



Bachelorarbeit zum Thema

Tango Bäume

Andreas Windorfer
q8633657

ausgeführt am
Lehrgebiet Theoretische Informatik
Leitung Prof. Dr. André Schulz

Zusammenfassung

Bei binären Suchbäumen sind besonders die Ausführungszeiten ihrer Operationen von Interesse. Hier werden speziell die Ausführungszeiten von Folgen von *access* Operationen betrachtet, im Bezug zur Anzahl der Knoten des binären Suchbaumes n . Beim 1985 vorgestellten Splay Baum [1] wird vermutet, dass dieser jede Folge von *access* Operationen asymptotisch betrachtet, mindestens genau so schnell ausführt, wie jeder andere binäre Suchbaum. Dies wurde als „Dynamische Optimalitäts Vermutung“ bekannt. Bis heute ist offen, ob der Splay Baum diese Eigenschaft besitzt. 2007 wurde dann der Tango Baum [2] vorgestellt. Bei ihm ist bekannt, dass er jede solche Folge asymptotisch betrachtet, höchstens um einen Faktor $\log(\log(n))$ langsamer ausführt, als der jeweils schnellste binäre Suchbaum. Um dies zu beweisen wurde die „Interleave Lower Bound“, eine untere Schranke für die Ausführungszeit solcher Folgen verwendet. Bis dahin war für keinen binären Suchbaum ein Faktor kleiner als $\log(n)$ bewiesen und dieser wird von den balancierten binären Suchbäumen trivial erreicht. Der Tango Baum wird im Detail vorgestellt und zusätzlich noch zwei Variationen zu ihm. Abschließend werden Laufzeittests zwischen dem Tango Baum und dem Splay Baum durchgeführt, wobei der Splay Baum durchgängig die niedrigeren Zeiten liefern wird.

Inhaltsverzeichnis

1 Einleitung	5
2 Binäre Suchbäume	6
2.1 Die Definition des binären Suchbaumes.	6
2.2 Weitere Begriffe und Eigenschaften zum binären Suchbaum . .	8
2.3 Die Grundoperationen <i>search</i> , <i>insert</i> und <i>delete</i>	13
3 Rot-Schwarz-Baum	17
3.1 Grundoperationen	19
4 Dynamische Optimalität	28
4.1 BST Zugriffsfolgen	29
4.2 Die erste untere Schranke von Wilber.	30
4.3 Bit Reversal Permutation	37
4.4 Amortisierte Laufzeitanalyse	39
4.5 Eigenschaften eines dynamisch optimalen BSTs.	41
5 Tango Baum	43
5.1 Die Interleave Lower Bound	44
5.2 Der Aufbau des Tango Baumes.	48
5.3 Die <i>access</i> Operation beim Tango Baum.	52
5.4 Laufzeitanalyse für <i>access</i>	61
5.5 Tango Baum konformes Vereinigen beim Rot-Schwarz-Baum. .	63
5.6 Tango Baum konformes Aufteilen beim Rot-Schwarz-Baum. .	66
6 Splay Baum	70
6.1 Die <i>access</i> Operation beim Splay Baum.	70
6.2 Amortisierte Laufzeitanalyse von <i>splay</i>	73
6.3 Dynamische Optimalitäts Vermutung	74
7 Weitere dynamische Suchbäume	76
7.1 Zipper Baum	76
7.2 Multisplay Baum	79
8 Dokumentation zur Implementierung	81
8.1 Beschreibung der Klassen	83
9 Laufzeittests mit dem Tango Baum und dem Splay Baum.	85
9.1 Verwendete Zugriffsfolgen zu den Laufzeittests.	86
9.2 Durchführung der Laufzeittests.	87

9.2.1	Zufällig erzeugte Zugriffsfolgen	87
9.2.2	Bit Reversal Permutations	91
9.2.3	Static Finger Property	92
9.2.4	Dynamic Finger Property	96
9.2.5	Working Set Property	98
9.2.6	Weitere Laufzeittests	101
9.3	Fazit zu den Laufzeittests	104
10	Allgemeines Fazit und Ausblick	110

1 Einleitung

Binäre Suchbäume werden zur Lösung des Wörterbuchproblems eingesetzt. Hierbei existiert eine Menge von Schlüsseln, denen Daten zugeordnet sind. Im Telefonbuch wäre ein Name beispielsweise ein Schlüssel, dem als Datum eine Telefonnummer zugeordnet ist. Da solche Mengen von Schlüsseln in der Praxis extrem groß werden können, ist es wichtig, diese möglichst effizient zu verwalten. Auf einen einmal eingefügten Schlüssel kann beliebig oft zugegriffen werden. Deshalb müssen vor allem diese Zugriffe effizient sein. Das binäre Suchen bietet sich hierzu als Möglichkeit an. Die Schlüssel müssen dazu auf irgendeine Art und Weise sortiert vorliegen.

Beim Telefonbuch wird die alphabetische Sortierung verwendet. Somit kann mit der Suche eines Namens in etwa der Mitte des Telefonbuches gestartet werden. Nach einem Vergleich können dann zumindest die Schlüssel einer Hälfte des Buches ausgeschlossen werden. Dieses Verhalten iteriert bis der gesuchte Name gefunden ist. Obwohl es viele Varianten von binären Suchbäumen mit zum Teil aufwändigem Verhalten gibt, ahmen sie im Grunde das binäre Suchen nach.

Binäres Suchen könnte jedoch auch mit einem einfachen Array umgesetzt werden. Hierbei ergibt sich bei Änderungen an der Schlüsselmenge jedoch das Problem, das Array effizient anzupassen. Wird z.B. ein Schlüssel entfernt, müssen alle Nachfolgenden um ein Feld verschoben werden. Beim Einfügen eines Schlüssels könnte sogar ein Umzug in ein größeres Array notwendig werden. Bei binären Suchbäumen können solche Änderungen effizienter durchgeführt werden.

Das Auffinden eines Schlüssels erfolgt beim binären Suchbaum mit einer *access* Operation. Eine Eigenschaft des binären Suchbaumes ist es, dass weiter oben liegende Schlüssel schneller erreicht werden können als andere. Ist bekannt, wie häufig auf welchen Schlüssel zugegriffen wird, kann diese Eigenschaft ausgenutzt werden. Ein Schlüssel wird dann umso weiter oben platziert, je häufiger auf ihn zugegriffen wird.

Ein weiterer Versuch ein effizientes Verhalten zu erreichen sind die dynamischen binären Suchbäume. Im Gegensatz zu statischen binären Suchbäumen ändern diese ihre Struktur auch bei *access* Operationen. Dies macht sie unabhängiger von den oft unbekannten Häufigkeiten der Zugriffe auf die einzelnen Schlüssel. Der Tango Baum ist ein solcher Vertreter, ebenso wie der Splay Baum, der in dieser Arbeit als Vergleichspartner dient.

2 Binäre Suchbäume

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

2.1 Die Definition des binären Suchbaumes.

Ein **Baum** T ist ein minimal zusammenhängender gerichteter Graph. Ein Baum ohne Knoten ist ein **leerer Baum**. In einem nicht leeren Baum gibt es genau einen Knoten ohne eingehende Kante. Dieser wird als **Wurzel** bezeichnet. Alle anderen Knoten haben genau eine eingehende Kante. Enthält der Baum eine Kante von Knoten v_1 zu Knoten v_2 , so ist v_2 ein **Kind** von v_1 und v_1 ist der **Elternknoten** von v_2 . Die Wurzel hat also keinen Elternknoten, alle anderen Knoten haben genau einen. Ein **Pfad** P ist eine Folge von Knoten (v_0, v_1, \dots, v_n) mit $\forall i \in \{1, 2, \dots, n\}: v_{i-1} \text{ ist der Elternknoten von } v_i$. n ist die **Länge des Pfades**. (v_0) ist ein Pfad der Länge 0. Jeder Knoten v in T ist die Wurzel eines **Teilbaumes** $T(v)$. Dieser entsteht indem alle Knoten u aus T entfernt werden, zu denen es keinen Pfad mit $v_0 = v$ und $v_n = u$ gibt. Knoten ohne ausgehende Kante werden **Blatt** genannt. Jeder andere Knoten wird als **innerer Knoten** bezeichnet.

Bei einem **binären Baum** kommt folgende Einschränkung hinzu:

Ein Knoten hat maximal zwei Kinder.

Entsprechend ihrer Zeichnung werden die Kinder in Binärbäumen als **linkes Kind** oder **rechtes Kind** bezeichnet. Sei w das linke, bzw. rechte Kind von v , dann wird der Teilbaum mit der Wurzel w als **linker Teilbaum**, bzw. **rechter Teilbaum** von v bezeichnet.



Abbildung 1: Ein binärer Suchbaum.



Abbildung 2: Kein binärer Suchbaum.

Bei einem **binären Suchbaum** ist jedem Knoten v ein innerhalb der Baumstruktur eindeutiger **Schlüssel** $key(v)$ aus einem **Universum** zugeordnet, auf dem eine totale Ordnung definiert ist. Auf totale Ordnungen wird in diesem Kapitel noch eingegangen. Falls nicht explizit anders angegeben, wird hier und in den folgenden Kapiteln als Universum immer \mathbb{N} verwendet, wobei die 0 enthalten ist. Die in einem binären Suchbaum enthaltenen Schlüssel bezeichnen wir als seine **Schlüsselmenge**. Damit aus dem binären Baum ein binärer Suchbaum wird, ist noch folgende Eigenschaft notwendig:

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.

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

2.2 Weitere Begriffe und Eigenschaften zum binären Suchbaum

Zwei verschiedene Knoten mit dem selben Elternknoten sind **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 die Höhe 0 hat. Ein BST T mit Höhe h_T wird von oben nach unten in die **Ebenen** $1, 2, \dots, h_T$ unterteilt. Die Wurzel liegt in der Ebene eins, deren Kinder in der Ebene zwei, usw. Enthält eine Ebene ihre maximale Anzahl an Knoten ist sie **vollständig besetzt**.



Abbildung 3: Ein weiterer binärer Suchbaum

Da im linken Teilbaum nur kleinere Schlüssel vorhanden sein dürfen und im rechten Teilbaum nur größere, kann die Schlüsselmenge eines binären Suchbaumes, von links nach rechts, in aufsteigend sortierter Form abgelesen werden. Aus Platzgründen passiert es bei Zeichnungen von BSTs manchmal, dass ein Knoten in einem linken Teilbaum weiter rechts liegt als die Wurzel dieses Teilbaumes oder umgekehrt, weshalb bei der Betrachtung solcher Zeichnungen etwas vorsichtig vorgegangen werden muss. Abbildung 4 enthält keine solche Konstellation.



Abbildung 4: Die Schlüssel sind aufsteigend sortiert ablesbar.

Algorithmisch können die im BST enthaltenen Schlüssel aufsteigend sortiert durch eine **Inorder-Traversierung** ausgegeben werden. Es ist ein rekursives Verfahren, das an der Wurzel startet und pro Aufruf drei Schritte ausführt:

Algorithmus *inorder* (Node v)

1. Existiert das linke Kind v_l von v , rufe *inorder*(v_l) auf.
2. Gib den Schlüssel von v aus.
3. Existiert das rechte Kind v_r von v , rufe *inorder*(v_r) auf.

Dass das Verfahren funktioniert, wird sichtbar, 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 der Knotenzahl $n + 1$ und der Wurzel w . Sowohl für den linken, als auch für den rechten Teilbaum von w gilt, dass die Anzahl der enthaltenen 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 Knotens v mit dem Schlüssel k_v wird der Knoten mit dem größten im BST enthaltenen Schlüssel k_p , für den $k_p < k_v$ gilt, bezeichnet. Aus der Inorder-Traversierung kann eine Anleitung zum Finden des Vorgängers abgeleitet werden. Falls ein linker Teilbaum vorhanden ist, wird der größte Schlüssel in diesem, also der am weitesten rechts liegende, direkt vor k_v ausgegeben. Ansonsten 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 wird der Knoten mit dem kleinsten im BST enthaltenen Schlüssel k_s , für den $k_s > k_v$ gilt, bezeichnet. Da dieser Schlüssel bei der Inorder-Traversierung direkt nach v ausgegeben wird, ist der zugehörige Knoten ganz links im rechten Teilbaum von v zu finden, falls ein solcher vorhanden ist. Ansonsten ist es der tiefste Knoten auf dem Pfad von der Wurzel zu v , bei dem v im linken Teilbaum liegt. Abbildung 5 zeigt den Vorgänger und den Nachfolger eines Knotens.

Als **Vorfahre** eines Knotens v werden alle Knoten auf dem Pfad von der Wurzel zu v , inklusive v selbst, bezeichnet.

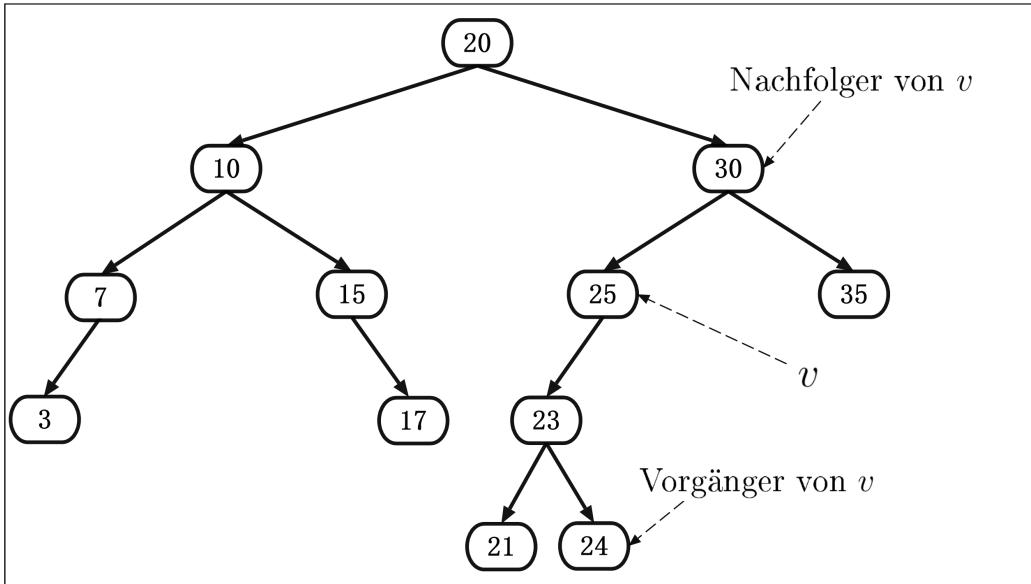


Abbildung 5: Darstellung von Vorgänger und Nachfolger.

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 BSTs durch Rotationen: Wird ein BST durch eine Veränderung in einen anderen BST überführt, kann es passieren, dass sich die Eigenschaften eines Knotens 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 Hochstrich bezieht sich dann auf den selben Knoten nach der Änderung. Beispielsweise bezieht sich x auf den Knoten mit dem Schlüssel k in der Ausgangssituation, dann bezieht sich x' auf den Knoten mit dem Schlüssel k nach dem Ausführen der Änderung.

Rotationen: können verwendet werden, um lokale Änderungen an der Struktur eines BSTs 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' das linke, bzw. rechte Kind von dem Knoten, von dem z das linke, bzw. rechte Kind war. z' ist das linke Kind von x' . Die Wurzel von B' ist das rechte Kind von z' . Unabhängig von der Anzahl der im BST enthaltenen Knoten und der Ausführungsstelle im BST ist eine Linksrotation daher mit dem Aufwand verbunden, maximal drei Zeiger umzusetzen. Abbildung 7 zeigt die symmetrische Rechtsrotation. Dass es durch eine Rotation zu keiner Verletzung der BST Eigenschaften kommt, kann den Abbildungen direkt entnommen werden. In Abbildung 8 ist zu erkennen, dass sich die Wirkung einer Rotation auf x durch eine gegenläufige Rotation auf z' aufheben lässt.

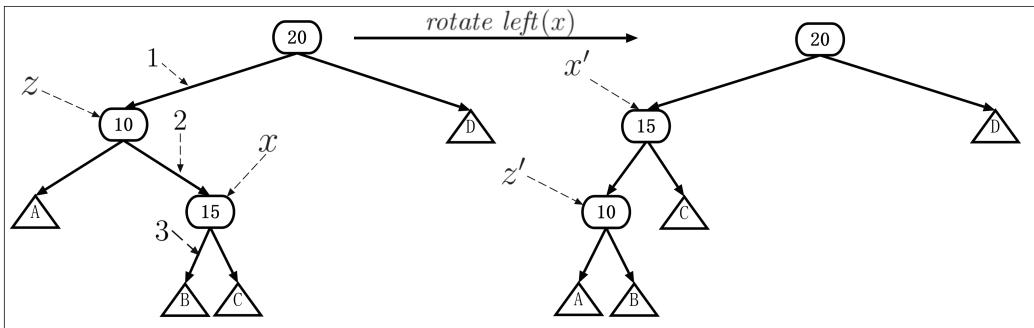


Abbildung 6: Linksrotation auf Knoten x .

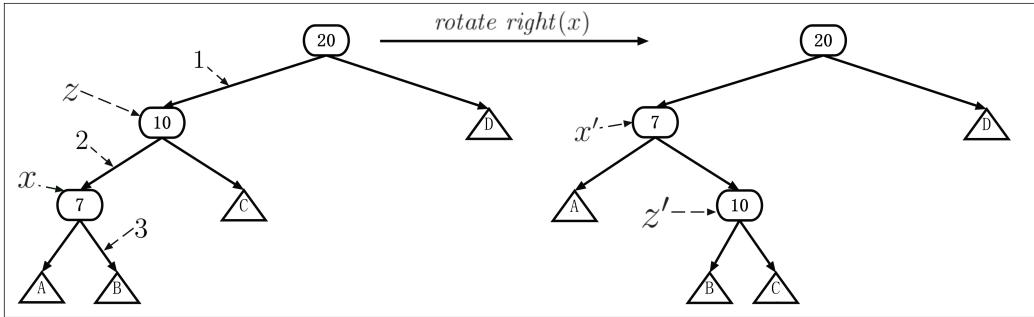


Abbildung 7: Rechtsrotation auf Knoten x.

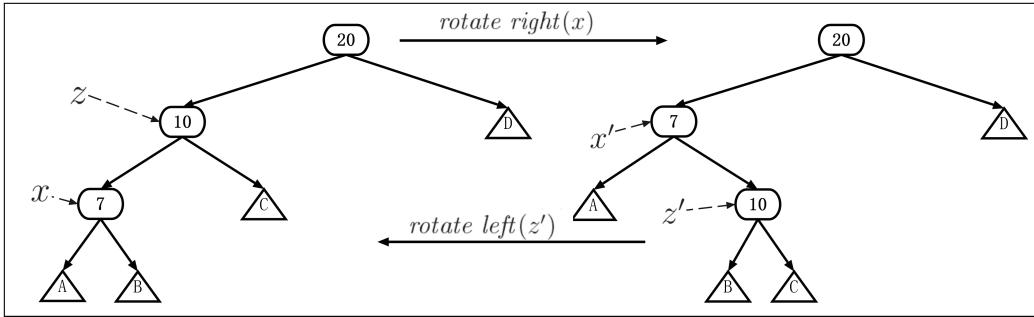


Abbildung 8: Gegenseitiges Aufheben von Rotationen.

2.3 Die Grundoperationen *search*, *insert* und *delete*.

Hier werden nur die Standardvarianten eines BSTs gezeigt. Später werden Varianten gezeigt, die von diesem Verhalten zum Teil deutlich abweichen. Innerhalb von Operationen wird häufig von einem Knoten aus direkt auf dessen Elternknoten zugegriffen, so dass 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. Innerhalb eines Pfades werden in dieser Arbeit jedoch entweder nur Zeiger auf Kinder oder nur auf Elternknoten verwendet.

Es sei ein BST T gegeben. Die Operation $\text{search}(\text{key } k)$ 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 betrachteten Knotens 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.

Mit $insert(key k)$ kann eine Schlüsselmenge um den Schlüssel k erweitert werden. Bei $insert(key k)$ wird sich zunächst wie bei $search(key k)$ verhalten. Wird k gefunden, wird die Operation abgebrochen und der BST bleibt unverändert. Wird ein leerer Teilbaum T_2 erreicht, wird ein neu erzeugter Knoten mit dem 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 einem Schlüssel $> k$, bzw. in rechten Teilbäumen von Knoten mit einem Schlüssel $< k$ enthalten ist.

