sein. Damit ist auch in diesem Fall ein Knoten u_k mit $key(u_k) = key(v_k)$ linkes Kind von u_w.

Nun kommen wir zum Lemma:

Sei v ein Knoten in T, dann wird ein Knoten in T_l^r mit Schlüssel $key(v)$ mit v^* bezeichnet.

Lemma 4.1. *Es sei T ein BST mit Knoten u, v so, dass u ein Kind von v ist. T' ist der BST, der durch ausführen der Rotation $(key(u), key(v))$ aus T entsteht. Gilt $key(u), key(v) \in [l, r]$, dann ist T'^r_l der BST der aus T_l^r durch Ausführen von $(key(u), key(v))$ entsteht. Anderenfalls gilt $T'^r_l = T_l^r$.*

Beweis. Für $u, v \notin [l, r]$ wird bei keinem inneren Pfad ein Knoten mit Schlüssel aus $[l, r]$ entfernt oder hinzugefügt. Nun werden die vier Fälle betrachtet bei denen entweder $key(u)$ oder $key(v)$ in $[l, r]$ liegt.

1. u ist das linke Kind von v und $key(u) < l$:
   Sei w ein Knoten aus T_l^r und w' einer aus T'^r_l, mit $key(w) = key(w')$ und $key(w) \in [l, r]$. Es muss gezeigt werden, dass wenn w ein linkes bzw. rechtes Kind mit Schlüssel k hat, dann gilt dies auch für w'. Da $key(u) < l \leq key(w)$ gilt, kann weder u noch v im rechten Teilbaum von w liegen. Somit ist bezüglich der rechten Kinder nichts zu zeigen. Sei P_l der linke innere Pfad von w. Ist v nicht in P_l enthalten und gilt $v \neq w$ dann gilt $P_l = P_l'$. Sei $w = v$, dann gilt $P_l = u \circ P_l'$, vergleiche Abbildung **??**, und da $key(u) < l$, bleibt das linke Kind von w unverändert. Nun sei v in P_l enthalten. Dann unterscheiden sich P_l und P_l' dadurch, dass ein Knoten mit $key(u)$ in P_l' enthalten ist. Mit $u < l$ gilt aber, dass sich w und w' bezüglich des Schlüssels ihres linken Kindes nicht unterscheiden.

2. u ist das linke Kind von v und $key(v) > r$:
   Mit vertauschen der Bezeichnungen von v und u, erreicht man von T' aus Fall 3, mit Ausführung der Rotation auf dieser Konstellation wieder T aus Fall 3. Somit muss nichts weiter gezeigt werden.

3. u ist das rechte Kind von v und $key(u) > r$:
   Links-Rechts-Symmetrisch zu Fall 1.

4. u ist das rechte Kind von v und $key(v) < l$:
   Links-Rechts-Symmetrisch zu Fall 2.

Übrig bleibt noch die Konstellation $key(u), key(v) \in [l, r]$. Betrachtet wird eine Rechtsrotation $(key(u), key(v))$, die Linksrotation ist wieder symmetrisch. Zu zeigen ist ${T'}_l^r = T_l^{r\prime}$.
In T verändern sich maximal drei innere Pfade.

1. Sei u_r das rechte Kind von u. Sei $u, u_r, v_1, .., v_n$ der linke innere Pfad von v, dann ist ${u_r}', {v_1}', .., {v_n}'$ der linke innere Pfad von v'. Es gilt $l \leq key(u) < key(u_r) < key(v) \leq r$. Damit ist ${u_r}'^*$ das linke Kind von v'^*.

2. Sei $v_1, .., v_n$ der rechte innere Pfad von u, dann ist $v', {v_1}', .., {v_n}'$ der rechte innere Pfad von u'. Damit v'^* ist das rechte Kind von u'^*.

3. Ist v das linke bzw. rechte Kind eines Knoten z mit $key(z) \in [r, l]$, dann sei $v, v_1, .., v_n$ der linke bzw. rechte innere Pfad von z. Dann ist $u', v', {v_1}', .., {v_n}'$ der linke bzw. rechte innere Pfad von z'. Dann u'^* das linke bzw. rechte Kind von z'^*.