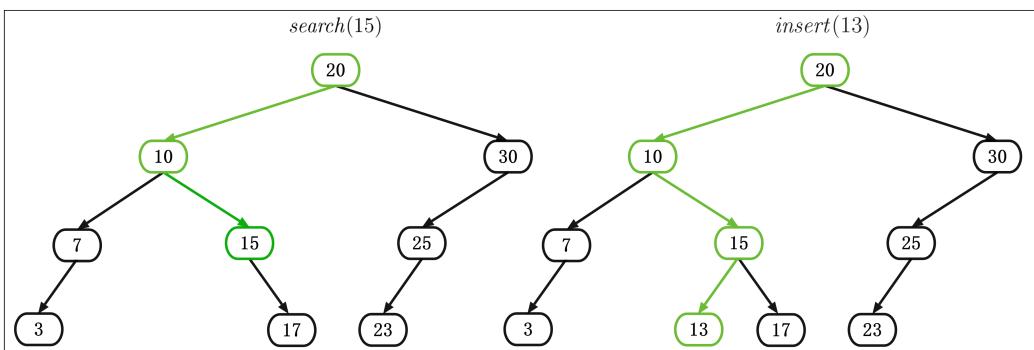


Abbildung 9: Links zeigt eine Suche nach dem Schlüssel 15. Rechts das Einfügen des Schlüssels 13.

Auch bei $delete(key k)$ wird sich zunächst wie bei $search(key k)$ verhalten. Ist k im BST nicht vorhanden, wird abgebrochen und der BST bleibt unverändert. Ansonsten werden drei Fälle unterschieden. Sei v der Knoten mit dem 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 das linke oder das rechte Kind eines Knotens 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.

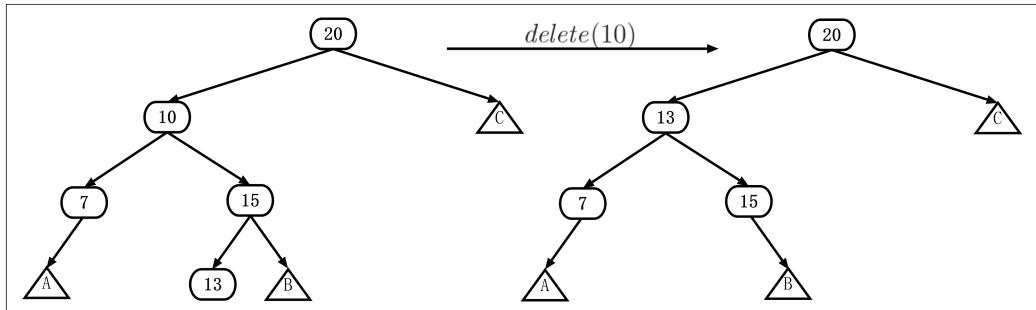


Abbildung 10: Löschen des Schlüssels 10

Laufzeit: Die 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 Einbinden 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 der 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 genau ein Blatt im BST gibt und jeder andere Knoten genau ein Kind hat. Die Baumstruktur geht in diesem Fall zu einer Listenstruktur über. Dies wird als **entartet** 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.

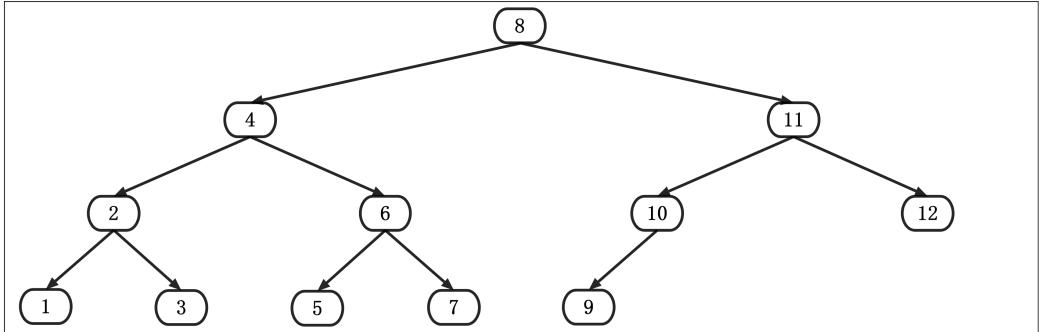


Abbildung 11: Kompletter BST mit 12 Knoten

Lemma 2.1. Die Höhe eines vollständig balancierten BSTs T mit n Knoten ist $\lfloor \log_2(n) \rfloor + 1$.

Beweis. Es sei $N(h)$ die maximale Anzahl an Knoten in einem vollständig balancierten BST mit Höhe h . $N(h)$ berechnet sich, indem die Summe der maximalen Anzahl an Knoten jeder Ebene gebildet 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:

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

□

3 Rot-Schwarz-Baum

Der Tango Baum besteht intern aus Hilfsbäumen. Der Rot-Schwarz-Baum gehört zur Gruppe der **balancierten BST** und erfüllt alle Eigenschaften, um ihn als Hilfsbaum im Tango Baum verwenden zu können. Genau das ist auch die Rolle des Rot-Schwarz-Baumes in dieser Ausarbeitung. Bei balancierten BSTs gilt für die Höhe $h = O(\log(n))$, mit n = Anzahl der Knoten. Jeder Knoten benötigt ein zusätzliches Attribut, um eine Farbinformation zu speichern. Der Name der Datenstruktur kommt daher, dass die beiden durch das zusätzliche Attribut unterschiedenen Zustände als *rot* und *schwarz* bezeichnet werden. Die Farbe ist also eine Eigenschaft der Knoten und im Folgenden wird einfach von roten, bzw. schwarzen Knoten gesprochen. Als Blätter werden schwarze Sonderknoten verwendet, deren Schlüssel auf einen Wert außerhalb des Universums, hier *null*, gesetzt wird, um sie eindeutig erkennen zu können. *null* gehört nicht zur Schlüsselmenge des Rot-Schwarz-Baumes. Fehlende Kinder von Knoten mit einem gewöhnlichen Schlüssel werden durch solche Blätter ersetzt.

Folgende zusätzliche Eigenschaften müssen bei einem Rot-Schwarz-Baum erfüllt sein.

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Jedes Blatt (Sonderknoten) ist schwarz.
4. Der Elternknoten eines roten Knotens ist schwarz.
5. Für jeden Knoten gilt, dass alle Pfade, die an ihm starten und an einem Blatt (Sonderknoten) enden, die gleiche Anzahl an schwarzen Knoten enthalten.

Sei (v_0, v_1, \dots, v_n) ein Pfad von einem Knoten v_0 zu einem Blatt v_n . Die Anzahl der schwarzen Knoten innerhalb (v_1, \dots, v_n) wird als **Schwarz-Höhe** $bh(v_0)$

von Knoten v_0 bezeichnet. Die eigene Farbe des betrachteten Knotens bleibt dabei also außen vor. Dadurch hat ein Knoten die gleiche Schwarz-Höhe wie ein rotes Kind von ihm und eine um eins erhöhte Schwarz-Höhe gegenüber einem schwarzen Kind. Die Schwarz-Höhe der Wurzel entspricht der **Schwarz-Höhe des Baumes** $bh(T)$, wobei ein leerer Baum die Schwarz-Höhe 0 hat. Die Schwarz-Höhe eines Knotens x ist genau dann eindeutig, wenn er die Eigenschaft 5 nicht verletzt. Hält x die Eigenschaft 5 ein und sei i die Anzahl schwarzer Knoten in den entsprechenden Pfaden, so gilt $bh(x) = i$, wenn x rot ist und $bh(x) = i - 1$, wenn x schwarz ist. Ist $bh(x)$ eindeutig, so enthält jeder Pfad, der mit x startet und an einem Blatt endet, $bh(x) + 1$ schwarze Knoten, wenn x schwarz ist und $bh(x)$ schwarze Knoten, wenn x rot ist. Jeder Knoten speichert seine Schwarz-Höhe als weiteres Attribut, da wir dieses in Abschnitt 5.5 benötigen. Natürlich muss das Attribut dann auch gesetzt und gepflegt werden, wobei es bei Sonderknoten fest mit 0 belegt ist. Im Folgenden wird **RBT** (Red Black Tree) als Abkürzung für Rot-Schwarz-Baum verwendet. Aufgrund der Sonderknoten gibt es eine etwas spezielle Situation bei einem RBT mit Höhe 1. Diese Konstellation ist nur mit einem einzelnen Sonderknoten erreichbar, so dass statt dessen auch einfach der leere Baum verwendet werden könnte. Auch diese Konstellation erfüllt aber die fünf Eigenschaften, so dass sie kein Problem darstellt.

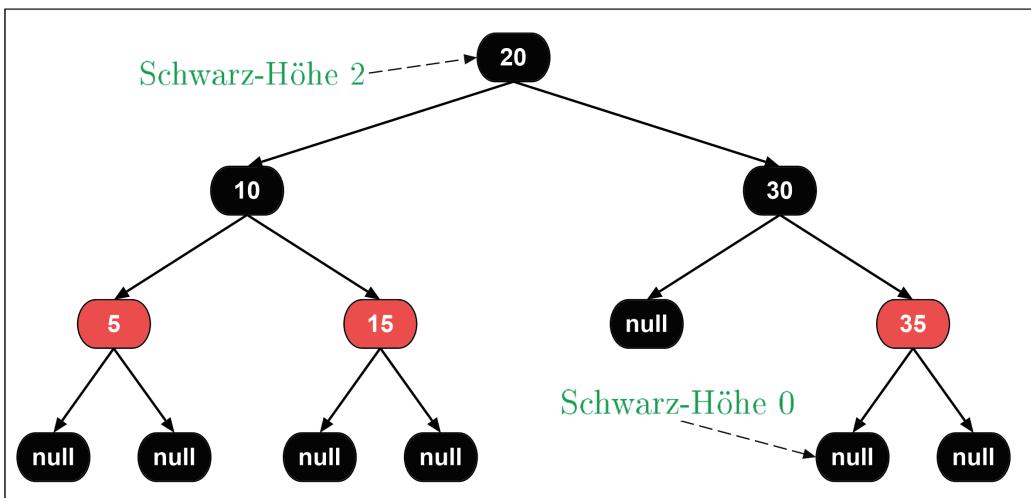


Abbildung 12: RBT ohne Verletzung von Eigenschaften.



Abbildung 13: Kein RBT, da die Eigenschaften vier und fünf verletzt sind.

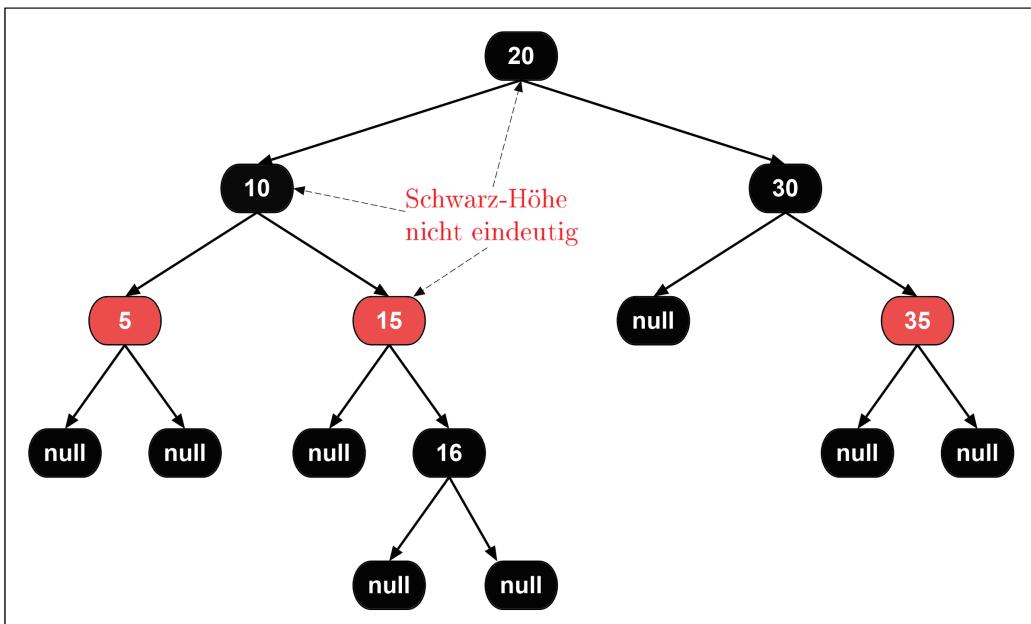


Abbildung 14: Kein RBT, da die Eigenschaft fünf verletzt ist.

3.1 Grundoperationen

Suchen im Rot-Schwarz-Baum: Die Suche unterscheidet sich nur in einem Punkt von der in Abschnitt 2.3 vorgestellten. Wird nach einem Schlüssel gesucht, der im RBT nicht vorhanden ist, so wird einer der Sonderknoten erreicht. In diesem Fall wird die Suche abgebrochen und eine leere Referenz

zurückgegeben. Die Operation verändert den RBT nicht.

Einfügen in den Rot-Schwarz-Baum: *insert* wird für einen Hilfsbaum im Tango Baum eigentlich nicht benötigt. Im Kapitel zum Tango Baum wird später jedoch eine Hilfsoperation benötigt, die am besten zu *insert* beschrieben werden kann.

Sei k der einzufügende Schlüssel. Zunächst wird wie beim Suchen vorgegangen. Wird k gefunden, wird der RBT nicht verändert. Ansonsten wird ein Sonderknoten b erreicht (Annahme der RBT ist nicht leer). Ein neu erzeugter roter Knoten v_k mit dem Schlüssel k und Schwarz-Höhe 1 nimmt den Platz von b ein. v_k werden Sonderknoten als linkes und rechtes Kind angefügt. v_k ist nun im Baum enthalten. Es muss jedoch auf mögliche Verletzungen der fünf Eigenschaften geachtet werden. Welche können betroffen sein?

1. Es ist immer noch jeder Knoten entweder rot oder schwarz.
2. Wurde in einen RBT mit leerer Schlüsselmenge eingefügt, so ist der neu erzeugte rote Knoten die Wurzel, was eine Verletzung darstellt. Waren bereits Schlüssel im Baum vorhanden blieb die Wurzel unverändert.
3. Aufgrund der Sonderknoten sind die Blätter immer noch schwarz.
4. Der Baum wird nur direkt an der Einfügestelle verändert, so dass nur v_k betrachtet werden muss. v_k hat schwarze Kindknoten. Er könnte jedoch einen roten Elternknoten haben, so dass diese Eigenschaft verletzt wäre.
5. Die Schwarz-Höhe von v_k ist korrekt gesetzt. Waren bereits Knoten im RBT vorhanden, so kann sich ihre Schwarz-Höhe nicht verändert haben, denn den Platz eines schwarzen Knotens mit Schwarz-Höhe 0 nimmt dann ein roter Knoten mit Schwarz-Höhe 1 ein. Eigenschaft fünf bleibt also erhalten.

Es können also die Eigenschaften zwei und vier betroffen sein. Jedoch nur eine von ihnen, denn Eigenschaft zwei wird genau dann verletzt, wenn der neue Knoten die Wurzel des Baumes ist. Dann kann er aber keinen roten Elternknoten haben.

Zur Korrektur wird zum Ende von *insert* eine zusätzliche Operation, *insertFixup(Node v_{in})*, aufgerufen. Diese Operation arbeitet sich von v_{in} startend, solange in einer Schleife nach oben im RBT durch, bis alle Eigenschaften wieder erfüllt sind. Die Schleifenbedingung ist, dass eine Verletzung vorliegt. Dazu muss geprüft werden, ob der betrachtete Knoten x die rote Wurzel des Gesamtbaumes ist oder ob er und sein Elternknoten beide rot sind. Vor dem

ersten Durchlauf wird $x = v_{in}$ gesetzt. Innerhalb der Schleife werden sechs Fälle unterschieden. Im Folgenden wird auf vier Fälle detailliert eingegangen. Die restlichen zwei verhalten sich symmetrisch zu einem solchen. Jeder der Fälle verantwortet, dass zum Start der nächsten Iteration wieder nur maximal eine der beiden Eigenschaften zwei oder vier verletzt sein kann und Eigenschaft vier höchstens an einem Knoten verletzt ist. Die Fallauswertung geschieht in aufsteigender Reihenfolge. Deshalb kann innerhalb einer Fallbehandlung verwendet werden, dass die vorherigen Fallbedingungen nicht erfüllt sind. Die Eigenschaft eins bleibt in der Beschreibung außen vor, da es während der gesamten Ausführungszeit der Operation nur Knoten gibt, die entweder rot oder schwarz sind.

Fall 1: x ist die rote Wurzel des RBTs: Dieser Fall wird behandelt, indem die Wurzel schwarz gefärbt wird. Es muss noch gezeigt werden, dass es durch das Umfärben zu keiner anderen Verletzung gekommen ist.

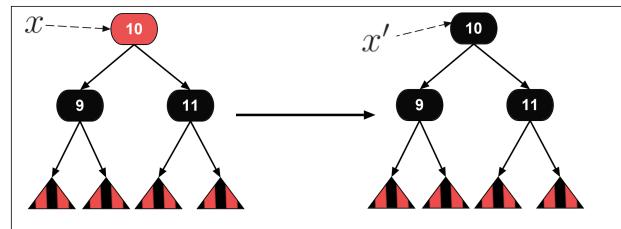


Abbildung 15: *insertFixup*. Dargestellt ist der Fall 1.

Betrachtung der Eigenschaften:

1. -
2. Die Wurzel wurde schwarz gefärbt.
3. Die Blätter (Sonderknoten) sind unverändert.
4. Es wurden weder rote Knoten hinzugefügt, noch wurde die Kantenmenge verändert.
5. Das Umfärben der Wurzel kann hierauf keinen Einfluss haben, da sie in der Berechnung der Schwarz-Höhe jedes Knotens außen vor ist.

Es wird also keine Eigenschaft mehr verletzt und die Schleife wird keine weitere Iteration durchführen.

Die Fälle 2 - 6 behandeln nun die Situationen, in denen sowohl x als auch

dessen Elternknoten y rote Knoten sind. Da Eigenschaft fünf nach jeder Iteration erfüllt ist, muss y einen Geschwisterknoten haben. Denn da zu Beginn einer Iteration nur eine Eigenschaft verletzt sein kann, kann der rote y nicht die Wurzel sein. Also muss auch y einen Elternknoten z haben. Da z kein Blatt (Sonderknoten) ist, müssen beide Kinder vorhanden sein.

Außerdem muss z schwarz sein, da die Eigenschaft vier ansonsten an zwei Knoten verletzt wäre.

Fall 2: y hat einen roten Geschwisterknoten: Diesen Fall veranschaulicht Abbildung 16. Es wird z rot gefärbt und beide Kinder von z , also y und dessen Geschwisterknoten, schwarz. Die Schwarz-Höhe von z wird um eins erhöht. Somit ist der Elternknoten von x nun schwarz und die Verletzung der Eigenschaft vier wurde an dieser Stelle behoben. Wie sieht es aber mit den Verletzungen insgesamt aus ?

Betrachtung der Eigenschaften:

1. -
2. Wenn z die Wurzel des Baumes ist, wurde sie rot gefärbt und eine Verletzung liegt vor.
3. Der rot umgefärzte Knoten z' hat zwei Kinder. Somit wurde kein Blatt rot gefärbt.
4. Wenn der rot gefärbte Knoten z' nicht die Wurzel ist, könnte er einen roten Elternknoten haben und Eigenschaft vier ist weiterhin verletzt. Das Problem liegt nun aber zwei Baumebenen höher.
5. Die Schwarz-Höhen der Vorfahren von z' bleiben unverändert, da jeder Pfad von ihnen zu einem Blatt auch entweder y' oder dessen Geschwisterknoten enthält. Die Schwarz-Höhe von z' ist um eins erhöht gegenüber derer von z , bleibt aber eindeutig. An keinem anderen Knoten ändert sich die Schwarz-Höhe.

Es kann also wieder nur entweder Eigenschaft zwei oder vier verletzt sein. Wenn das Problem noch nicht an der Wurzel ist, liegt es zumindest zwei Ebenen näher daran. x wird auf z' gesetzt.