Nun wird auf T_l^r die Rotation $(key(u^*), key(v^*))$ ausgeführt. ${u_r}^{*\prime}$ ist linkes Kind von $v^{*\prime}$. $v^{*\prime}$ das rechte Kind von $u^{*\prime}$. Ist v^* das linke bzw. rechte Kind eines Knoten z^*, dann ist $u^{*\prime}$ das linke bzw. rechte Kind von $z^{*\prime}$ und $u^{*\prime}$ das linke bzw. rechte Kind von $z^{*\prime}$. Damit gilt ${T'}_l^r = T_l^{r\prime}$.

□

Satz 4.1. *Es sei T ein standard offline BST mit Schlüsselmenge $K = \{i \in \mathbb{N} | i \in [j, k] \text{ mit } j, k \in \mathbb{N}\}$. Sei Y ein für T erstellter lower bound tree und X eine zu T erstellte Zugriffsfolge mit Länge m. Dann gilt $W(X, T) \geq {}_X(T_0, Y, X)$.*

Beweis. Sei U die Menge der internen Knoten von Y. Die Kosten zum Ausführen von X sind die *Anzahl der Einzelschritte* $+ m$. Es reicht also aus zu zeigen, dass mehr als $\sum_{u \in U} score(u)$ Rotationen benötigt werden. Es wird Induktion über $n = |K|$ angewendet. Sei $n = 1$, dann gibt es keinen internen Knoten in Y und $\sum_{u \in U} score(u) = 0$. Der Induktionsanfang ist somit gemacht. Im folgenden sei $n \geq 2$.
Sei $R = r_1, r_2, .., r_l$ die Folge der insgesamt durchgeführten Rotationen. Für $i \in \{1, .., r\}$ sei T_i der BST, der entsteht nachdem r_i auf T_{i-1} ausgeführt wurde. Sei w die Wurzel von Y, mit Schlüssel k_w,. Sei Y^1 bzw. Y^2 der linke bzw. rechte Teilbaum von w. Es ist zu beachten, dass Y^1 ein lower bound tree zu $T_1^{k_w}$ ist und Y^2 einer zu $T_{k_w}^{\infty}$. $T_{i1}^{k_w}$ wird im folgenden als T_i^1 bezeichnet und $T_{ik_w}^{\infty}$ als T_i^2. Da $n \geq 2$ muss w ein interner Knoten sein. Sei

$R^1 = r^1{}_1, r^1{}_2, .., r^1{}_{l^1} = R_1^{k_w}$ und $R^2 = r_1^2, r_2^2, .., r_{l^2}^2 = R_{k_w}^{\infty}$. Mit M wird die Folge bezeichnet, die entsteht, wenn aus R alle Rotationen entfernt werden, die in R^1 oder R^2 enthalten sind. Sei l_M die Länge von M. Es muss $l = l^1 + l^2 + l_M$ gelten, da keine Rotation sowohl in R^1 als auch in R^2 enthalten sein kann. X_1 ist die Folge die entsteht wenn aus X alle Schlüssel $k > k_w$ entfernt werden. X_2 entsteht durch entfernen aller Schlüssel $k < k_w$ aus X. Für $j \in \{1, 2\}$, sei U^j die Menge der internen Knoten von Y^j. Sei $T_0^{j*}, T_1^{j*}, .., T_{l^t}^{j*}$ die entstehende Folge, wenn aus $T_0^j, T_1^j, .., T_l^j$ die T_t^j entfernt werden für die $T_{t-1}^j = T_t^j$ gilt.
Mit Lemma 4.2 kann T_t^{j*} durch Ausführung der Rotation r_t^j auf T_{t-1}^{j*} abgeleitet werden. Dadurch folgt durch dieses Lemma, dass wenn ein Knoten mit Schlüssel $k < w$ bzw. $k > w$ die Wurzel von T_t ist dann muss die Wurzel von T_t^1 bzw. T_t^2 auch Schlüssel k haben. R^j bringt also der Reihe nach, die Knoten mit den Schlüsseln aus X^j an die Wurzel von T^j und X^j kann als Zugriffsfolge für T^j aufgefasst werden. Da die Knotenzahl in T^j kleiner n sein muss gilt mit der Induktionsvoraussetzung $l_j \geq \sum_{u \in U^j} score(u)$.
Sei $\sigma = key(w) \circ X$. Sei a eine w-Transition. Nun wird angenommen dass $\sigma_{a-1} < key(w) \wedge \sigma_a > key(w)$. Der andere Fall kann davon problemlos abgeleitet werden. Sei y der Knoten in T mit $key(y) = \sigma_{a-1}$ und z der Knoten in T mit $key(z) = \sigma_a$. Nach *access(σ_{a-1})* ist y die Wurzel von T. z muss sich im rechten Teilbaum von y befinden. Nach *access(σ_a)* ist z die Wurzel von T. y muss sich im linken Teilbaum von z befinden. Somit muss während *access(σ_a)* die Rotation $(key(z), key(y))$ ausgeführt worden sein. $(key(z), key(y))$ muss in M enthalten sein. Für jede w-Transition ist also mindestens eine Rotation in M enthalten, also $l_M \geq score\,(w)$.
Zusammengefasst ergibt sich:

$$l = l^1 + l^2 + l_M \geq \sum_{u \in U^1} score(u) + \sum_{u \in U^2} score(u) + score(w)$$

□

Daraus folgt direkt $OPT\,(X) \geq {}_X(T, Y, X)$ für beliebige BST T.

4.3 bit reversal permutation

In diesem Abschnitt wird gezeigt, dass es Zugriffsfolgen mit Länge m für BST T gibt, so dass für die Laufzeit $\Theta\,(m \log n)$ gilt, mit n ist die Anzahl der Knoten von T. Hier werden speziell die Zugriffsfolgen betrachtet, die als **bit reversal permutation** bezeichnet werden. Auf $O\,(m \log n)$ wird hier nicht weiter eingegangen. Die balancierten BST garantieren jedoch diese Schranke

und mit dem Rot-Schwarz-Baum wird später ein solcher noch vorgestellt. $\Omega(m \log n)$ wird mit Hilfe der ersten unteren Schranke von Wilber gezeigt und ein Beweis ist ebenfalls in [2] enthalten.
Nun wird zunächst der Aufbau einer solchen Zugriffsfolge eingegangen. Sei $l \in \mathbb{N}$ und $i \in \{0, 1, .., l-1\}$. Eine Folge $b_{l-1}, b_{l-2}, .., b_0$ mit $b_i \in \{0, 1\}$, kann als Zahl zur Basis 2 interpretiert werden. T enthält alle Schlüssel die als solche Folge dargestellt werden können. Die Schlüsselmenge von T ist deshalb $K_l = \{0, 1, .., 2^l - 1\}$. Die Funktion $br_l(k)\colon K \to K$ ist wie folgt definiert. Sei $b_{l-1}, b_{l-2}, .., b_0$ die Binärdarstellung von k, dann gilt