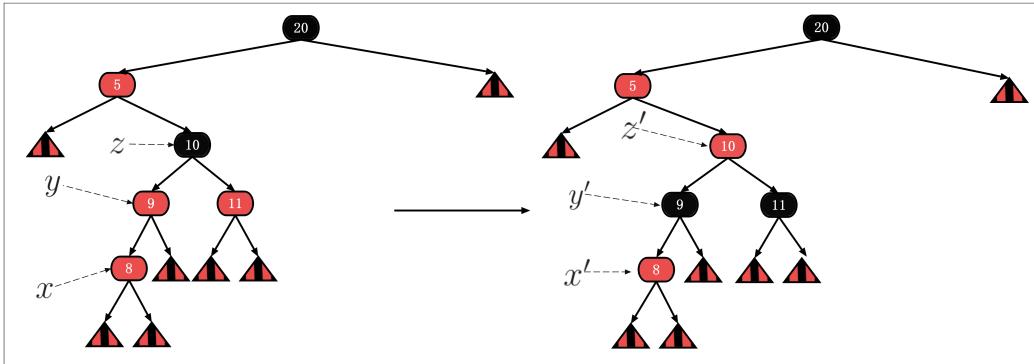


Abbildung 16: *insertFixup*. Dargestellt ist der Fall 2.

Fall 3: x ist ein linkes Kind und y ist ein linkes Kind:

Abbildung 17 zeigt eine entsprechende Situation. Es wird eine Rechtsrotation auf y ausgeführt. Anschließend wird z rot gefärbt und y schwarz.

Betrachtung der Eigenschaften:

Dazu werden vier weitere Variablen auf Knoten verwendet. Es zeigt x_l auf das linke Kind von x , x_r entsprechend auf das rechte Kind. y_r und z_r bezeichnen die rechten Kinder von y , bzw. z . Nachfolgend wird verwendet, dass die Teilbäume mit den Wurzeln x_l , x_r , y_r und z_r durch die Ausführung unverändert bleiben.

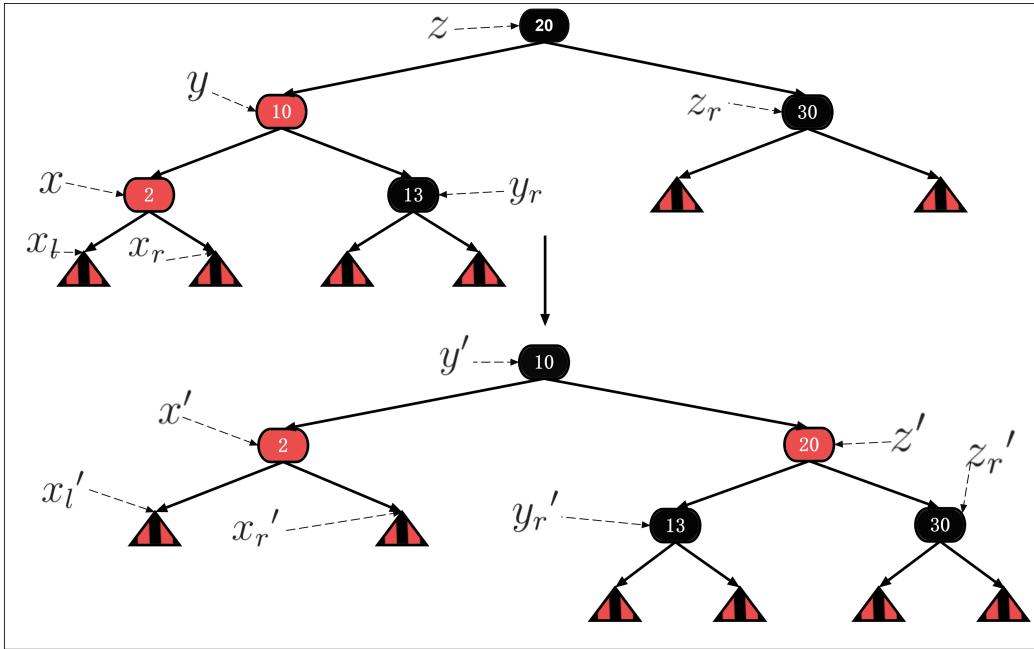


Abbildung 17: *insertFixup*. Dargestellt ist der Fall 3.

1. -
2. Wenn z zu Beginn nicht die Wurzel des Gesamtbaumes war, bleibt diese unverändert. Ansonsten wurde durch die Rotation y' zur neuen Wurzel und y' wurde schwarz gefärbt.
3. In der zweiten Ebene unter y' befinden sich ausschließlich die unveränderten Teilbäume mit den Wurzeln x'_l , x'_r , y'_r oder z'_r . An den Blättern verändert sich also durch die Ausführung nichts.
4. Knoten x' ist das linke Kind des schwarzen y' . Die Teilbäume von x' blieben unverändert. Der linke Teilbaum von y' enthält somit keine aufeinanderfolgenden roten Knoten. Das rechte Kind von y' ist der rote Knoten z' . Das rechte Kind von z' ist z'_r . z_r ist der Geschwisterknoten von y und muss aufgrund der Fallunterscheidung schwarz sein. Das linke Kind von z' ist y'_r . y_r ist das rechte Kind von y . Das rechte Kind von y muss schwarz sein, ansonsten wäre Eigenschaft vier an zwei Knoten verletzt gewesen. Die Teilbäume mit den Wurzeln z_r und y_r bleiben unverändert. Im Teilbaum mit der Wurzel y' gibt es also keine Verletzung dieser Eigenschaft. Da y' schwarz gefärbt wurde, kann auch außerhalb dieses Teilbaumes durch y' keine neue Verletzung entstanden sein.

5. Es gilt $bh(x_l) = bh(x_r) = bh(y_r) = bh(z_r) = bh(z) - 1$. Wie oben bereits erwähnt wird die zweite Ebene unter der Wurzel y' von den unveränderten Teilbäumen mit den Wurzeln x'_l, x'_r, y'_r und z'_r gebildet. Es müssen also lediglich die Knoten x', y' und z' betrachtet werden. Die Kinder von x' und z' sind schwarze Knoten mit der Schwarz-Höhe $bh(z) - 1$. Die Schwarz-Höhen von x' und z' sind also eindeutig und es gilt $bh(x') = bh(z') = bh(z)$. Die Kinder von y' sind die roten Knoten x' und z' . Da beide Kinder rot sind gilt auch $bh(y') = bh(z)$. Somit sind alle Schwarz-Höhen im betrachteten Teilbaum eindeutig. y' hat die gleiche Schwarz-Höhe und die gleiche Farbe wie z . Damit kann es auch im Gesamtbaum zu keiner Verletzung der Eigenschaft gekommen sein.

Es ist keine der Eigenschaften verletzt. Daher wird es zu keiner Iteration mehr kommen.

Fall 4: x ist ein rechtes Kind und y ist ein linkes Kind:

Dieser in Abbildung 18 gezeigte Fall wird so umgeformt, dass eine Situation entsteht bei der Fall drei angewendet werden kann. Dazu wird eine Linksrotation an Knoten x durchgeführt.



Abbildung 18: *insertFixup*. Dargestellt ist der Fall 4.

Betrachtung der Eigenschaften:

Zu Veränderungen kommt es durch die Rotation lediglich im linken Teilbaum von z . Es sei x_l das linke Kind von x , und x_r das rechte Kind von x . y_l ist das linke Kind von y . Die Knoten z , x_l , x_r und y_l müssen schwarz sein, ansonsten wäre Eigenschaft vier mehrfach verletzt gewesen.

1. -
2. Die Wurzel bleibt unverändert.
3. Die Teilbäume mit den Wurzeln x_l , x_r und y_l enthalten alle Blätter innerhalb des linken Teilbaumes von z . Diese Teilbäume bleiben durch die Rotation unverändert und x'_l , x'_r und y'_l enthalten auch alle Blätter des linken Teilbaumes von z' .
4. Nach der Rotation ist y' das linke Kind von x' . x' ist ein Kind vom schwarzen Knoten z' . Die weiteren Kinder der Knoten x' und y' sind x'_l , x'_r und y'_l . Diese müssen schwarz sein, da kein Knoten umgefärbt

wurde. Durch die Rotation verbleibt es also bei einer Verletzung der Eigenschaft vier in der gleichen Baumebene. Die beiden beteiligten roten Knoten sind nun aber jeweils linke Kinder.

5. $bh(y_l) = bh(x_l) = bh(x_r) = bh(y'_l) = bh(x'_l) = bh(x'_r)$. Die Schwarz-Höhen von x und y bleiben deshalb unverändert. Damit kommt es auch bei z zu keiner Veränderung der Schwarz-Höhe.

Es sind also weiterhin zwei aufeinanderfolgende rote Knoten in den gleichen Baumebenen vorhanden. Diese sind nun aber beides linke Kinder. Der Geschwisterknoten des oberen der beiden roten Knoten ist der selbe schwarze Knoten wie vor der Ausführung von Fall 4. Damit kann direkt mit dem Bearbeiten von Fall 3 begonnen werden. Es benötigt keine weitere Iteration.

Fall 5: x ist ein rechtes Kind und y ist ein rechtes Kind:

Links-rechts-symmetrisch zu Fall 3

Fall 6: x ist ein linkes Kind und y ist ein rechtes Kind:

Links-rechts-symmetrisch zu Fall 4

Laufzeit: Sei h die Höhe des Gesamtbaumes vor Aufruf von *insertFixup*. Fall 2 kann maximal $\lfloor h/2 \rfloor$ mal ausgewählt werden, bevor x oder y auf die Wurzel zeigen muss. Nach einer Iteration bei der nicht Fall 2 ausgewählt wird, terminiert *insertFixup*. Der Aufwand innerhalb jeder Fallbehandlung ist $O(1)$. Für die Gesamlaufzeit gilt deshalb $O(h)$.

Löschen aus dem Rot-Schwarz-Baum: Die Reparatur des RBT nach dem Entfernen eines Knotens ist aufwendiger, als die nach dem Einfügen. Da der RBT in der Rolle als Hilfsbaum im Tango Baum keine solche Operation benötigt, entfällt diese Beschreibung. In [3] ist eine detaillierte Beschreibung enthalten.

Laufzeit der Grundoperationen: Zu Beginn des Kapitels wurde erwähnt, dass für die Höhe h eines RBTs mit n Knoten $h = O(\log(n))$ gilt. Das wird nun gezeigt:

Lemma 3.1. *Für die Höhe h eines RBTs T mit n Knoten gilt $h = O(\log(n))$.*

Beweis. Sei w die Wurzel von T und m die Anzahl der inneren Knoten von T . Zunächst wird gezeigt, dass T mindestens $2^{bh(w)} - 1$ innere Knoten enthält. Dies geschieht mit Induktion über h . Für $h = 0$ und $h = 1$ mit $2^0 - 1 = 0$ stimmt die Behauptung, denn der Baum ist leer oder enthält lediglich einen

einzelnen Sonderknoten.

Induktionsschritt:

Sei T_l der linke Teilbaum von w und T_r der rechte Teilbaum von w . Im Induktionsschritt kann nun verwendet werden, dass $h > 1$ gilt und w ein innerer Knoten sein muss. T_l und T_r haben die Schwarz-Höhe $bh(w) - 1$, wenn ihre Wurzel schwarz ist und Schwarz-Höhe $bh(w)$, wenn ihre Wurzel rot ist. Ihre Höhe ist kleiner als h und somit enthalten sie nach der Induktionsnahme mindestens $2^{bh(w)-1} - 1$ innere Knoten. Zusammenfassen ergibt die Behauptung:

$$m \geq 2^{bh(w)-1} - 1 + 1 + 2^{bh(w)-1} - 1 = 2^{bh(w)} - 1$$

Daraus folgt $\log_2(m+1) \geq bh(w)$.

Es gilt folgender Zusammenhang, da höchstens jeder zweite Knoten in einem Pfad rot sein kann.

$$\begin{aligned} h(w) &\leq 2 \cdot bh(w) + 1 \\ \Rightarrow \frac{h(w) - 1}{2} &\leq bh(w) \end{aligned}$$

Einsetzen liefert:

$$\begin{aligned} \log_2(m+1) &\geq \frac{h(w) - 1}{2} \\ \Rightarrow 2 \cdot \log_2(m+1) + 1 &\geq h(w) \\ \Rightarrow h(w) &= O(\log(m)) \end{aligned}$$

Es kann nur maximal doppelt so viele Blätter wie innere Knoten geben:

$$\begin{aligned} n &\leq 3m \\ \Rightarrow h(w) &= O(\log(n)) \end{aligned}$$

□

search und *insert* haben also die Laufzeit $O(\log(n))$.

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 . Wird der Parameter von $search$ auf $k \in K$ beschränkt, wird die Operation als *access* bezeichnet. Ein BST der seine Struktur während einer solchen Operation verändern kann, wird als **dynamisch** bezeichnet. Der RBT ist also kein dynamischer BST. Der Tango Baum ist dynamisch, wie wir noch sehen werden. 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, \dots, x_m wird also zunächst $access(x_1)$ ausgeführt, dann $access(x_2)$ usw.. Dabei ist m die Länge von X . Bei BSTs wird bezüglich Zugriffsfolgen zwischen online und offline Varianten unterschieden. Bei **offline BSTs** ist die Zugriffsfolge zu Beginn bereits bekannt, somit kann ein Startzustand gewählt werden, der die Kosten minimiert. Bei **online BSTs** ist die Zugriffsfolge zu Beginn nicht bekannt. Bei einer worst case Laufzeitanalyse muss somit von einem Startzustand ausgegangen werden, bei dem die Kosten am höchsten sind. In dieser Arbeit werden *access* Operation betrachtet die folgende Eigenschaften besitzen:

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 die Operation muss p auf den Knoten mit dem Schlüssel k zeigen.
2. Die Operation 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 wird $n = |K|$ gesetzt. Außerdem werden pro Knoten als Hilfsdaten nur konstant viele Konstanten und Variablen zugelassen, die jeweils eine Größenordnung von $O(\log(n))$ 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 durchgeföhrten Einzelschritte während der Ausführung einer Zugriffsfolge X mit Länge m . Dann berechnen sich die Gesamtkosten zum Ausführen von X mit $a + m$. Es muss zu X zumindest einen offline BST geben, so dass die Gesamtkosten keines anderen niedriger sind. Diese Kosten werden als $OPT(X)$

bezeichnet.

Culik und Wood [4] zeigten, dass der Zustand eines BSTs mit maximal $2n - 2$ Rotationen in jeden anderen BST Zustand 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 Zugriffsfolge X in $O(OPT(X))$ Zeit ausführen kann. Ein BST der X 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 Die erste untere Schranke von Wilber.

Robert Wilber hat zwei Methoden zur Berechnung unterer Schranken für die Laufzeit zur Ausführung von Zugriffsfolgen bei BSTs vorgestellt [5]. Hier wird auf die Erste davon eingegangen. Im Folgenden werden BSTs betrachtet bei denen während einer $access(k)$ Operation, der Knoten mit dem Schlüssel k durch Rotationen zur Wurzel des BSTs wird. Ein solcher BST wird als **standard BST** bezeichnet. Asymptotisch betrachtet entsteht hierdurch kein Verlust der Allgemeinheit. Denn 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 von $d + 1$ 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 standard BST T und einer Zugriffsfolge X notieren wir die minimalen Kosten, die zum Ausführen von X entstehen mit $W(X, T)$. Im Folgenden wird angenommen, dass $K = \{i \in \mathbb{N} | i \in [j, k] \text{ mit } j, k \in \mathbb{N}\}$ gilt. Dadurch entsteht kein Verlust der Allgemeinheit, denn anderenfalls könnte die Schlüsselmenge einfach aufsteigend sortiert mit j startend durchnummieriert werden. 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 Elternknotens von v , vor Ausführung der Rotation. Aus einer Folge von Rotationen $R = (i_1, j_1), (i_2, j_2), \dots, (i_m, j_m)$ wird die Folge $R_x^y = (i_{1'}, j_{1'}), (i_{2'}, j_{2'}), \dots, (i_{m'}, j_{m'})$ gebildet, indem aus R jede Rotation entfernt wird, bei der $i \notin [x, y] \vee j \notin [x, y]$ gilt. Ähnlich wird aus X die Zugriffsfolge X_x^y gebildet, indem 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 $|K| - 1$ inneren Knoten enthalten die Schlüssel aus der Menge $\{r \in R \mid \exists i, j \in K : (i + 1 = j \wedge r = i + 0, 5)\}$. Y kann immer erstellt werden, indem zunächst ein BST Y_i mit den inneren Knoten von Y erzeugt wird. Die Blätter werden dann an den Positionen angefügt, an der die Standardvariante von *insert* angewendet auf Y_i ihren Schlüssel einfügen würde. Dass hierbei für zwei verschiedene Blätter mit den Schlüsseln k_1, k_2 die gleiche Position gewählt wird ist ausgeschlossen, da es einen inneren Knoten mit einem 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 BSTs mehrere mögliche Lower Bound Trees existieren. Abbildung 19 zeigt eine beispielhafte Konstellation.

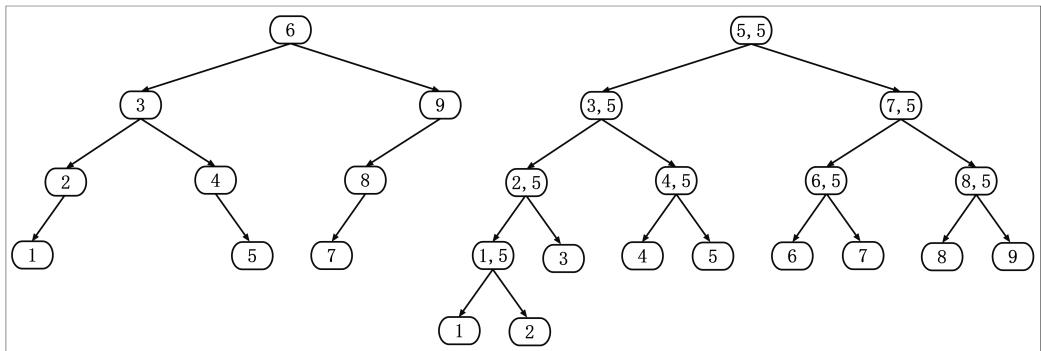


Abbildung 19: Rechts ist ein möglicher Lower Bound Tree zum linken BST dargestellt.

Nun wird die Funktion $x(T, Y, X)$ 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 inneren Knoten von Y und m die Länge von X . Sei $u \in U$ und l der kleinste Schlüssel eines Blattes im Teilbaum mit der Wurzel u , sowie r der größte Schlüssel eines solchen Blattes. Sei v der gemeinsame Vorfahre der Knoten mit einem Schlüssel aus $[l, r]$ in T mit der größten Tiefe. Sei o die Folge $o_0, o_1, \dots, o_m = \text{key}(v) \circ X_l^r$. $i \in [1, m]$ ist eine **u-Transition**, wenn $(o_{i-1} < u \wedge o_i > u) \vee (o_{i-1} > u \wedge o_i < u)$ gilt. Mit Hilfe der Funktion $\text{score}(X, u) = |\{i \in \mathbb{N} \mid i \text{ ist eine } u\text{-Transition bezogen auf } X\}|$ kann nun $x(T, Y, X)$ definiert werden:

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

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, \dots, v_n) eines Knotens v ist der längst mögliche Pfad für den gilt: v_0 ist das linke Kind von v und für $i \in \{1, 2, \dots, n\}$, v_i ist das rechte Kind von v_{i-1} . Der **rechte innere Pfad** (v_0, v_1, \dots, v_n) eines Knotens v ist der längst mögliche Pfad für den gilt: v_0 ist das rechte Kind von v 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 gemeinsame Vorfahre der Knoten mit einem Schlüssel aus $[l, r]$ in T mit der größten Tiefe. (Existiert ein solcher nicht, ist T_l^r der leere Baum). Es muss $key(v_d) \in [l, r]$ gelten. Denn falls v_d ein Blatt ist, ist sein Schlüssel der Einzige aus $[l, r]$. Hat v_d genau ein Kind v_c und $key(v_d) \notin [l, r]$, dann wäre v_c ein gemeinsamer Vorfahre der entsprechenden Knoten, mit größerer Tiefe. Hat v_d zwei Kinder gibt es drei Fälle:

- Im linken und im 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 $key(v_d) \notin [l, r]$, dann wäre v_c ein gemeinsamer Vorfahre der entsprechenden Knoten, mit größerer Tiefe.
- 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 dem 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 $key(w) = key(v)$.
3. Existiert das linke Kind von v nicht, weiter bei Punkt 4. Ansonsten sei P_l der linke innere Pfad von v . Enthält P_l keinen Knoten mit einem

Schlüssel $\geq l$, weiter bei Punkt 4. Ansonsten sei k_l der Schlüssel des Knotens mit der kleinsten Tiefe in P_l , für den gilt $k_l \geq l$. Erzeuge einen Knoten w_l mit dem Schlüssel k_l und füge ihn als das linke Kind an w an. Füge w_l zu W hinzu.

4. Existiert das rechte Kind von v nicht, weiter bei Punkt 5. Ansonsten sei P_r der rechte innere Pfad von v . Enthält P_r keinen Knoten mit einem Schlüssel $\leq r$, weiter bei Punkt 5. Ansonsten sei k_r der Schlüssel des Knotens mit der kleinsten Tiefe in P_r , für den gilt $k_r \leq r$. Erzeuge einen Knoten w_r mit dem Schlüssel k_r und füge ihn als das rechte Kind an w an. Füge w_r zu W hinzu.
5. Füge w zu U hinzu, weiter bei Punkt 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 20 dar.

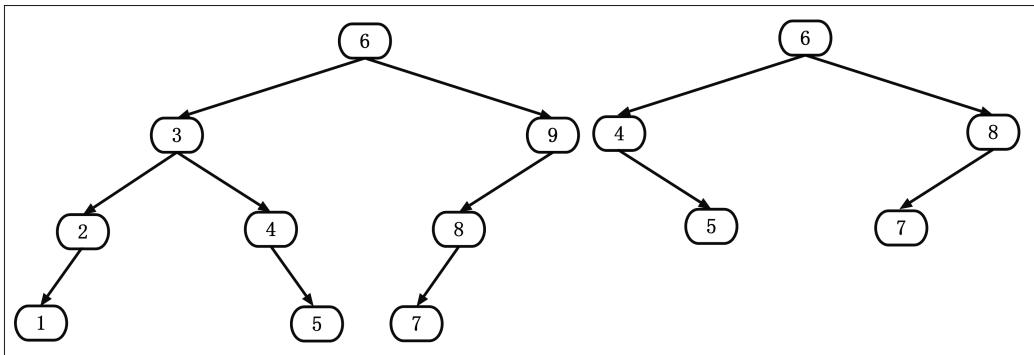


Abbildung 20: Links ein BST T , rechts ein davon abgeleiteter BST T_4^8 .

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.

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

$K_l^r \subseteq K_2$:

Sei $k \in K_l^r$ und v_k der Knoten in T mit $key(v_k) = k$. Es muss einen Pfad $P_T = (v_0, \dots, v_n)$ in T geben, mit $v_0 = v_d$ und $v_n = v_k$. Sei m die Anzahl der Knoten in P_T , mit einem Schlüssel aus $[l, r]$. Nun folgt Induktion über m .

Für $m = 1$ gilt $k = key(v_d)$ und $k \in K_2$.

Induktionsschritt:

Sei w der Index des Knotens mit der größten Tiefe in v_0, \dots, v_{n-1} , mit

$\text{key}(v_w) \in K_2$. Nach Induktionsannahme gibt es einen Knoten u_w mit $\text{key}(u_w) = \text{key}(v_w)$ in T_l^r . Es sei $\text{key}(v_w) > \text{key}(v_k)$, der andere Fall ist symmetrisch. Ist v_k das linke Kind von v_w , dann enthält das linke Kind von u_w den Schlüssel $\text{key}(v_k)$. Andernfalls gilt für alle v_j , mit $w < j < k$, $\text{key}(v_j) < l \leq \text{key}(v_k)$. 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 $\text{key}(u_k) = \text{key}(v_k)$ das linke Kind von u_w .

Nun kommen wir zum Lemma:

Sei v ein Knoten in T . Ein Knoten in T_l^r mit dem Schlüssel $\text{key}(v)$ wird dann 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 $(\text{key}(u), \text{key}(v))$ aus T entsteht. Gilt $\text{key}(u), \text{key}(v) \in [l, r]$, dann ist T'^r_l der BST, der aus T_l^r durch Ausführen von $(\text{key}(u), \text{key}(v))$ entsteht. Andernfalls gilt $T'^r_l = T_l^r$.*

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

1. u ist das linke Kind von v und $\text{key}(u) < l$:

Sei w^* ein Knoten aus T_l^r und w'^* der aus T'^r_l , mit $\text{key}(w^*) = \text{key}(w'^*)$. Es muss gezeigt werden, dass wenn w^* ein linkes, bzw. rechtes Kind mit dem Schlüssel k hat, dann gilt dies auch für w'^* . Da $\text{key}(u) < l \leq \text{key}(w)$ gilt, kann weder u noch v im rechten Teilbaum von w enthalten sein. Somit ist bezüglich des rechten Kindes nichts zu zeigen. Sei P_l der linke innere Pfad von w und 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 7 und da $\text{key}(u) < l$, haben die linken Kinder von w^* und w'^* den gleichen Schlüssel. Nun sei v in P_l enthalten. Dann unterscheiden sich P_l und P'_l dadurch, dass ein Knoten mit $\text{key}(u)$ 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 rechte Kind von v und $\text{key}(u) > r$:

Links-rechts-symmetrisch zu Fall 1.

3. u ist das linke Kind von v und $\text{key}(v) > r$:

Durch Vertauschen der Bezeichnungen von v und u entsteht aus T' von

Fall 2 durch Ausführung der Rotation auf dieser Konstellation wieder T aus Fall 2. Somit muss nichts weiter gezeigt werden.

4. u ist das rechte Kind von v und $\text{key}(v) < l$:
Links-rechts-symmetrisch zu Fall 3.

Übrig bleibt noch die Konstellation $\text{key}(u), \text{key}(v) \in [l, r]$. Betrachtet wird eine Rechtsrotation $(\text{key}(u), \text{key}(v))$. Die Linksrotation ist wieder symmetrisch. $T_l^{r'}$ entsteht durch das Ausführen der Rotation $(\text{key}(u), \text{key}(v))$ auf T_l^r . Zu zeigen ist $T_l^{r'} = T_l^r$.

Es werden zuerst die möglichen Veränderungen an inneren Pfaden von T betrachtet.

1. Sei u_r das rechte Kind von u . Sei $(u, u_r, v_1, v_2, \dots, v_n)$ der linke innere Pfad von v , dann ist $(u_r', v_1', v_2', \dots, v_n')$ der linke innere Pfad von v' . Es gilt $l \leq \text{key}(u) < \text{key}(u_r) < \text{key}(v) \leq r$. Damit ist $u_r'^*$ das linke Kind von v'^* .
2. Sei (v_1, v_2, \dots, v_n) der rechte innere Pfad von u , dann ist $(v', v_1', v_2', \dots, v_n')$ der rechte innere Pfad von v' . Damit ist v'^* das rechte Kind von u'^* .
3. Ist v das linke Kind eines Knotens z mit $\text{key}(z) \in [r, l]$, dann sei $(v, v_1, v_2, \dots, v_n)$ der linke innere Pfad von z . Dann ist $(u', v', v_1', v_2', \dots, v_n')$ der linke innere Pfad von z' . Damit ist u'^* das linke Kind von z'^* .
Ist v das rechte Kind eines Knotens z mit $\text{key}(z) \in [r, l]$, dann sei $v, u, v_1, v_2, \dots, v_n$ der rechte innere Pfad von z . Dann ist $(u', v_1', v_2', \dots, v_n')$ der rechte innere Pfad von z' . Damit ist u'^* das rechte Kind von z'^* .
4. Sei nun v das Kind eines Knotens z mit $\text{key}(z) \notin [r, l]$. Sei a ein Knoten mit $\text{key}(a) \in [r, l]$. Als erstes seien z und v im linken inneren Pfad von a enthalten. Dann ist u'^* das linke Kind von a'^* .
Nun seien z und v im rechten inneren Pfad von a enthalten. Dann ist u'^* das rechte Kind von a'^* .

Nun wird auf T_l^r die Rotation $(\text{key}(u^*), \text{key}(v^*))$ ausgeführt. $u_r'^*$ ist das linke Kind von v'^* . v'^* ist das rechte Kind von u'^* . Ist v^* das linke, bzw. rechte Kind eines Knotens b^* , dann ist u'^* das linke, bzw. rechte Kind von b'^* . Damit gilt $T_l^{r'} = T_l^r$.

□

Satz 4.1. Es sei T_0 ein standard BST mit der Schlüsselmenge $K = \{i \in \mathbb{N} \mid i \in [j, k] \text{ mit } j, k \in \mathbb{N}\}$. Sei Y ein für T_0 erstellter Lower Bound Tree und X eine zu T_0 erstellte Zugriffsfolge mit Länge m . Dann gilt $W(X, T_0) \geq X(T_0, Y, X)$.

Beweis. Sei U die Menge der inneren Knoten von Y . Die Kosten zum Ausführen von X sind die Anzahl der Einzelschritte + m . Es reicht also aus zu zeigen, dass mindestens $\sum_{u \in U} \text{score}(X, u)$ Rotationen benötigt werden. Es wird Induktion über $n = |K|$ angewendet. Sei $n = 1$, dann gibt es keinen inneren Knoten in Y und $\sum_{u \in U} \text{score}(X, u) = 0$. Der Induktionsanfang ist somit gemacht. Im Folgenden sei $n \geq 2$.

Sei $R = r_1, r_2, \dots, r_l$ die Folge der insgesamt durchgeföhrten Rotationen. Für $i \in \{1, \dots, l\}$ sei T_i der BST, der entsteht, nachdem r_i auf T_{i-1} ausgeführt wurde. Sei w die Wurzel von Y , mit dem 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 . Aus $n \geq 2$ folgt, dass w ein innerer Knoten ist. Sei $R^1 = r_1^1, r_2^1, \dots, r_{l1}^1 = R_1^{k_w}$ und $R^2 = r_1^2, r_2^2, \dots, r_{l2}^2 = R_{k_w}^\infty$. 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 entsteht durch das Entfernen aller Schlüssel $k > k_w$ aus X und X^2 entsteht durch das Entfernen aller Schlüssel $k < k_w$ aus X . Für $j \in \{1, 2\}$ sei U^j die Menge der inneren Knoten von Y^j . Sei $T_0^{j*}, T_1^{j*}, \dots, T_{l_j}^{j*}$ die entstehende Folge, wenn aus $T_0^j, T_1^j, \dots, T_l^j$ die T_i^j entfernt werden, für die $T_{i-1}^j = T_i^j$ gilt.

Mit Lemma 4.1 kann für $t \in \{1, 2, \dots, l^1\}$, T_t^{1*} durch Ausführung der Rotation r_t^1 auf T_{t-1}^{1*} abgeleitet werden. Für $t \in \{1, 2, \dots, l^2\}$ gilt analog das Gleiche. Dadurch folgt durch dieses Lemma, falls ein Knoten mit einem Schlüssel $k < w$, bzw. $k > w$ die Wurzel von T_i ist, dann muss die Wurzel von T_i^1 , bzw. T_i^2 auch den 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 Induktionsannahme $l^j \geq \sum_{u \in U^j} \text{score}(X^j, u)$.

Sei $\sigma = \text{key}(w) \circ X$. Sei a eine w -Transition. Nun wird angenommen, dass $\sigma_{a-1} < \text{key}(w) \wedge \sigma_a > \text{key}(w)$ gilt. Der andere Fall kann davon problemlos abgeleitet werden. Sei y der Knoten in T mit $\text{key}(y) = \sigma_{a-1}$ und z der Knoten in T mit $\text{key}(z) = \sigma_a$. Nach $\text{access}(\sigma_{a-1})$ ist y die Wurzel von T . z muss sich im rechten Teilbaum von y befinden. Nach $\text{access}(\sigma_a)$ ist z die Wurzel von T . y muss sich im linken Teilbaum von z befinden. Somit muss während $\text{access}(\sigma_a)$ die Rotation $(\text{key}(z), \text{key}(y))$ ausgeführt worden sein. $(\text{key}(z), \text{key}(y))$ muss

somit in M enthalten sein. Für jede w -Transition ist also mindestens eine Rotation in M enthalten. Also gilt $l_M \geq score(X, w)$.

Zusammengefasst ergibt sich:

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

□

$x(T, Y, X)$ ist also eine untere Schranke für die Ausführungszeit von standard BSTs und damit asymptotisch auch für allgemeine BSTs.

4.3 Bit Reversal Permutation

In diesem Abschnitt wird gezeigt, dass es Zugriffsfolgen mit der Länge n gibt, die insgesamt n voneinander verschiedene Elemente enthalten, so dass für die Laufzeit zum Ausführen bei einem beliebigen BST T $\Omega(n \log(n))$ gilt. Hier werden speziell die Zugriffsfolgen betrachtet, die als **Bit Reversal Permutations** bezeichnet werden. Dafür wird die erste untere Schranke von Wilber verwendet und ein Beweis ist ebenfalls in [5] enthalten.

Nun wird zunächst auf den Aufbau einer solchen Zugriffsfolge eingegangen. Sei $l \in \mathbb{N}$ und $i \in \{0, 1, \dots, l-1\}$. Eine Folge $b_{l-1}, b_{l-2}, \dots, b_0$ mit $b_i \in \{0, 1\}$, kann als Zahl zur Basis 2 interpretiert werden. T enthält alle Schlüssel die als eine solche Folge dargestellt werden können. Die Schlüsselmenge von T ist deshalb $K_l = \{0, 1, \dots, 2^l - 1\}$. Die Funktion $br_l(k): K \rightarrow K$ ist wie folgt definiert. Sei $b_{l-1}, b_{l-2}, \dots, b_0$ 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 X_l zu l ist die Zugriffsfolge $br_l(0), br_l(1), \dots, br_l(2^l - 1)$. Tabelle 1 zeigt die Bit Reversal Permutation mit $l = 4$.

Sei $y = \max(K_l)/2 = 2^{l-1} - 0,5$. Da b_0 in den Binärdarstellungen zu $0, 1, \dots, 2^l - 1$ alterniert, alterniert b_{l-1} in X_l . 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 $|K_l| = 2^l$, sind im vollständig balancierten Lower Bound Tree Y zu T alle Ebenen vollständig mit Knoten besetzt. Sei w die Wurzel von Y . Da im linken Teilbaum von w genau so viele Blätter wie im rechten Teilbaum vorhanden sein müssen, kann nur y der Schlüssel von w sein. Zu einer Zugriffsfolge $X = x_0, x_1, \dots, x_m$

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: Die Bit Reversal Permutation für $l = 4$

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 $x_0 + i, x_1 + i, \dots, x_m + i$.

Korollar 4.1. Sei $l \in \mathbb{N}$. Sei T ein standard BST mit der Schlüsselmenge $K_l = \{0, 1, \dots, 2^l - 1\}$ und $n = 2^l$. Sei $X = x_0, x_1, \dots, x_{n-1}$ 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 inneren Knoten von Y . Mit Satz 4.1 reicht es aus,

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

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

$$\sum_{u \in U} score(X, u) = 0 = n \log_2(n) + 1 - n.$$

Nun sei $l > 0$. Sei w die Wurzel von Y , mit $k_w = key(w)$. Sei $T_0^{k_w}$ ein BST mit

der 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 der 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 inneren 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 $X_{k_w}^\infty = X_0^{k_w} + 2^{l-1}$. Nun wird $X_0^{k_w}$ als X^1 bezeichnet und $X_{k_w}^\infty$ als X^2 . Mit der Induktionsannahme gilt deshalb für $i \in \{1, 2\}$:

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

Sei $j \in \{1, 2, \dots, n-1\}$. Aus $(x_j < k_w \Rightarrow x_{j-1} > k_w) \wedge (x_j > k_w \Rightarrow x_{j-1} < k_w)$ folgt $\text{score}(X, w) \geq n-1$.

Zusammenfassen ergibt:

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

□

Die Schlüsselmenge wurde beim Korollar auf $K_l = \{0, 1, \dots, 2^l - 1\}$ festgelegt. Vielleicht wäre es aber mit einer anderen Schlüsselmenge K möglich X schneller auszuführen? In jedem Fall müsste $K_l \subseteq K$ gelten. Sei R die Folge von Rotationen, die beim Ausführen von X bei einem BST T mit der Schlüsselmenge K entsteht. Sei $y = 2^l - 1$. Mit Lemma 4.1 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 Abschnitt werden die Kosten von amortisierten Laufzeitanalysen verwendet. Deshalb wird diese hier nun vorgestellt. 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. Sei $i \in \{0, \dots, m\}$. Die **tatsächlichen Kosten** t_i stehen für die exakt bestimmten Kosten zum Ausführen der i -ten Operation. Durch das Bilden

der Summe der tatsächlichen Kosten jeder einzelnen Operation ergeben sich die **tatsächlichen Gesamtkosten**. Stehen für die Laufzeit der Operationen jeweils nur obere Schranken zur Verfügung, kann mit diesen genau so vorgegangen werden, um eine obere Schranke für die Gesamtlaufzeit zu erhalten. So ermittelte 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 BSTs wird in der Regel die **Potentialfunktionsmethode** verwendet.

Potentialfunktionsmethode: 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 der 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, wird die Summe der amortisierten Kosten aller Operationen gebildet.

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

Folgendes gilt für die Summe der t_i :

$$\begin{aligned} \sum_{i=1}^m t_i &= \sum_{i=1}^m (a_i - \Phi(D_i) + \Phi(D_{i-1})) = \Phi(D_0) - \Phi(D_m) + \sum_{i=1}^m a_i \\ &\Rightarrow \left(\Phi(D_m) \geq \Phi(D_0) \Rightarrow \sum_{i=1}^m a_i \geq \sum_{i=1}^m t_i \right) \end{aligned}$$

Ist das Potenzial nach der 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 $\Phi(D_m) \geq \Phi(D_0)$ gilt. Dies wird jetzt noch an einem einfachen Beispiel demonstriert.

Potentialfunktionsmethode 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 Elements. 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 können. *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 Gesamlaufzeit einer Folge von m Operationen kann $O(mn)$ angegeben werden. Mit einer amortisierten Analyse wird nun aber $O(m)$ für einen zu Beginn leeren Stack gezeigt. Als Φ verwenden wir eine Funktion, welche die aktuelle Anzahl der im Stack enthaltenen Elemente zurück gibt. *push* erhöht 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. In jedem Fall gilt $\Phi_m \geq \Phi_0 = 0$. Für die Ausführungszeit der Folge gilt deshalb $O(m)$. Das Potential kann aufgrund einer Folge von Operationen um maximal n verringert werden. Deshalb kann $O(m + n)$ verwendet werden, wenn der Stack zu Beginn nicht leer ist.

4.5 Eigenschaften eines dynamisch optimalen BSTs.

Hier werden einige obere Laufzeitschranken für Zugriffsfolgen vorgestellt. Es ist bekannt, dass es obere Schranken sind, da mit dem Splay Baum ein BST bekannt ist, der jede der Schranken einhält. Der Splay Baum wird später noch vorgestellt. Es wird wieder ohne Verlust der Allgemeinheit eine Schlüsselmenge $K = \{1, 2, \dots, n\}$ angenommen. Wenn nicht anders angegeben wird $X = x_1, x_2, \dots, x_m$ als Zugriffsfolge verwendet.

Balanced Property: Ein BST erfüllt die Balanced Property, wenn er X in amortisiert $O(m + (m + n) \log(n))$ Zeit ausführt.

Static Finger Property: Die Idee hinter dieser Eigenschaft ist, dass es einfacher ist, Zugriffsfolgen schnell auszuführen, wenn ihre Schlüssel betragsmäßig nahe beieinander liegen. Ein BST erfüllt die Static Finger Property,

wenn zu jedem Schlüssel $k_f \in K$ für die amortisierte Laufzeit zum Ausführen von X

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

gilt.

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

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

Zeit ausführt. Der Name kommt daher, dass es sich hierbei um eine untere Schranke für die Ausführungszeit von X bei statischen BST handelt [6]. Solche ändern ihre Struktur während *access* nicht.

Working Set Property: Ein BST mit der 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 $t_{xi} = \max(J_i)$, falls J_i nicht leer ist, ansonsten $t_{xi} = 0$. t_{xi} liefert also den Index des vorherigen Zugriffes auf x_i , falls ein solcher existiert. Sei $a_i = |\{x_j | t_{xi} < j \leq i\}|$. Ein BST erfüllt die Working Set Property, wenn für seine amortisierte Laufzeit zum Ausführen von X

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

gilt.

Dynamic Finger Property: Diese Eigenschaft ist der Static Finger Property sehr ähnlich. Ein BST erfüllt die Dynamic Finger Property, wenn für seine amortisierte Laufzeit zum Ausführen von X

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

gilt.

Abbildung 21 zeigt Implikationen zwischen den Eigenschaften und basiert auf einer Abbildung aus [7].