$$br_l(k) = \sum_{i=0}^{l-1} b_{(l-1-i)} \cdot 2^i$$

$br_l(k)$ gibt also gerade den Wert der „umgekehrten“ Binärdarstellung von k zurück. Die bit reversal permutation zu l ist die Zugriffsfolge
$br_l(0), br_l(1), .., br_l(2^l - 1)$. Diese wird ab jetzt mit X bezeichnet. Tabelle 1 zeigt die bit reveral permutation mit $l = 4$. Sei $y = \max(K_l)/2 = 2^{l-1} - 0,5$. Da b_0 in den Binärdarstellungen zu $0, 1, .., 2^l - 1$ alterniert, alterniert b_{l-1} in X. Aus $2^{l-1} > y$ folgt $br_l(k) < y \Rightarrow br_l(k+1) > y$ und $br_l(k) > y \Rightarrow br_l(k+1) < y$. Da $|K_l| = 2^l$ kann zu T ein vollständig balancierter lower bound tree Y erstellt werden. Sei w die Wurzel von Y. Da im linken Teilbaum von w genau so viele Blätter wie im rechten vorhanden sein müssen, kann nur y der Schlüssel von w sein. Zu einer Zugriffsfolge $X = x_0, x_1, .., x_m$ bezeichnet X_l^r wieder die Zugriffsfolge, die entsteht wenn aus X alle Schlüssel k, mit $k < l \vee k > r$ entfernt werden. $X + i$ mit $i \in \mathbb{N}$ bezeichnet im Folgenden die Folge $x_0 + i, x_1 + i, .., x_m + i$.

Korollar 4.1. *Sei $l \in \mathbb{N}$. Sei T ein BST mit Schlüsselmenge $K_l = \{0, 1, .., 2^l - 1\}$ und $n = 2^l$. Sei $X = x_0, x_1, .., x_{n-1}$ die bit reversal permutation zu l und Y der vollständig balancierte lower bound tree zu T. Dann gilt $W(X, T) \geq n \log_2(n) + 1$.*

Beweis. Sei U die Menge der internen Knoten von Y. Mit Satz 4.2 reicht es aus

$$\sum_{u \in U} score(u) \geq n \log_2 n + 1 - n$$

zu zeigen. Dies geschieht mit Induktion über l. Für $l = 0$ besteht Y aus einem einzigen Blatt. Damit gilt

i	$bin\,(i)$	$bin\,(br\,(i))$	x_i
0	0000	0000	0
1	0001	1000	8
2	0010	0100	4
3	0011	1100	12
4	0100	0010	2
5	0101	1010	10
6	0110	0110	6
7	0111	1110	14
8	1000	0001	1
9	1001	1001	9
10	1010	0101	5
11	1011	1101	13
12	1100	0011	3
13	1101	1011	11
14	1110	0111	7
15	1111	1111	15

Tabelle 1: bit reveral permutation für $l = 4$

$W\,(X,T) \geq \sum_{u\in U} score\,(u) + 1 > 0 = n\log_2 n + 1 - n$.
Nun sei $l > 0$. Sei w die Wurzel von Y, mit $k_w = key(w)$. Sei $T_0^{k_w}$ ein BST mit Schlüsselmenge $K_0^{k_w} = \{k \in \mathbb{N} | k \leq k_w\} = \{k \in \mathbb{N} | k \leq 2^{l-1} - 1\}$ und $T_{k_w}^{\infty}$ ein BST mit Schlüsselmenge $K_{k_w}^{\infty} = \{k \in \mathbb{N} | \exists n \in K_0^{k_w} : k = n + 2^{l-1}\}$. Sei Y^1 bzw. Y^2 der linke bzw. rechte Teilbaum von w und U^1 bzw. U^2 die Menge der internen Knoten von Y^1 bzw. Y^2. Y^1 und Y^2 sind vollständig balancierte lower bound trees zu $T_0^{k_w}$ und $T_{k_w}^{\infty}$. $X_0^{k_w}$ ist die bit reversal permutation für $T_0^{k_w}$. Außerdem gilt $X_{k_w}^{\infty} = X_0^{k_w} + 2^{l-1}$. Mit der Induktionsvoraussetzung gilt deshalb, für $i \in \{1, 2\}$,

$$\sum_{u\in U^i} score\,(u) \geq \frac{n}{2}\log_2\left(\frac{n}{2}\right) + 1 - \frac{n}{2}$$

Aus $(x_j < k_w \Rightarrow x_{j-1} > k_w) \wedge (x_j > k_w \Rightarrow x_{j-1} < k_w)$ folgt $score\,(w) \geq n - 1$.