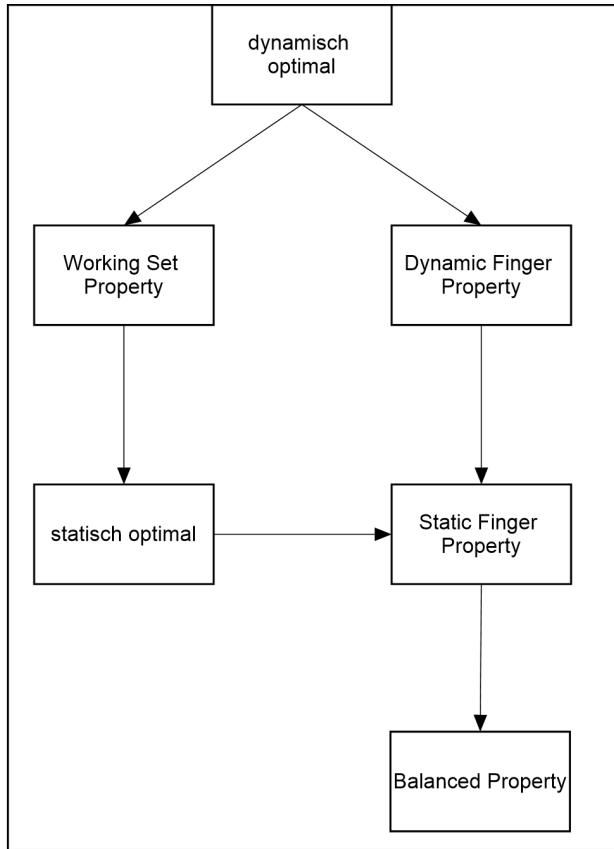


Abbildung 21: Implikationen zwischen den Eigenschaften, abgeleitet aus einer Abbildung aus [7].

5 Tango Baum

Der Tango Baum ist ein aus BSTs, den **Hilfsbäumen**, bestehender BST. Auf die Anforderungen an die Hilfsbäume wird in Abschnitt 5.2 eingegangen. Der Tango Baum wurde von Demaine, Harmon, Iacono und Patrascu beschrieben [2], inklusive eines Beweises über seine $\log(\log(n))$ -competitiveness. Ebenfalls in [2] enthalten ist eine als **Interleave Lower Bound** bezeichnete Variation der ersten unteren Schranke von Wilber. Da diese für das Verständnis des Tango Baumes wesentlich ist, wird mit ihr gestartet, bevor es zur Beschreibung der Struktur selbst kommt.

5.1 Die Interleave Lower Bound

Sei $X = x_1, x_2, \dots, x_m$ eine Zugriffsfolge und sei $K = \{k \in \mathbb{N} \mid k \text{ ist in } X \text{ enthalten}\}$. Auch hier wird ein Lower Bound Tree verwendet. Dieser ist jedoch etwas anders definiert als in Abschnitt 4.2. Hier ist der Lower Bound Tree Y zu einer Zugriffsfolge X der komplette BST mit der Schlüsselmenge K . Anders als in Abschnitt 4.2 gibt es hier somit zu jeder Zugriffsfolge nur genau einen Lower Bound Tree. Abbildung 22 zeigt den Lower Bound Tree zur Zugriffsfolge 1, 2, ..., 15. Zu jedem Knoten v in Y werden zwei Mengen definiert. Die **linke Region** von v enthält den Schlüssel von v , sowie die im linken Teilbaum von v enthaltenen Schlüssel. Die **rechte Region** von v enthält die im rechten Teilbaum von v enthaltenen Schlüssel. Sei l der kleinste Schlüssel im Teilbaum mit der Wurzel v und r der größte. Sei $X_l^r = x_{1'}, x_{2'}, \dots, x_{m'}$ wie in Abschnitt 4.2 definiert. $i \in \{2, 3, \dots, m'\}$ ist ein **Interleave** durch v , wenn $x_{(i-1)}$ in der linken Region von v liegt und x_i in der rechten Region von v oder umgekehrt. In Y sind Knoten enthalten, bei denen die rechte Region leer ist. Durch diese kann es keinen Interleave geben. Sei U die Menge der Knoten von Y mit einer nicht leeren rechten Region. Sei $\text{inScore}(X, u)$ die Funktion, die zu dem Knoten $u \in U$ die Anzahl der Interleaves durch u , bezogen auf X zurückgibt. Die Funktion $IB(X)$ ist definiert durch:

$$IB(X) = \sum_{u \in U} \text{inScore}(X, u)$$

Sei T_0 der BST mit der Schlüsselmenge K auf der X ausgeführt wird. Für $i \in \{1, 2, \dots, m\}$ sei T_i der BST, der entsteht nachdem $\text{access}(x_i)$ auf T_{i-1} ausgeführt wurde. Zu $u \in U$ und $j \in \{0, 1, \dots, m\}$ gibt es einen **transition point** v in T_j . v ist ein Knoten mit den folgenden Eigenschaften:

1. Der Pfad von der Wurzel von T_j zu v enthält einen Knoten, dessen Schlüssel in der linken Region von u enthalten ist.
2. Der Pfad von der Wurzel von T_j zu v enthält einen Knoten, dessen Schlüssel in der rechten Region von u enthalten ist.
3. In T_j ist kein Knoten mit den Eigenschaften 1 und 2 enthalten, der eine kleinere Tiefe als v hat.

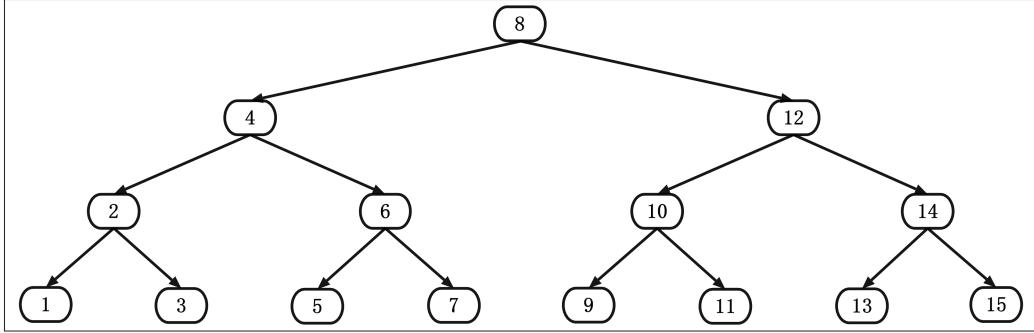


Abbildung 22: Der Lower Bound Tree zur Zugriffsfolge 1, 2, .., 15.

Im Beweis dieses Abschnitts wird gezeigt, dass $OPT(X) \geq \frac{IB(X)}{2} - n$ gilt, wobei n die Anzahl der Knoten im Lower Bound Tree ist. Dafür werden jedoch noch drei Lemmata zu den Eigenschaften der transition points benötigt. Bei allen dreien wird $X = x_1, x_2, \dots, x_m$ die Zugriffsfolge sein, die auf einem BST T_0 ausgeführt wird. Y ist ein zu X erstellter Lower Bound Tree mit der Schlüsselmenge K . Für $i \in \{1, 2, \dots, m\}$ ist T_i der BST der durch Ausführen von $access(x_i)$ auf $T_{(i-1)}$ entsteht. Und U ist die Menge der Knoten von Y , bei denen die rechte Region nicht leer ist. Außerdem gilt $j \in \{0, 1, \dots, m\}$.

Lemma 5.1. *Es gibt zu jedem Knoten $u \in U$ genau einen transition point in T_j .*

Beweis. Sei l der kleinste Schlüssel in der linken Region von u und r der größte Schlüssel in der rechten Region. Im Teilbaum mit der Wurzel u sind genau die Schlüssel $K_l^r = \{k \in K | k \in [l, r]\}$ enthalten. Sei v_l der gemeinsame Vorfahre aller Knoten mit einem Schlüssel aus der linken Region von u in T_j mit der größten Tiefe. Sei v_r der gemeinsame Vorfahre aller Knoten mit einem Schlüssel aus der rechten Region von u in T_j mit der größten Tiefe. $key(l)$, bzw. $key(r)$ muss selbst in der linken, bzw. rechten Region von u enthalten sein, vergleiche das Bilden von T_l^r in Abschnitt 4.2. Sei w der gemeinsame Vorfahre aller Schlüssel aus der linken und der rechten Region von u in T_j mit der größten Tiefe. Es muss $key(w) \in [l, r]$ gelten. Somit muss $key(w)$ entweder in der linken oder rechten Region von u enthalten sein. Da w der Knoten mit der kleinsten Tiefe sein muss, für den $key(w) \in [l, r]$ gilt, muss entweder $w = v_l$ oder $w = v_r$ gelten, je nachdem wessen Tiefe kleiner ist. Für den Fall $w = v_l$ ist v_r der transition point in T_j zu u und für den Fall $w = v_r$ ist es v_l . Es wird der Fall $w = v_l$ betrachtet, der andere kann direkt daraus abgeleitet werden. Im Pfad $P_u = v_0, v_1, \dots, v_r$ von der Wurzel v_0 zu v_r ist v_l enthalten und da v_r ein gemeinsamer Vorfahre der Schlüssel aus der rechten Region von u ist, muss v_r der einzige Knoten mit einem Schlüssel aus

der rechten Region von u in P_u sein. Jeder Pfad P in T_j von der Wurzel zu einem Knoten mit einem Schlüssel aus der rechten Region von u muss mit v_0, v_1, \dots, v_r beginnen. Somit kann es keinen weiteren transition point für u in T_j geben.

□

Der Knoten auf den der Zeiger p zum Ausführen von *access* gerade zeigt wird als **berührter Knoten** bezeichnet. Im zweiten Lemma geht es darum, dass sich der transition point v eines Knotens nicht verändern kann, solange v nicht wenigstens einmal der berührte Knoten war.

Lemma 5.2. *Sei v der transition point in T_j zu einem Knoten $u \in U$. Sei $l \in N$, mit $j < l \leq m$. Gilt für alle x_i mit $i \in [j+1, l]$: Während der Ausführungszeit von $\text{access}(x_i)$ war v nicht wenigstens einmal der berührte Knoten, dann ist v während der gesamten Ausführungszeit von $\text{access}(x_j), \text{access}(x_{j+1}), \dots, \text{access}(x_l)$ der transition point zu u .*

Beweis. Sei v_l der gemeinsame Vorfahre aller Knoten mit einem Schlüssel aus der linken Region von u in T_j mit der größten Tiefe. Sei v_r der gemeinsame Vorfahre aller Knoten mit einem Schlüssel aus der rechten Region von u in T_j mit der größten Tiefe. Hier wird wieder ohne Verlust der Allgemeinheit der Fall $v = v_r$ betrachtet. Da v_r nicht berührt wird, wird auch kein Knoten mit einem Schlüssel aus der rechten Region von u berührt. v_r ist somit während der gesamten Ausführungszeit von $\text{access}(x_j), \text{access}(x_{j+1}), \dots, \text{access}(x_l)$ der gemeinsame Vorfahre der Schlüssel aus der rechten Region von u mit der größten Tiefe. Knoten mit einem Schlüssel aus der linken Region von u könnten berührt werden. Zu einem Ausführungszeitpunkt t kann deshalb ein Knoten $v_{lt} \neq v_l$ der gemeinsame Vorfahre der Knoten mit einem Schlüssel aus der linken Region von u mit der größten Tiefe sein. Da v_r nicht berührt wird kann zu keinem Zeitpunkt v_l im Teilbaum mit der Wurzel v_r enthalten sein. Somit kann auch v_{lt} nicht in diesem Teilbaum enthalten sein. Somit muss die Tiefe von v_{lt} kleiner sein, als die von v_r und v_r bleibt der transition point von u .

□

Im dritten Lemma wird gezeigt, dass ein Knoten v in T_j nur der transition point zu einem Knoten aus U sein kann.

Lemma 5.3. *Sei $u_1, u_2 \in U$, mit $u_1 \neq u_2$. Sei v_1 der transition point zu u_1 und v_2 der zu u_2 in T_j . Dann muss $v_1 \neq v_2$ gelten.*

Beweis. Sei v_l , bzw. v_r der gemeinsame Vorfahre aller Knoten mit einem Schlüssel aus der linken, bzw. rechten Region von u_1 in T_j mit der größten Tiefe. Sei w_l , bzw. w_r der gemeinsame Vorfahre aller Knoten mit einem

Schlüssel aus der linken, bzw. rechten Region von u_2 in T_j mit der größten Tiefe. Ist weder u_1 ein Vorfahre von u_2 , noch u_2 einer von u_1 , dann muss auch $w_l \neq v_l \wedge w_l \neq v_r$, sowie $w_r \neq v_l \wedge w_r \neq v_r$ gelten, da die Teilbäume mit den Wurzeln u_1 und u_2 dann über disjunkte Schlüsselmengen verfügen. Somit müssen die transition points von u_1 und u_2 unterschiedlich sein. Sei u_1 ein Vorfahre von u_2 . Es werden drei Fälle unterschieden:

1. Ist $\text{key}(v_1)$ nicht im Teilbaum mit der Wurzel u_2 enthalten, so kann v_1 nicht der transition point von u_2 sein.
2. $\text{key}(v_1)$ ist im Teilbaum mit der Wurzel u_2 enthalten und $\text{key}(v_1)$ ist in der linken Region von u_1 enthalten:

Da u_1 ein Vorfahre von u_2 ist und $\text{key}(v_1)$ im Teilbaum mit der Wurzel u_2 enthalten ist, müssen alle Schlüssel im Teilbaum mit der Wurzel u_2 in der linken Region von u_1 enthalten sein. Da der Schlüssel von v_1 in der linken Region von u_1 liegt, muss v_r ein Vorfahre von v_l in T_j sein. $\text{key}(v_1)$ muss somit der Schlüssel von w_l , bzw. w_r sein, je nachdem wessen Tiefe kleiner ist. Denn andererseits könnte ein Pfad von der Wurzel von T_j zu v_1 angegeben werden, der zwei Knoten aus der linken Region von u_1 enthält. Das ist jedoch ein Widerspruch dazu, dass $\text{key}(v_1)$ in der linken Region von u_1 enthalten ist und v_1 zudem der transition point für u_1 ist.

v_2 ist entweder der Knoten w_l oder w_r , je nachdem wessen Tiefe größer ist. Somit gilt $v_1 \neq v_2$.

3. $\text{key}(v_1)$ ist im Teilbaum mit der Wurzel u_2 enthalten und $\text{key}(v_1)$ ist in der rechten Region von u_1 enthalten:

Symmetrisch zu Fall 2.

□

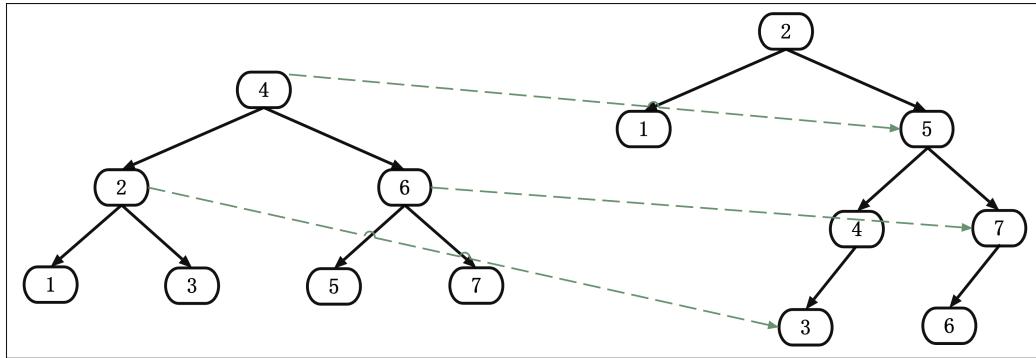


Abbildung 23: Transition point Zuordnung. Links ein Lower Bound Tree, rechts ein möglicher T_j .

Satz 5.1. Sei $X = x_0, x_1, \dots, x_m$ eine Zugriffsfolge und n die Anzahl der Knoten im zu X erstellten Lower Bound Tree Y . Dann gilt
 $OPT(X) \geq IB(X)/2 - n$.

Beweis. Es wird die Mindestanzahl der Berührungen von transition points gezählt. Sei U die Menge der Knoten von Y mit einer nicht leeren rechten Region. Durch Lemma 5.1 und Lemma 5.3 kann die Anzahl der Berührungen für jeden Knoten $u \in U$ einzeln bestimmt werden. Diese müssen dann lediglich noch addiert werden. Sei l, r, v_r und v_l wie in Lemma 5.1 zu u definiert, so dass entweder v_l oder v_r der transition point zu u sein muss, je nachdem welcher der beiden Knoten die größere Tiefe hat. Für $k \in \{1, 2, \dots, m\}$ sei $X_l^{r'} = x_{i_0}, x_{i_1}, \dots, x_{i_p}$ die Folge, die entsteht, wenn aus X_l^r alle x_k entfernt werden, für die gilt, x_k ist in der gleichen Region von u wie x_{k-1} . Damit gilt $inScore(X, u) = p$. Nun wird angenommen, dass die x_{i_j} mit, j ist eine gerade Zahl, in der rechten Region von u liegen und die x_{i_j} mit, j ist eine ungerade Zahl, in der linken Region. Der andere Fall kann wieder direkt abgeleitet werden. Sei $q \in \mathbb{N}$ mit $1 \leq q \leq \lfloor p/2 \rfloor$. $access(x_{i_{2q-1}})$ muss v_l berühren und $access(x_{i_{2q}})$ muss v_r berühren. Sei k_1 der Schlüssel des transition points von u zu Beginn von $access(x_{i_{2q-1}})$ und k_2 der Schlüssel des transition points von u zu Beginn von $access(x_{i_{2q}})$. Gilt $k_1 = k_2$, so muss der transition point von u in $access(x_{i_{2q}})$ berührt worden sein. Gilt $k_1 \neq k_2$, so muss der transition point von u nach Lemma 5.2 in $access(x_{i_{2q-1}})$ berührt worden sein. Aus der Konstruktion von $X_l^{r'}$ folgen daraus mindestens $\lfloor p/2 \rfloor \geq p/2 - 1$ Berührungen des transition point von u .

Addieren über alle Knoten aus U ergibt bei den Werten der $inScore$ Funktion die Interleave Lower Bound und bei den Berührungen von transition points zumindest $IB(X)/2 - |U| \geq IB(X)/2 - n$.

□

5.2 Der Aufbau des Tango Baumes.

Wie bereits erwähnt besteht ein Tango Baum T aus Hilfsbäumen. Eine Anforderung an einen Hilfsbaum mit n Knoten ist es, dass für seine Höhe $h = O(\log(n))$ gilt. T bietet lediglich eine $access$ Operation an. Ist T also erst einmal für die Schlüsselmenge K erzeugt, ist diese unveränderlich. Sei P der Lower Bound Tree aus Abschnitt 5.1 mit der Schlüsselmenge K . P wird auch als **Referenzbaum** für T bezeichnet. P ist kein Hilfsbaum und muss in Implementierungen auch nicht erstellt werden. Er dient aber dazu, den Aufbau von T vor und nach einer $access$ Operation zu veranschaulichen. Jeder innere Knoten p in P kann ein **preferred child** haben. Wurde während der Ausführungszeit von X noch keine $access$ Operation mit einem im

Teilbaum mit der Wurzel p enthalten Schlüssel als Parameter ausgeführt, so hat p kein preferred child. Ansonsten sei $access(k)$ die zuletzt ausgeführte Operation mit einem Schlüssel, der im Teilbaum mit der Wurzel p enthalten ist. Ist k in der linken Region von p enthalten, dann ist das linke Kind von p das preferred child von p . Ist k in der rechten Region von p enthalten, dann ist das rechte Kind von p , das preferred child von p . Wir erweitern die Knoten von P mit einer weiteren Variable $prefChild$, welche drei Werte annehmen kann. Sie enthält *none*, wenn ihr Knoten kein preferred child besitzt, *left*, wenn das linke Kind das preferred child ist, ansonsten entsprechend *right*. Hier ist bereits die Kopplung zur Interleave Lower Bound erkennbar. Ein Wechsel von $prefChild$ von *left* zu *right* oder umgekehrt, findet genau dann statt, wenn es zu einem interleave durch den Knoten kommt. Abbildung 24 stellt einen möglichen Zustand von P zwischen zwei $access$ Operationen dar. Dieser Zustand wird in diesem Abschnitt nun als durchgängiges Beispiel dienen. Es ist sofort ersichtlich, dass der Parameter der letzten $access$ Operation 8, 4, 2 oder 1 gewesen sein muss, da die Knoten mit diesen Schlüsseln von der Wurzel aus über preferred children erreichbar sind. Die Schlüssel 10 und 9 können noch nie Parameter einer $access$ Operation gewesen sein. Ansonsten müsste der Knoten mit dem Schlüssel 10 ein preferred child haben. Mit Hilfe der preferred children lassen sich die **preferred path** erstellen. Sei v ein Knoten in P , der nicht das preferred child eines anderen Knotens aus P ist. Dann ist der preferred path zu v , der längst mögliche Pfad (v_0, v_1, \dots, v_l) , mit $v_0 = v$ und $\forall i \in \{1, 2, \dots, l\}: v_i$ ist das preferred child von v_{i-1} .