Zusammenfassen ergibt

$$\begin{aligned}\sum_{u \in U} score\,(u) &\geq 2\left(\frac{n}{2}\log_2\left(\frac{n}{2}\right) + 1 - \frac{n}{2}\right) + n - 1 \\ &= n(l-1) + 1 \\ &= nl + 1 - n \\ &= n\log_2(n) + 1 - n\end{aligned}$$

□

Die Schlüsselmenge wurde beim Korollar auf $K_l = \{0, 1, .., 2^l - 1\}$ festgelegt. Vielleicht wäre es aber mit einer anderen Schlüsselmenge K möglich X schneller auszuführen ? In jedem Fall müsste $K_l \subsetneqq K$ gelten. Sei R die Folge von Rotationen, die beim Ausführen von X bei einem BST T mit Schlüsselmenge K entsteht. Sei $y = 2^l - 1$ Mit Lemma 4.2 ist dann R_0^y eine Folge von Rotationen zum Ausführen von X auf T_0^y und die Länge von R kann nicht kleiner als die von R_0^y sein.

4.4 Amortisierte Laufzeitanalyse

Im nächsten Anschnitt werden die Kosten von amortisierten Laufzeitanalysen verwendet. Deshalb wird diese hier nun vorgestellt. Sei $i \in \{0, .., m\}$. Bei der **amortisierten Laufzeitanalyse** wird eine Folge von m Operationen betrachtet. Hierbei kann es sich m mal um die gleiche Operation handeln, oder auch um verschiedene. Die **tatsächlichen Kosten** t_i stehen für die Kosten zum ausführen der i-ten Operation. Durch aufaddieren der tatsächlichen Kosten jeder einzelnen Operation erhält man **tatsächlichen Gesamtkosten**. Stehen für die Laufzeit der Operationen jeweils nur obere Schraken zur Verfügung, kann man mit diesen genau so vorgehen, um eine obere Schranke für die Gesamtlaufzeit zu erhalten. So erzeugte obere Schranken können jedoch unnötig hoch sein. Die Idee bei einer amortisierten Analyse ist es, eingesparte Zeit durch schnell ausgeführte Operationen, den langsameren Operationen zur Verfügung zu stellen. Dabei wird insbesondere der aktuelle Zustand der zugrunde liegenden Datenstruktur vor und nach einer Operation betrachtet. Es gibt drei Methoden zur amortisierten Analyse, bei BST wird in der Regel die **Potentialfunktionmethode** verwendet.

Potentialfunktionmethode Eine Potentialfunktion $\Phi(D)$ ordnet einem Zustand einer Datenstruktur D eine natürliche Zahl, **Potential** genannt, zu. Es bezeichnet $\Phi(D_i)$ das Potential von D nach Ausführung der i-ten

Operation. Die **amortisierten Kosten** a_i einer Operation berücksichtigen die von der Operation verursachte Veränderung am Potential, $a_i = t_i + \Phi(D_i) - \Phi(D_{i-1})$. Um die **amortisierten Gesamtkosten** A zu berechnen bildet man die Summe der amortisierten Kosten aller Operationen.

$$A = \sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \left(t_i + \Phi\left(D_i\right) - \Phi\left(D_{i-1}\right)\right) = \Phi\left(D_m\right) - \Phi\left(D_0\right) + \sum_{i=1}^{m} t_i$$

Folgendes gilt für die Summe der t_i:

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} \left(a_i - \Phi\left(D_i\right) + \Phi\left(D_{i-1}\right)\right) = \Phi\left(D_0\right) - \Phi\left(D_m\right) + \sum_{i=1}^{m} a_i$$

$$\Rightarrow \left(\Phi\left(D_m\right) \geq \Phi\left(D_0\right) \Rightarrow \sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i\right)$$

Ist das Potenzial nach Ausführung der Operationsfolge also nicht kleiner als zu Beginn, dann sind die amortisierten Gesamtkosten eine obere Schranke für die tatsächlichen Gesamtkosten. Die wesentliche Aufgabe ist es nun eine Potentialfunktion zu finden, bei der die amortisierten Gesamtkosten möglichst niedrig sind und für die gilt $\Phi\left(D_m\right) \geq \Phi\left(D_0\right)$. Dies wird jetzt noch an einem einfachen Beispiel demonstriert.