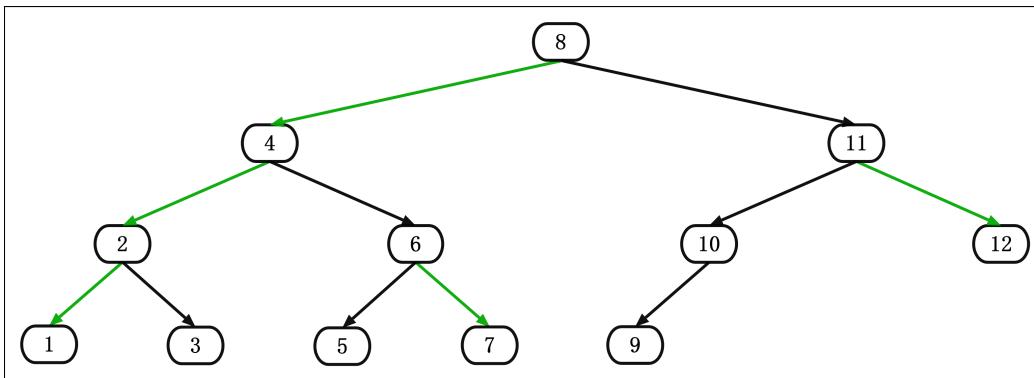


Abbildung 24: Die preferred children im Referenzbaum werden durch die grünen Pfeile markiert.

Nun werden die preferred paths des BSTs aus Abbildung 24 angegeben, wobei der Schlüssel jeweils als Bezeichner für den ihn enthaltenden Knoten

verwendet wird.

$$\begin{aligned}
 P_1 &= (8, 4, 2, 1) \\
 P_2 &= (3) \\
 P_3 &= (6, 7) \\
 P_4 &= (5) \\
 P_5 &= (11, 12) \\
 P_6 &= (10) \\
 P_7 &= (9)
 \end{aligned}$$

Da jeder Knoten nur das preferred child eines Knotens sein kann und Knoten die kein preferred child sind als Startknoten eines Pfades dienen, muss jeder Knoten in genau einem preferred path enthalten sein.

Zu jedem preferred path gibt es einen Hilfsbaum, der genau die Schlüssel enthält, die in den Knoten des Pfades enthalten sind. Da der Tango Baum den inneren Aufbau der Hilfsbäume nicht exakt vorschreibt, zeigt Abbildung 25 nur eine mögliche Konstellation.

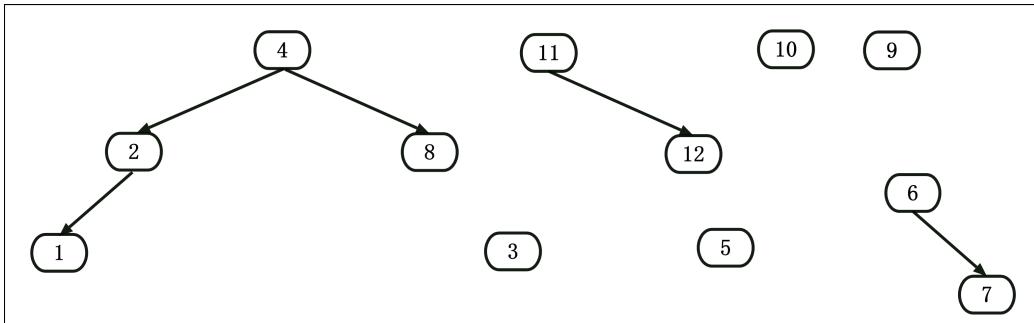


Abbildung 25: Die Hilfsbäume zu den preferred paths aus dem Beispiel.

Sei H die Menge der erstellten Hilfsbäume aus P . Mit dem folgenden Verfahren können Hilfsbäume zu einem Tango Baum zusammengefügt werden:

1. Gilt $|H| = 1$, dann ist das in H enthaltene Element der Tango Baum und es wird abgebrochen.
2. Wähle $h_1 \in H$ so, dass h_1 nicht den Schlüssel der Wurzel von P enthält.
3. Aufgrund der Konstruktion der preferred paths muss es genau einen Knoten v in h_1 geben, so dass der Knoten u in P mit $\text{key}(v) = \text{key}(u)$ nicht das preferred child seines Elternknotens ist. Sei h_2 der Hilfsbaum, der den Schlüssel des Elternknotens von u enthält. Entferne h_1 und h_2 aus H .

4. Sei w_1 die Wurzel von h_1 . Sei a der Knoten in h_2 , an dem die Standardvariante von *insert* einen für Schlüssel $key(w_1)$ erzeugten Knoten anfügen würde. Dann wird h_1 an a angefügt. Aufgrund der Links-Rechts-Beziehung in BST kann es nur eine Möglichkeit dafür geben. Sei h_3 der so entstandene BST.
5. Füge h_3 zu H hinzu, weiter bei Punkt 1.

Bei Punkt 4 ist sofort ersichtlich, dass es durch $key(w_1)$ zu keiner Verletzung der Links-Rechts-Beziehung kommt. Wie sieht es aber mit den anderen Schlüsseln aus h_1 aus?

In P sind alle in h_1 enthaltenen Schlüssel im Teilbaum mit der Wurzel u enthalten. Sei l der kleinste Schlüssel in diesem Teilbaum und r der größte. In P kann es außerhalb des Teilbaumes mit der Wurzel u keinen Schlüssel k mit $l \leq k \leq r$ geben. h_2 kann nur Schlüssel enthalten, die kleiner l oder größer r sind. Im Tango Baum kann es also keine Verletzung der Links-Rechts-Beziehung geben.

Abbildung 26 zeigt unseren Tango Baum T zum Beispiel. Die Wurzeln von Hilfsbäumen sind grün dargestellt.

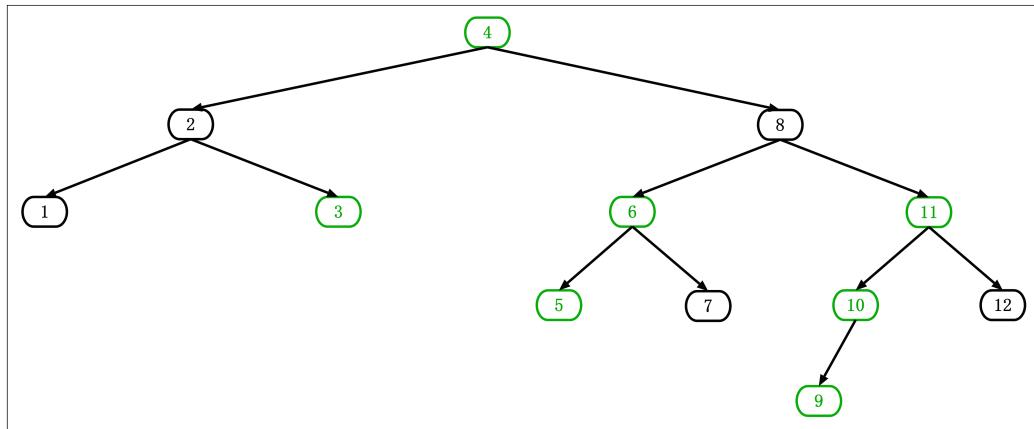


Abbildung 26: Der Tango Baum zu dem Beispiel.

Nehmen wir an, dass auf T $access(9)$ ausgeführt wird. Abbildung 27 zeigt den Zustand von P' .

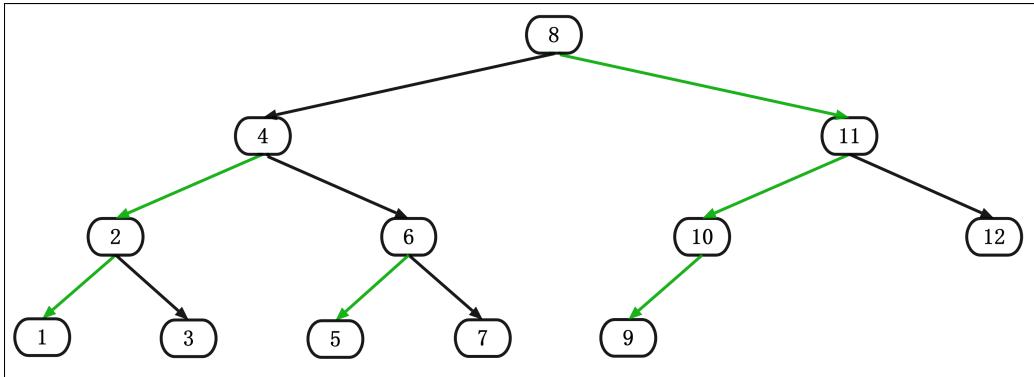


Abbildung 27: Die preferred children nachdem $access(9)$ ausgeführt wurde.

Abbildung 28 zeigt einen möglichen Zustand von T' .

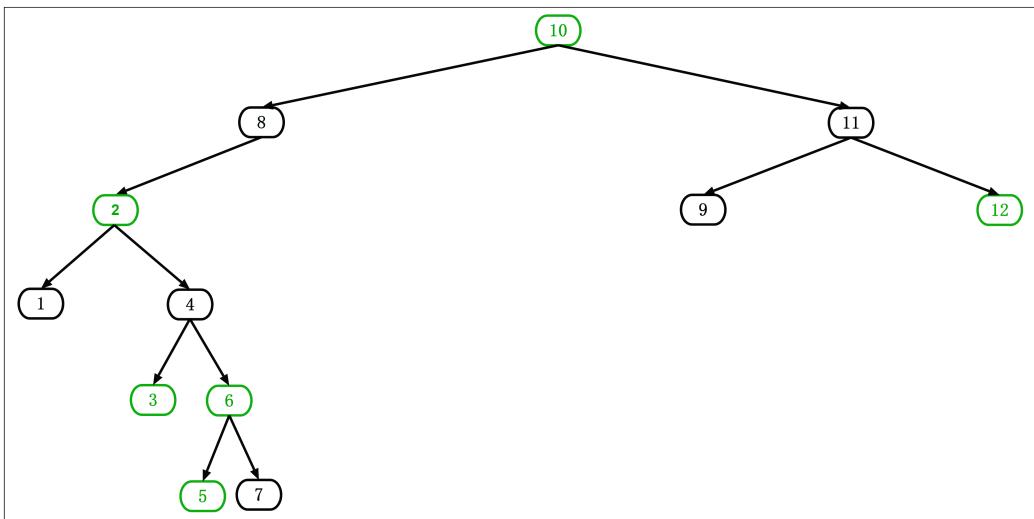


Abbildung 28: Der Tango Baum nachdem $access(9)$ ausgeführt wurde.

Im nächsten Abschnitt wird es vor allem darum gehen, wie eine Transformation, wie die von T zu T' , effizient durchgeführt werden kann.

5.3 Die $access$ Operation beim Tango Baum.

Die Knoten in einem Tango Baum sind mit zusätzlichen Daten erweitert. Sei v ein Knoten im Tango Baum. Es gibt eine boolesche Variable $isRoot$, die genau dann den Wert $true$ annimmt, wenn v die Wurzel eines Hilfsbaumes ist. In einer Konstante $depth$ wird die Tiefe des Knotens mit dem Schlüssel $key(v)$ in P gespeichert. Außerdem gibt es noch die Variablen $minDepth$ und

maxDepth. Sei v im Hilfsbaum H enthalten und sei H_v der Teilbaum mit der Wurzel v in H . Da H die Schlüssel von Knoten aus einem preferred path enthält, können die *depth* Konstanten zweier Knoten in H nicht den gleichen Wert haben. Sei *min* der kleinste Wert aller *depth* Konstanten der Knoten in H_v , dann entspricht *min* dem Wert der *minDepth* Variable von v . Sei *max* der größte Wert aller *depth* Konstanten der Knoten in H_v , dann entspricht *max* dem Wert der *maxDepth* Variable von v . Auf die Variablen und Konstanten eines Knotens v wird im Folgenden mit dem Punkt als Trennzeichen zugegriffen, z. B. $v.depth$.

Nun werden die Anforderungen an einen Hilfsbaum H aufgezählt:

1. Sei n die Anzahl der Knoten von H . Für die Höhe h von H gilt $h = O(\log(n))$.
2. H aktualisiert seine Zeiger auf andere Hilfsbäume.
3. H aktualisiert die Variablen *minDepth* und *maxDepth*.
4. H bietet eine Operation *concatenate(HB H₁, Node v_k, HB H₂)* an. HB ist eine Abkürzung für Hilfsbaum. Bei maximal einem HB darf die *isRoot* Variable der Wurzel den Wert *true* haben. Sei K_1 die Schlüsselmenge von H_1 , K_2 die Schlüsselmenge von H_2 und k der Schlüssel von v_k . Die Operation kann verwenden, dass für $k_1 \in K_1$ und $k_2 \in K_2$, $k_1 < k < k_2$ gilt. Es werden drei Fälle unterschieden. Sei w_1 die Wurzel von H_1 und w_2 die von H_2 .
 - (a) $w_1.isRoot = false$ und $w_2.isRoot = false$:
Die Operation gibt die Wurzel eines Hilfsbaumes H mit der Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ zurück.
 - (b) $w_1.isRoot = true$:
Die Operation gibt die Wurzel eines Hilfsbaumes H mit der Schlüsselmenge $K_2 \cup \{k\}$ zurück. An H ist ein Hilfsbaum H_3 mit der Schlüsselmenge K_1 angefügt. Die *isRoot* Variable der Wurzel von H_3 hat den Wert *true*.
 - (c) $w_2.isRoot = true$:
Die Operation gibt die Wurzel eines Hilfsbaumes H mit der Schlüsselmenge $K_1 \cup \{k\}$ zurück. An H ist ein Hilfsbaum H_3 mit der Schlüsselmenge K_2 angefügt. Die *isRoot* Variable der Wurzel von H_3 hat den Wert *true*.

In allen Fällen hat die *isRoot* Variable der Rückgabe den Wert *false*. Für die Laufzeit der Operation muss $O(\log(|K_1| + |K_2|))$ gelten.

5. H bietet eine Operation $split(key k)$ an. Die Operation kann verwenden, dass in H ein Knoten mit dem Schlüssel k existiert. Sei K die Schlüsselmenge von H . Die Operation gibt einen Knoten v mit dem Schlüssel k zurück. Das linke Kind von v muss die Wurzel eines Hilfsbaumes mit der Schlüsselmenge $K_l = \{i \in K \mid i < k\}$ sein. Das rechte Kind von v muss die Wurzel eines Hilfsbaumes mit der Schlüsselmenge $K_r = \{i \in K \mid i > k\}$ sein. Die $isRoot$ Variablen von v und dessen Kindern haben den Wert $false$. Für die Laufzeit der Operation muss $O(\log(|K|))$ gelten.

Jetzt werden noch zwei Hilfsoperationen vorgestellt, die für $access$ benötigt werden.

cut Operation: $cut(depth d)$ zerteilt einen Hilfsbaum A in zwei Hilfsbäume. Es dürfen nur Werte für d übergeben werden, zu denen es in A einen Knoten v mit $v.depth = d$ gibt. Die Rückgabe ist ein Hilfsbaum G mit den Schlüsseln der Knoten mit $depth \leq d$ in A . An G ist ein Hilfsbaum mit den restlichen Schlüsseln aus A angefügt. Zunächst werden die Knoten mit den Schlüsseln l, l', r und r' in A gesucht. l ist der kleinste Schlüssel eines Knotens v_l in A , mit $v_l.depth > d$. r ist der größte Schlüssel eines Knotens v_r in A , mit $v_r.depth > d$. l' ist der Schlüssel des Vorgängers von v_l und r' der Schlüssel des Nachfolgers von v_r . l und r müssen in A enthalten sein, l' und r' könnten auch fehlen. v_l kann wie folgt gefunden werden:

Zu Beginn wird der Zeiger p auf die Wurzel von A gesetzt. Zeigt p nicht auf v_l , muss es im linken Teilbaum von p einen Knoten v mit $v.depth > d$ geben. Dies ist an der $maxDepth$ Variable des linken Kindes von p direkt abfragbar. Ist v_l erreicht, kann l' über eine Suche nach dem Vorgänger von v_l gefunden werden. Die Suche nach r und r' verläuft analog.

A enthält genau die Schlüssel der Knoten aus einem preferred path. Somit muss für jeden Schlüssel k eines Knotens v mit $v.depth \leq d$ in A entweder $k > r$ oder $k < l$ gelten, denn alle Schlüssel aus $[l, r]$ sind in P entweder im linken oder im rechten Teilbaum des Knotens mit dem Schlüssel k enthalten. Es wird nun der Ablauf von cut gezeigt. Wobei angenommen wird, dass sowohl l' als auch r' existieren. Die anderen Fälle können einfach abgeleitet werden.

1. Sei w_a die Wurzel von A . Setze $w_a.isRoot$ auf $false$.
2. Führe $split(l')$ auf A aus. Sei v_l die Rückgabe von $split(l')$. Sei B der linke Teilbaum von v_l und C der rechte Teilbaum.

3. Führe $split(r')$ auf C aus. Sei v_r die Rückgabe von $split(r')$. Sei D der linke Teilbaum von v_r und E der Rechte.
4. Setze v_r als das rechte Kind von v_l .
5. Setze die *isRoot* Variable der Wurzel von D auf *true*.
6. Führe $w_F = concatenate(D, v_r, E)$ aus. Sei F der Hilfsbaum von dem w_F die Wurzel ist.
7. Führe $w_G = concatenate(B, v_l, F)$ aus. Sei G der Hilfsbaum von dem w_G die Wurzel ist.
8. Setze die *isRoot* Variable der Wurzel von G auf *true*.

Nach der Operation ist die Wurzel von G auch immer die Wurzel des Tango Baumes. Abbildung 29 demonstriert den Ablauf nochmals und Abbildung 30 zeigt einen verkürzten Ablauf bei fehlendem r' .