Potentialfunktionmethode am Beispiel eines Stack Der Stack verfügt wie gewöhnlich über eine Operation *push* zum Ablegen eines Elementes auf dem Stack und über *pop* zum Entfernen des oben liegenden Elementes. Zusätzlich gibt es eine Operation *popAll*, die so oft *pop* aufruft, bis der Stack leer ist. Sei n die Anzahl der Elemente die maximal im Stack enthalten sein kann. *push* und *pop* können in konstanter Zeit durchgeführt werden und wir berechnen jeweils eine Kosteneinheit. Für die Laufzeit von *popAll* gilt $O(n)$, da *pop* bis zu n mal aufgerufen wird. Für die Gesamtlaufzeit einer Folge von m Operationen kann $O(mn)$ angegeben werden. Mit einer amortisierten Analyse wird nun aber $O(m)$ für *popAll* gezeigt. Als Φ verwenden wir eine Funktion, welche die aktuelle Anzahl der im Stack enthaltenen Elemente zurück gibt. Φ_0 setzen wir auf 0, das heißt wir starten mit einem leeren Stack. *push* erhöht also das Potential um eins, während *pop* es um eins vermindert. Nun werden die amortisierten Kosten bestimmt.

$$\begin{aligned} a_{push} &= t_{push} + \Phi_i - \Phi_{i-1} &&= 2 \\ a_{pop} &= t_{pop} + \Phi_i - \Phi_{i-1} &&= 0 \\ a_{popAll} &= n \cdot a_{pop} &&= 0 \end{aligned}$$

Alle drei Operationen haben konstante amortisierte Kosten. Auf jedem Fall gilt $\Phi_m \geq \Phi_0 = 0$. Für die Ausführungszeit der Folge gilt deshalb $O(m)$.
Bei diesem einfachen Beispiel ist sofort klar warum es funktioniert. Aus einem zu Beginn leerem Stack kann nur entfernt werden, was zuvor eingefügt wurde. *push* zahlt für die Operation, welche das eingefügte Element eventuell wieder entfernt gleich mit, bleibt bei den Kosten aber konstant. Deshalb kann *pop* amortisiert kostenlos durchgeführt werden, wodurch einer der beiden Faktoren zur Berechnung der Kosten von *popAll* zu 0 wird.

4.5 Eigenschaften eines dynamisch optimalen BST

Im folgendem werden einige obere Laufzeitschranken für Zugriffssequenzen vorgestellt. Es ist bekannt, dass es obere Schranken sind, da mit dem Splaybaum ein BST bekannt ist, der jede der Schranken einhält. Der Splaybaum wird später noch vorgestellt. Es wird wieder ohne Verlust der Allgemeinheit eine Schlüsselmenge $K = \{1, 2, .., n\}$ angenommen. Wenn nicht anders angegeben wird $X = x_1, x_2, .., x_m$ als Zugriffssequenz verwendet. Es wird $m \geq n$ und angenommen.

Balanced Property Ein BST erfüllt das balanced property, wenn er X in amortisiert $O((m \log(n))$ Zeit ausführt.

Static Finger Property Die Idee hinter dieser Eigenschaft ist, dass es einfacher ist, Zugriffssequenzen schnell auszuführen, wenn ihre Schlüssel betragsmäßig nahe beieinander liegen. Sei $k_f \in K$. Ein BST erfüllt static finger wenn für die amortisierte Laufzeit von X

$$O\left(n \log_2 n + \sum_{i=1}^{m} \log |k_f - x_i| + 1\right)$$

gilt. Ein BST mit der static finger Eigenschaft erfüllt auch die balanced Eigenschaft, denn $|k_f - x_i| < n$.

Statisch optimal Sei $k \in K$ und $q(k)$ die Anzahl des Vorkommens von k in X. Ein BST ist statisch optimal wenn er Zugriffssequenzen, in denen jeder seiner Schlüssel zumindest einmal enthalten ist, in amortisiert

$$O\left(\sum_{k=1}^{n} q(k) \log\left(\frac{m}{q(k)}\right)\right)$$

Zeit ausführt. Der Name kommt daher, dass es sich hierbei um eine untere Schranke für die Ausführungszeit von X bei statischen BST handelt, siehe [3].

Working Set Property Ein BST mit dem working set property führt Zugriffsfolgen schnell aus, bei denen auf die gleichen Schlüssel in kurzen Abständen zugegriffen wird. Für x_i sei $J_i = \{j \in \mathbb{N} | j < i \wedge x_j = x_i\}$. Sei $t_{xi} = \max(J)$, falls J nicht leer ist, ansonsten $t_{xi} = 0$. t_{xi} liefert also den Index des vorherigen Zugriffes auf x_i, falls ein solcher existiert. Sei $w_i = |\{x_j | t_{xi} < j \leq i\}|$. Ein BST erfüllt das working set propery wenn seine amortisierte Laufzeit für X

$$O\left(n \log_2 n + \sum_{i=1}^{m} \log w_i\right)$$

gilt.

Dynamic Finger Property Diese Eigenschaft ist static finger sehr ähnlich, man kann jedoch durch das Unified Property nicht direkt auf dynamic finger schließen. Ein BST erfüllt das Dynamic Finger Property, wenn für die amortisierte Laufzeit von X

$$O\left(m + \sum_{i=2}^{m} \log\left(|x_{i-1} - x_i| + 1\right)\right)$$

gilt.
Abbildung 14 zeigt Implikationen zwischen den Eigenschaften und basiert auf einer Abbildung aus 14.

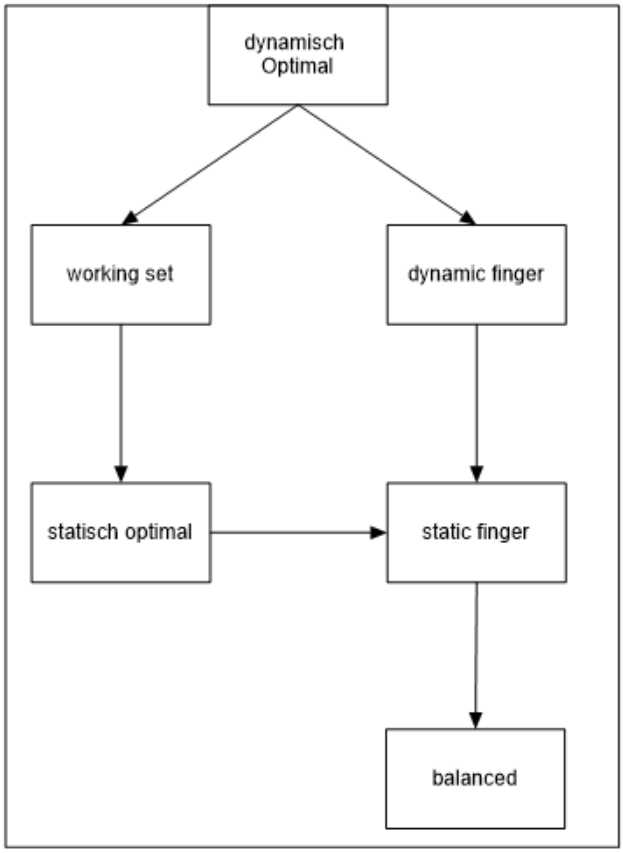

Abbildung 14: Implikationen zwischen den Eigenschaften, abgeleitet aus einer Abbildung aus [4]

Literatur

[1] Karel Culik and Derick Wood. A note on some tree similarity measures. *Information Processing Letters*, 15(1):39 – 42, 1982.

[2] Robert. Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.

[3] Norman Abramson. *Information theory and coding.* McGraw-Hill electronic sciences series. McGraw-Hill, New York, NY, 1963.

[4] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *CoRR*, abs/1603.04892, 2016.