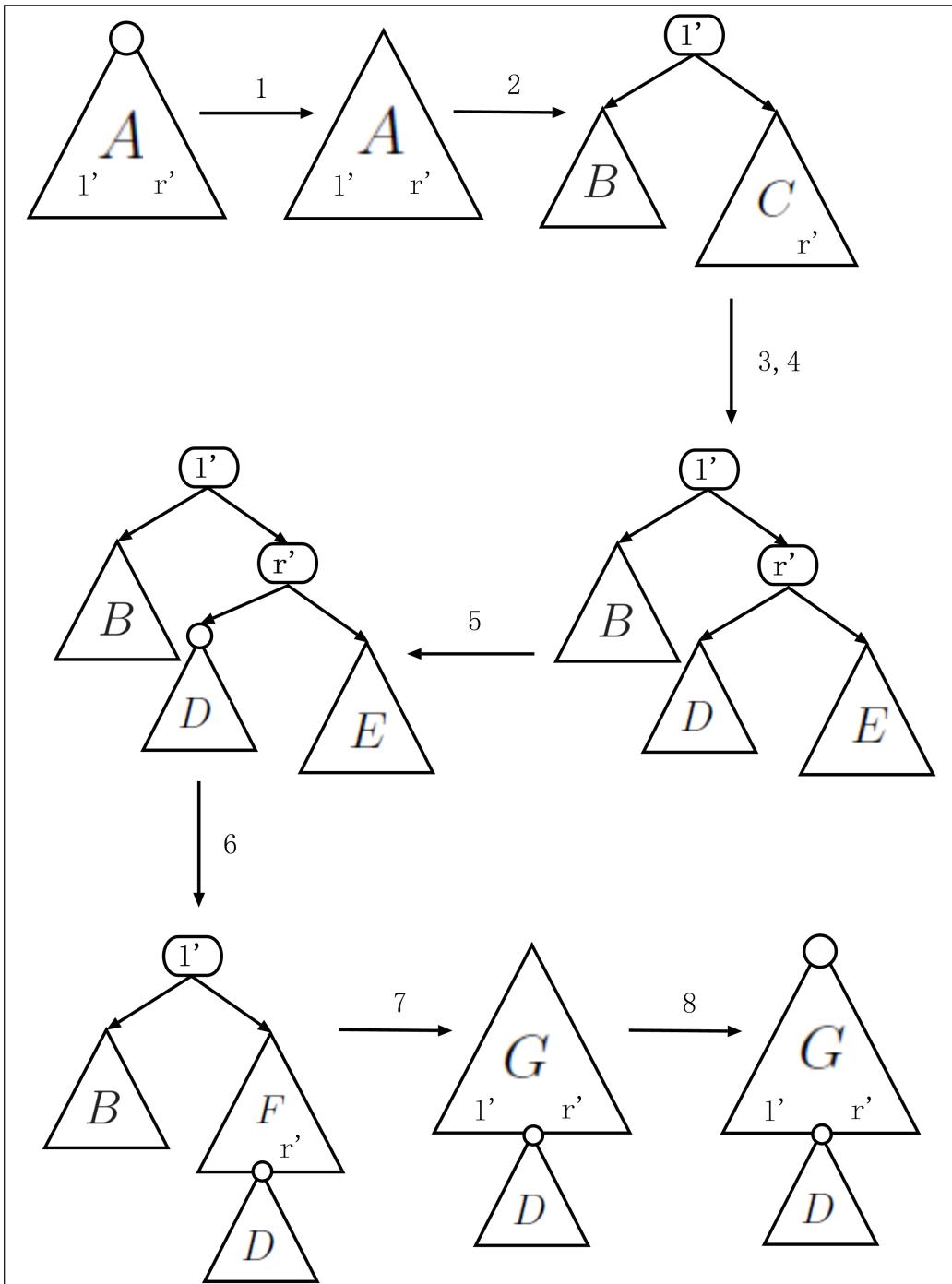


Abbildung 29: Ablauf von $\text{cut}(d)$. Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.

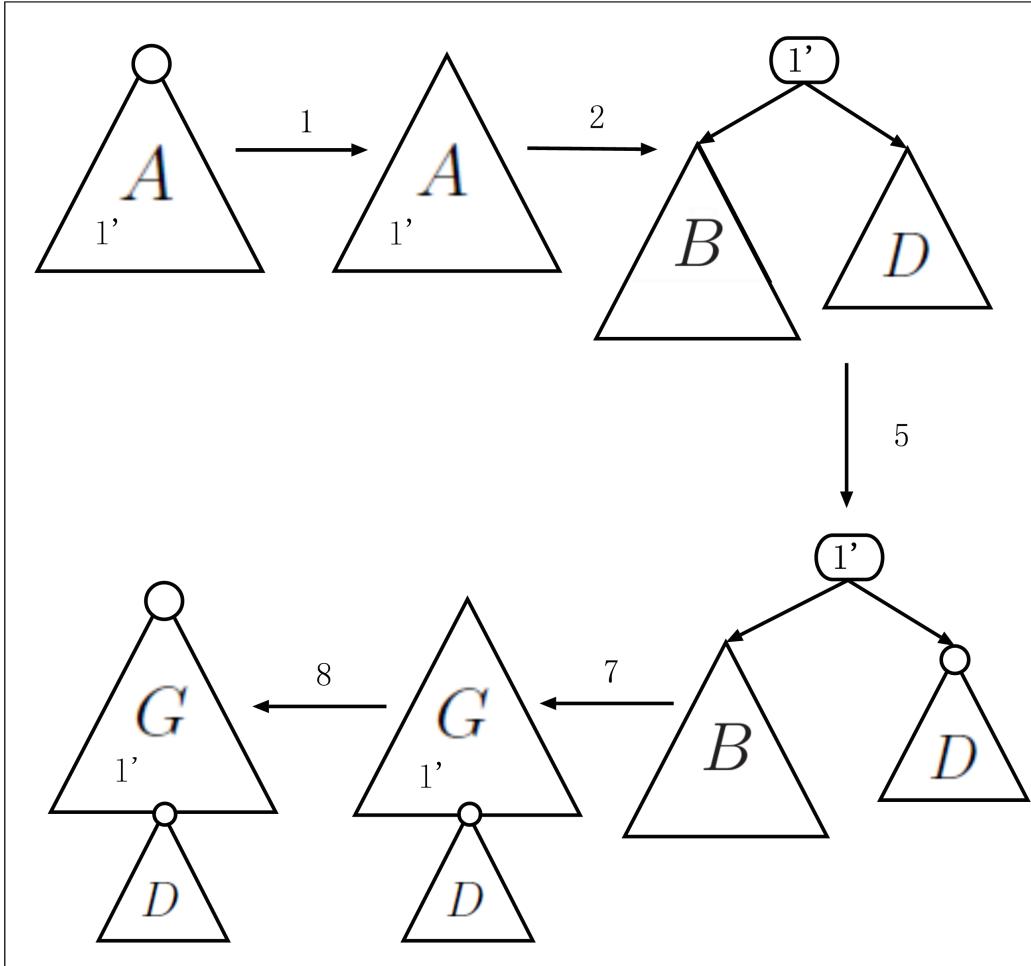


Abbildung 30: Ablauf von $cut(d)$ bei fehlenden r' . Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.

Sei n die Anzahl der Knoten von A . Jeder der acht Schritte kann in $O(\log(n))$ Zeit ausgeführt werden. Somit gilt auch für die Gesamtaufzeit $O(\log(n))$.

join Operation: $join(HB\ H_1, HB\ H_2)$ fügt die Hilfsbäume H_1 und H_2 zu einem Hilfsbaum H zusammen. Auch H repräsentiert wieder einen preferred path. Es muss also möglich sein einen Pfad im Referenzbaum zu bilden, der genau die Knoten mit den Schlüsseln aus H_1 und H_2 enthält.

Sei v_1 die Wurzel von H_1 und v_2 die von H_2 . Es muss

$v_1.maxDepth + 1 = v_2.minDepth$ gelten. Auch hier werden Schlüssel l , l' , r und r' verwendet. Sei l der kleinste Schlüssel in H_2 und r der größte Schlüssel in H_2 . Für jeden Schlüssel k in H_1 muss entweder $k < l$ oder $k > r$ gelten. l' ist der größte Schlüssel in H_1 mit $l' < l$. r' ist der kleinste Schlüssel in H_1

mit $r' > r$. Die Schlüssel l' und r' könnten wieder fehlen. Wird in H_1 ein Schlüssel aus H_2 gesucht, so muss die Suche entweder bei l' , bzw. r' erfolglos enden. Der andere Schlüssel kann dann mit einer Suche nach dem Nachfolger, bzw. Vorgänger gefunden werden. Der Ablauf von *join* ist dem von *cut* recht ähnlich. Wieder wird angenommen, dass l' und r' existieren.

1. Sei w_1 die Wurzel von H_1 und w_2 die von H_2 . Setze $w_1.isRoot$ und $w_2.isRoot$ auf *false*.
2. Führe *split*(l') auf H_1 aus. Sei v_l die Rückgabe von *split*(l'). Sei B der linke Teilbaum von v_l und C der Rechte.
3. Führe *split*(r') auf C aus. Sei v_r die Rückgabe von *split*(r'). Sei E der rechte Teilbaum von v_r . Der linke Teilbaum von v_r muss der leere Baum sein.
4. Setze v_r als das rechte Kind von v_l . Setze die Wurzel von H_2 als das linke Kind von v_r .
5. Führe $w_F = \text{concatenate}(H_2, v_r, E)$ aus. Sei F der Hilfsbaum von dem w_F die Wurzel ist.
6. Führe $w_H = \text{concatenate}(B, v_l, F)$ aus. Sei H der Hilfsbaum von dem w_H die Wurzel ist.
7. Setze die *isRoot* Variable der Wurzel von H auf *true*.

Nach der Operation ist die Wurzel von H auch immer die Wurzel des Tango Baumes. Abbildung 31 demonstriert den Ablauf nochmals und Abbildung 32 zeigt einen verkürzten Ablauf bei fehlendem r' .

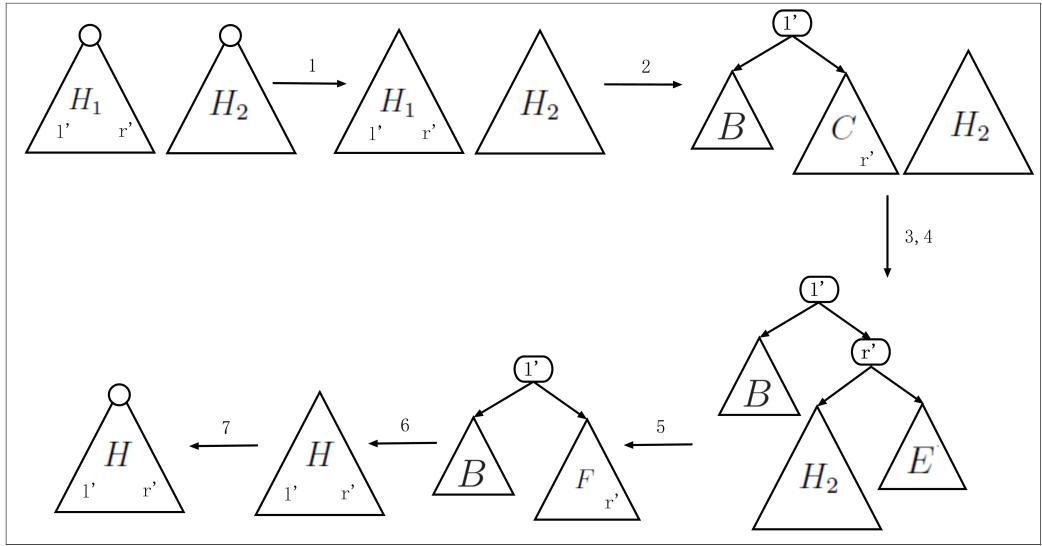


Abbildung 31: Ablauf von $\text{join}(H_1, H_2)$. Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert

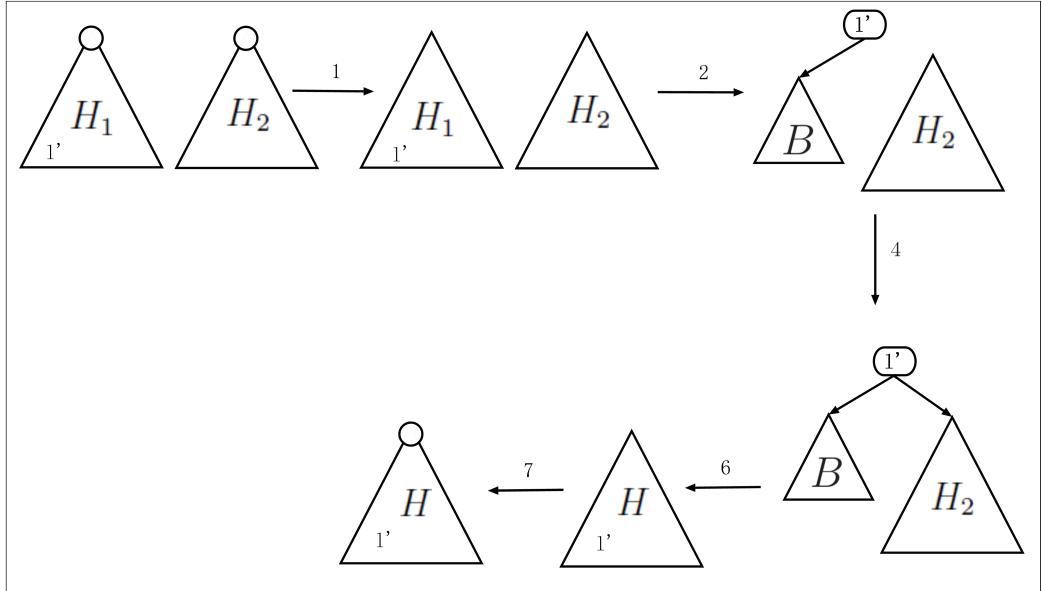


Abbildung 32: Ablauf von $\text{join}(H_1, H_2)$ bei fehlendem r' . Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.

Sei n die Anzahl der Knoten von H . Jeder der sieben Schritte kann in $O(\log(n))$ Zeit ausgeführt werden. Somit gilt auch für die Gesamtlaufzeit $O(\log(n))$.

access Operation: Nun wird die *access* Operation des Tango Baumes betrachtet. Sei k der Parameter der Operation und p der Zeiger der Operation in den BST. Solange p auf einen Knoten zeigt, der sich im Hilfsbaum mit der Wurzel des Tango Baumes T befindet, verhält sich die Operation wie die Standardvariante von *search*. Erreicht p die Wurzel eines anderen Hilfsbaumes H_2 , muss sich ein preferred child in P verändert haben. T wird mit *cut* und *join* so angepasst, dass er wieder die preferred paths in P repräsentiert. Anschließend startet p wieder an der Wurzel von T . Erreicht p den Knoten mit dem Schlüssel k , wird das preferred child des Knotens mit dem Schlüssel k in P auf *left* gesetzt. Daher kann nochmals eine Anpassung notwendig sein. Die Operation wird noch detaillierter beschrieben. Zur Vereinfachung bezeichnet T immer den aktuellen Zustand des Tango Baumes und H_1 immer den Hilfsbaum mit der Wurzel von T :

1. Setze p auf die Wurzel von H_1 .
2. Suche nach k . Wird k innerhalb von H_1 erreicht, weiter bei Punkt 5. Ansonsten wird die Wurzel eines Hilfsbaumes H_2 erreicht.
3. Sei w_2 die Wurzel von H_2 . Führe $H_3 = \text{cut}(w_2.\text{minDepth} - 1)$ auf H_1 aus.
4. Führe $\text{join}(H_3, H_2)$ aus. Weiter bei Punkt 1.
5. Sei v der Knoten mit $\text{key}(v) = k$. Führe $H_4 = \text{cut}(v.\text{depth})$ aus.
6. Suche ausschließlich im linken Teilbaum von v nach dem Vorgänger von v , bis die Wurzel eines Hilfsbaumes erreicht wird oder ein rechtes Kind fehlt. Wird keine Wurzel erreicht, weiter bei Punkt 8.
7. Sei H_5 der Hilfsbaum, auf dessen Wurzel p zeigt. Führe $\text{join}(H_4, H_5)$ aus.
8. Gib p zurück.



Abbildung 33: Darstellung von Schritt 3 und 4 der *access* Operation.

Abbildung 33 zeigt die Schritte 3 und 4 des Ablaufes. Zu klären ist noch, warum im sechsten Punkt die Wurzel des richtigen Hilfsbaumes gefunden werden muss.

Seien u und u_c Knoten in P , so dass u_c das linke Kind von u , aber nicht das preferred child von u ist. Sei v , bzw. v_c der Knoten in T mit $\text{key}(v) = \text{key}(u)$, bzw. $\text{key}(v_c) = \text{key}(u_c)$. Sei H_1 mit der Wurzel w_1 der Hilfsbaum, der v enthält und H_2 mit der Wurzel w_2 der Hilfsbaum, der v_c enthält. Es muss einen Pfad $P = (v_0, v_1, \dots, v_m)$ geben, mit $v_0 = w_1$, $v_m = w_2$ und v_{m-1} ist in H_1 enthalten. Aufgrund der Links-Rechts-Beziehung in H_1 , muss v_m entweder das linke Kind von v sein oder das rechte Kind des Vorgängers v_v von v in H_1 .

Sei v_m das rechte Kind von v_v . Dann kann v nicht im rechten Teilbaum von v_v liegen (im linken natürlich auch nicht). Angenommen v ist kein Vorfahre von v_v , dann muss es einen Knoten w geben, mit v_v liegt im linken Teilbaum von w und v im rechten Teilbaum. Ein Widerspruch dazu, dass v_v der Vorgänger von v ist.

Es gibt also in jedem Fall einen Pfad von v zu w_2 in T . w_2 kann bezogen auf T nur im linken Teilbaum von v enthalten sein. Für alle in H_1 enthaltenen Schlüssel k_1 gilt entweder $k_1 > \text{key}(v) > \text{key}(v_v)$ oder $\text{key}(v) > \text{key}(v_v) > k_1$. Somit muss w_2 gefunden werden, indem wie in Schritt sechs vorgegangen wird.

5.4 Laufzeitanalyse für access

Zunächst wird mit zwei Lemmata die Einzeloperation betrachtet, bevor es dann im Satz um Zugriffsfolgen geht. Alle drei Abschnitte basieren auf [2].

Lemma 5.4. *Sei n die Anzahl der Knoten eines Tango Baumes T_{i-1} . Sei k die Anzahl der Knoten, bei denen sich während der Ausführung von $\text{access}(x_i)$*

das preferred child geändert hat. Für die Laufzeit $access(x_i)$ gilt dann $O((k+1)(1 + \log(\log(n))))$.

Beweis. Bezeichne T_i den Tango Baum nach der Ausführung von $access(x_i)$. Zuerst werden die Kosten für das Suchen betrachtet. Der Zeiger p der Operationen startet maximal $k+1$ mal an der Wurzel des Tango Baumes. Für die Länge eines Pfades innerhalb eines Hilfsbaumes gilt $O(\log(\log(n)))$, denn für die Anzahl der Knoten eines preferred path gilt $O(\log(n))$ und ein Hilfsbaum muss ein balancierter BST sein. Die Gesamtkosten ergeben sich damit zu $O((k+1)(1 + \log(\log(n))))$.

Nun werden die Kosten zum Erzeugen von T_i aus T_{i-1} betrachtet. Pro Veränderung eines preferred child kommt es zu Kosten von $O(1 + \log(\log(n)))$ aufgrund einer *cut* und einer *join* Operation. Für das Suchen des Hilfsbaumes in Punkt 6 der Beschreibung entstehen auch wieder Kosten von $O(\log(\log(n)))$. Somit gilt auch für die Gesamtkosten $O((k+1)(1 + \log(\log(n))))$. \square

Sei $IB_i(X)$ die Differenz von $IB(x_1, x_2, \dots, x_i)$ und $IB(x_1, x_2, \dots, x_{i-1})$.

Lemma 5.5. *Sei T ein Tango Baum mit dem Referenzbaum P . Während der Ausführung von $access(x_i)$ wechselt in P bei genau $IB_i(X)$ Knoten das preferred child vom linken Kind zum rechten Kind oder vom rechten Kind zum linken Kind.*

Beweis. Sei $p \in P$. Das preferred child von p wechselt während $access(x_i)$ von links nach rechts, wenn x_i in der rechten Region von p liegt und der letzte Zugriff innerhalb des Teilbaumes mit der Wurzel p in der linken Region von p lag. Das preferred child von p wechselt während $access(x_i)$ von rechts nach links, wenn x_i in der linken Region von p liegt und der Schlüssel des vorherigen Zugriffs innerhalb des Teilbaumes mit der Wurzel p in der rechten Region von p lag. Das entspricht jeweils genau einem Interleave durch p . Zu beachten ist noch, dass der erste Zugriff auf den Teilbaum mit der Wurzel p weder zu einem Interleave, noch zu einem Wechsel eines preferred child von links, bzw. rechts zu rechts, bzw. links führt. \square

Satz 5.2. *Sei $X = x_1, x_2, \dots, x_m$ eine Zugriffsfolge und P ein dazu erstellter Referenzbaum. Für die Laufzeit eines mit P erstellten Tango Baumes mit n Knoten zum Ausführen von X gilt $O((OPT(X) + n) + (1 + \log(\log(n))))$.*

Beweis. Nach Lemma 5.5 gibt es nicht mehr als $IB(X)$ Wechsel der preferred children von links nach rechts oder umgekehrt. Zudem gibt es maximal n zusätzliche Änderungen bei preferred children. (Erstzugriff in den Teilbaum). Die Gesamtanzahl der Änderungen von preferred children ist somit höchstens

$IB(X) + n$. Mit Lemma 5.4 ergeben sich Gesamtkosten von $O((IB(X) + n + m)(1 + \log(\log(n))))$. Mit $OPT(X) \geq IB(X)/2 - n$ aus Satz 5.1 ergibt sich $O((OPT(X) + n + m)(1 + \log(\log(n))))$. Mit $OPT(X) \geq m$ ergibt sich dann die Behauptung. \square

Für $m \in \Omega(n)$ gilt dann auch $O(OPT(X)(1 + \log(\log(n))))$.

Kommt es bei $access(x)$ zu $\Omega(\log(n))$ Wechsel bei preferred children, muss der Hilfsbaum an der Wurzel des Tango Baumes $\Omega(\log(n))$ mal durchsucht werden. Somit erfüllt der Tango Baum die Balanced Property aus Abschnitt 4.5 nicht. Damit kann er aufgrund der Implikationen aus Abbildung 21 auch die anderen Eigenschaften aus diesem Abschnitt nicht erfüllen. Später werden zwei $\log(\log(n))$ -competitive BST vorgestellt, welche die Balanced Property erfüllen.

5.5 Tango Baum konformes Vereinigen beim Rot-Schwarz-Baum.

Die Ideen und Angaben zu den Laufzeiten in diesem und dem nächsten Abschnitt stammen aus [8], von Tarjan.

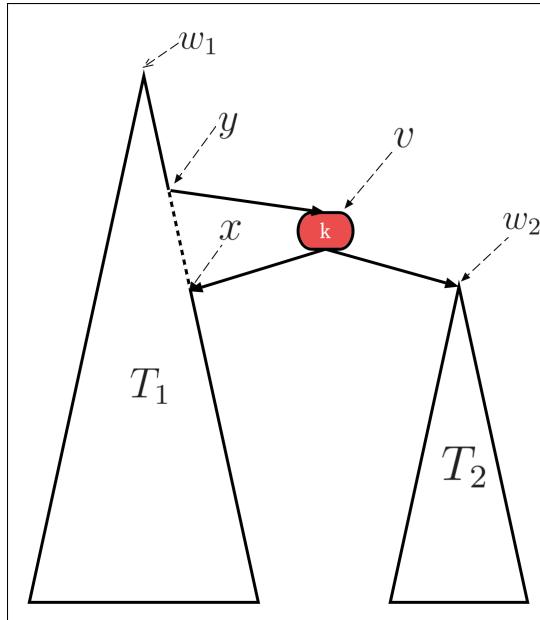


Abbildung 34: Beispielhaftes *concatenate* zweier RBTs unterschiedlicher Schwarz-Höhe, nach Schritt 1.



Abbildung 35: Beispielhaftes *concatenate* zweier RBTs gleicher Schwarz-Höhe, nach Schritt 1.

Hier wird die *concatenate*(RBT T_1 , Node v , RBT T_2) Operation beim Rot-Schwarz-Baum so eingeführt, wie es für den Tango Baum notwendig ist. Bei der Beschreibung der Operation wird auf die Pflege der Variablen *minDepth* und *maxDepth* verzichtet, damit sie nicht zu kleinteilig wird. Im Abschnitt zur Laufzeit wird darauf jedoch nochmal eingegangen. Sei K_1 die Schlüsselmenge von T_1 , K_2 die Schlüsselmenge von T_2 und k der Schlüssel von v . Die Operation gibt eine Referenz auf die Wurzel eines vereinigten RBTs T mit der Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ zurück. Dabei werden T_1 und T_2 zerstört. An die Parameter wird die Vorbedingung $(\forall i \in K_1 : i < k) \wedge (\forall j \in K_2 : k < j)$ gestellt.

Es werden im ersten Schritt der Ausführung drei Fälle unterschieden, wobei wieder der erste zutreffende Fall in aufsteigender Reihenfolge ausgewählt wird.

Fall 1: $bh(T_1) = bh(T_2) = 0$

In diesem Fall wird die Schwarz-Höhe von v auf 1 gesetzt, außerdem wird v rot gefärbt. An v werden zwei Sonderknoten angefügt. v ist die Wurzel von T .

In den restlichen Fällen ist nun immer zumindest ein Baum vorhanden, der über eine Wurzel verfügt. Der RBT mit der kleineren Schwarz-Höhe wird dabei an den mit der größeren „seitlich angefügt“. Abbildung 34 zeigt dies beispielhaft. Sind die Schwarz-Höhen gleich, wird wie in Abbildung 35 vor-

gegangen. Nun werden die verbleibenden Fälle beschrieben.

Fall 2: $bh(T_2) \leq bh(T_1)$

In diesem Fall wird T_2 bei T_1 mit Hilfe von v so angefügt, dass die Schwarz-Höhe jedes Knotens in T_1 und T_2 unverändert bleibt. Es sei w_1 die Wurzel von T_1 und w_2 die Wurzel von T_2 . Es sei P ein Pfad (r_0, r_1, \dots, r_l) in T_1 , so dass $r_0 = w_1$ gilt, r_l ein Blatt ist und $\forall i \in \{1, 2, \dots, l\}: r_i$ ist das rechte Kind von r_{i-1} gilt. P ist also der am weitesten rechts liegende Pfad von der Wurzel zu einem Blatt. Sei x der schwarze Knoten in P , mit $bh(x) = bh(w_2)$. x muss existieren, denn $bh(w_1) \geq bh(w_2)$ und $bh(r_l) \leq bh(w_2)$. Außerdem sind w_1 und r_l schwarz.

Nun wird die Schwarz-Höhe von v auf $bh(x) + 1$ gesetzt. Außerdem wird v rot gefärbt. Als das linke Kind von v wird x gesetzt, als rechtes Kind w_2 . Ist x die Wurzel von T_1 , so ist v die Wurzel von T . Ansonsten ist x das rechte Kind eines Knotens y . Das rechte Kind von y wird auf v gesetzt. Außerdem ist dann w_1 die Wurzel von T .

Fall 3: $bh(T_1) < bh(T_2)$

Hier wird T_1 bei T_2 mit Hilfe von v so angefügt, dass die Schwarz-Höhe jedes Knotens in T_1 und T_2 unverändert bleibt. Dieser Fall ist fast links-rechts-symmetrisch zu Fall 2. v kann lediglich nicht zur Wurzel von T werden, da $bh(T_1) \neq bh(T_2)$ gilt.

Resultat nach der Fallbehandlung: Dass ein Baum mit der Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ entstanden ist, ist an den Abbildungen 34 und 35 zu erkennen. Aufgrund der Vorbedingung an die Parameter muss T auch ein BST sein. Es müssen aber wieder die fünf Eigenschaften eines RBTs betrachtet werden:

1. Es ist immer noch jeder Knoten entweder rot oder schwarz.
2. Gilt $bh(T_1) \neq bh(T_2)$, so wurde mit w_1 oder w_2 ein schwarzer Knoten zur Wurzel von T . Andernfalls ist v die rote Wurzel von T und diese Eigenschaft ist verletzt.
3. Aufgrund der Sonderknoten sind die Blätter immer noch schwarz.
4. Da T_1 und T_2 RBTs waren muss nur die Situation um v betrachtet werden. v hat in jedem Fall schwarze Kinder. Gilt $bh(T_1) \neq bh(T_2)$ könnte der rote Knoten v jedoch einen roten Elternknoten y haben.
5. Die Schwarz-Höhe von v ist korrekt gesetzt. Existiert y , so hat sich seine Schwarz-Höhe nicht verändert, da v rot ist. Bei keinem anderen Knoten hat sich bezüglich bzgl. der Schwarz-Höhe etwas geändert.

Wir sind also in der Situation, dass nur entweder die Eigenschaft zwei oder vier verletzt sein kann. Wenn Eigenschaft vier verletzt ist, dann nur bei Knoten v . Das ist genau die Situation, für die *insertFixup* entworfen wurde. In Schritt zwei wird also *insertFixup* mit Parameter v aufgerufen und die Wurzel des resultierenden RBTs zurückgegeben.

Laufzeit: Sei n_1 die Anzahl der Knoten von T_1 , n_2 die Anzahl der Knoten von T_2 und $n = n_1 + n_2$. Der Tango Baum fordert von seinen Hilfsbäumen eine Laufzeit von $O(\log(n))$ für die eben vorgestellte Operation.

Für die Kosten zum Finden von x und dem Ausführen von

insertFixup gilt $O(\log(n))$. v zu erzeugen und in die Struktur einzubinden benötigt konstante Zeit. Die Vorgabe des Tango Baumes wird also eingehalten.

Nun wird noch kurz auf die Pflege der Variablen *minDepth* und *maxDepth* eingegangen. Nach dem ersten Schritt könnten diese Variablen bei allen Knoten im Pfad $P = (v_0, v_1, v_m)$ mit v_0 ist die Wurzel von T und $v_m = v$ falsch gesetzt sein. Um sie zu aktualisieren kann wie folgt vorgegangen werden:

Als erstes wird v_m betrachtet. Es werden die *minDepth* Variablen der Kinder von v und die *depth* Konstante von v betrachtet. $v.\text{minDepth}$ wird auf den kleinsten dieser drei Werte gesetzt. *maxDepth* wird analog aktualisiert. Nun wird mit den restlichen Knoten in P in der Reihenfolge $v_{m-1}, v_{m-2}, \dots, v_0$ genauso vorgegangen. Die Kosten für das Aktualisieren eines Knotens sind konstant und Kosten von $O(m)$ sind bereits bei der Suche nach x entstanden. Für den nächsten Abschnitt wird noch eine genauere Betrachtung der Gesamlaufzeit benötigt:

Es sei $d = |\text{bh}(T_1) - \text{bh}(T_2)|$. Die Suche nach x endet spätestens nachdem ein Pfad der Länge $2d + 1$ betrachtet wurde. Dabei steht die 1 für den Zugriff auf x selbst. Zu jedem schwarzen Knoten könnte noch ein roter Knoten dazukommen, bis x erreicht ist.

Jetzt wird noch auf die maximale Anzahl der Iterationen innerhalb *insertFixup* eingegangen. Sei w die Wurzel des vereinigten BSTs nach Schritt 1. v ist ein roter Knoten mit $\text{bh}(w) - \text{bh}(v) = d - 1$ und befindet sich in der Baumstruktur nicht tiefer als Ebene $2d + 2$. Deshalb führt *insertFixup* maximal $d + 1$ Iterationen durch.

5.6 Tango Baum konformes Aufteilen beim Rot-Schwarz-Baum.

Auch *split* (RBT T , key k) wird so vorgestellt, wie es für den Tango Baum notwendig ist. Vorbedingung an die Parameter ist, dass k in der Schlüssel-

menge K von T vorhanden ist. Zurückgegeben wird eine Referenz auf den Knoten v_k' mit dem Schlüssel k . Das linke Kind von v_k' ist die Wurzel eines RBTs T_L mit der Schlüsselmenge K_L , wobei gilt $K_L = \{i \mid i \in K \wedge i < k\}$. Das rechte Kind von v_k' ist die Wurzel eines RBTs T_R mit der Schlüsselmenge K_R , wobei gilt $K_R = \{i \mid i \in K \wedge i > k\}$. *split* gibt also in den meisten Fällen nicht die Wurzel eines RBTs zurück.

Die Operation setzt zunächst T_L auf den linken Teilbaum von v_k und T_R auf den rechten Teilbaum von v_k . Eventuell müssen die Wurzeln von T_R und T_L schwarz gefärbt werden. Sei (v_0, v_1, \dots, v_m) der Pfad von der Wurzel von T zu v_k . Es wird sich nun bei v_{m-1} startend, Knoten für Knoten in dem Pfad nach oben gearbeitet. Ist der Schlüssel eines Knotens kleiner als k , so wird dieser Knoten und der linke Teilbaum des Knotens zu T_L hinzugefügt. Dies übernimmt die *concatenate* Operation. Ist der Schlüssel größer als k , so wird der Knoten und dessen rechter Teilbaum zu T_R hinzugefügt. Folgende Aufzählung beschreibt den Vorgang genauer:

1. Verwende die *search* Operation, um den Knoten v_k mit dem Schlüssel k zu finden.
2. Setze den linken Teilbaum von v_k als T_L , den rechten als T_R . Löse beide Teilbäume und v_k aus T heraus.
3. Färbe die Wurzeln von T_L und T_R schwarz.
4. $\forall i \in \{0, 1, \dots, m-1\}$ absteigend sortiert: Färbe die Wurzel von t_i schwarz. Ist der Schlüssel k_i von v_i kleiner als k , führe $w = \text{concatenate}(T_L, v_i, t_i)$ aus und setze T_L auf den RBT, von dem w die Wurzel ist. Ansonsten führe $w = \text{concatenate}(T_R, v_i, t_i)$ aus und setze T_R auf den RBT, von dem w die Wurzel ist.
5. Füge T_R rechts an v an, T_L links.
6. Gib v_k' zurück.

Das T_L und T_R die gewünschten RBTs sind wird leicht erkannt. v_0 und einer seiner beiden Teilbäume wird korrekt zugeordnet. Die Wurzel des anderen Teilbaumes von v_0 ist v_1 . Alle Schlüssel, die nicht im Teilbaum mit der Wurzel v_1 liegen sind somit korrekt zugeordnet. Diese Betrachtung iteriert bis auf v_k getroffen wird. Die Schlüssel im Teilbaum mit der Wurzel v_k werden korrekt zugeordnet.

Laufzeit: Der Tango Baum fordert eine Laufzeit von $O(\log(n))$ von seinen Hilfsbäumen für *split*, mit n ist die Anzahl der Knoten. Punkt 1 kostet $O(\log(n))$. Das Durchführen von Punkt 2, 3, 5 und 6 kostet $O(1)$. Punkt 4 führt $O(\log(n))$ Aufrufe von *concatenate* durch. Das ergibt $O(\log(n) \log(n))$, was dann auch eine obere Schranke für die Gesamlaufzeit darstellt. Diese Schranke ist für unseren Einsatzzweck jedoch zu hoch.

Deshalb wird Punkt 4 nun genauer betrachtet, speziell die Konstruktion von T_L . Es sei l die Anzahl der Aufrufe von *concatenate* nach denen T_L neu gesetzt wird. Sei $\{t_1, t_2, \dots, t_l\}$ die Menge der aus T herausgelösten Teilbäume, die für die l Aufrufe verwendet werden, wobei t_1 zum ersten Aufruf gehört, t_2 zum zweiten, usw.. T_0 steht für den linken Teilbaum von v_k . Sei $j \in \{1, 2, \dots, l\}$. T_j ist der Zustand von T_L , nachdem *concatenate* mit Parameter t_j ausgeführt wurde. v_j ist der Elternknoten der Wurzel von t_j .

Sei $i \in \{1, 2, \dots, l-1\}$. Es gilt $bh(t_i) \leq bh(v_i)$ und $bh(t_i) < bh(v_{i+1})$. Nun wird durch Induktion über i gezeigt, dass eine der beiden folgenden Aussagen immer gelten muss:

1. $bh(T_i) \leq bh(v_{i+1})$
2. $bh(T_i) = bh(v_{i+1}) + 1$, beide Kinder der Wurzel von T_i sind schwarz und $bh(v_i) = bh(v_{i+1})$

Für $i = 0$ gilt T_i ist der linke Teilbaum von v_k und $bh(v_1) > bh(T_i)$.

Induktionsschritt:

$$T_i = \text{concatenate}(T_{i-1}, v_i, t_i).$$

Für T_{i-1} gilt Aussage 1:

Es gilt entweder $bh(T_i) = \max\{bh(T_{i-1}), bh(t_i)\}$ oder $bh(T_i) = \max\{bh(T_{i-1}), bh(t_i)\} + 1$. Ist das Erstere der Fall, ist nichts mehr zu zeigen. Beim anderen Fall galt entweder $bh(T_{i-1}) = bh(t_i)$ oder *insertFixup* hat die Schwarz-Höhe der Wurzel um eins erhöht. In beiden Fällen hat die Wurzel von T_i zwei schwarze Kinder, vergleiche Abschnitt 3.1. Gilt $bh(v_i) < bh(v_{i+1})$, ist Aussage eins korrekt, ansonsten Aussage zwei.

Für T_{i-1} gilt Aussage 2:

Es gilt $bh(T_i) = bh(T_{i-1})$, denn $bh(T_{i-1}) > bh(t_i)$ und *insertFixup* kann die Schwarz-Höhe eines RBTs ohne einem roten Kind der Wurzel nicht erhöhen, vergleiche Abschnitt 3.1. Sollte v_i durch den ersten Schritt der Operation zu einem roten Kind der Wurzel geworden sein, so hat v_i beim Aufruf

von *insertFixup* einen schwarzen Elternknoten. Auch in diesem Fall erhöht *insertFixup* die Schwarz-Höhe des Baumes nicht. Mit $bh(v_{i-1}) = bh(v_i)$ muss $bh(v_i) < bh(v_{i+1})$ gelten. Somit ist für T_i die Aussage eins korrekt.

Nun wird zur Betrachtung der Laufzeit von Punkt 4 zurückgekehrt:
Für $bh(t_i) < bh(T_{i-1})$ folgt daher $|bh(t_i) - bh(T_{i-1})| = O(1)$
und daraus

$$\sum_{i=1}^l |bh(t_i) - bh(T_{i-1})| = O(\log(n))$$

Der Gesamtaufwand für das Suchen von v in allen l Aufrufen berechnet sich mit:

$$\sum_{i=1}^l 2|bh(t_i) - bh(T_{i-1})| + 1 = 2 \left(\sum_{i=1}^l |bh(t_i) - bh(T_{i-1})| \right) + l = O(\log(n))$$

Für die Anzahl der Iterationen von *insertFixup* in allen l Aufrufen gilt:

$$\sum_{i=1}^l |bh(t_i) - bh(T_{i-1})| + 1 = \sum_{i=1}^l (|bh(t_i) - bh(T_{i-1})|) + l = O(\log(n))$$

Die Kosten innerhalb einer Iteration sind konstant. Damit ist $O(\log(n))$ eine obere Schranke für die Gesamtkosten zum Konstruieren von T_L . Für T_R gilt analog das Gleiche. Für die Gesamtkosten von *concatenate* innerhalb *split* gilt $O(\log(n))$, denn die Kosten für das Suchen und *insertFixup* überlagern die restlichen Kosten. Damit ist $O(\log(n))$ eine obere Schranke für die Kosten von Punkt 4 und somit auch für *split*. Der hier vorgestellte RBT hält also die Anforderung des Tango Baumes bezüglich der Laufzeit von *split* ein.

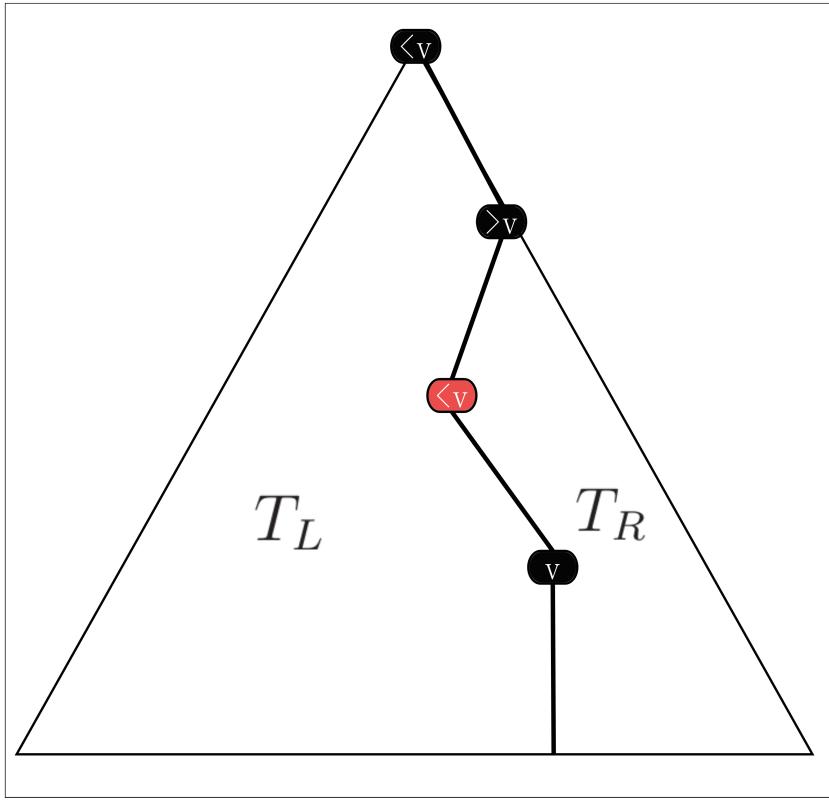


Abbildung 36: Beispielhaftes *split* eines RBTs mit Parameter v . Die Symbole in den Knoten beziehen sich auf die Schlüssel der Knoten.

6 Splay Baum

Der Splay Baum [1] ist ein dynamischer BST, der ohne zusätzliche Hilfsdaten in seinen Knoten auskommt. Nach einer *access*(k) Operation ist der Knoten mit dem Schlüssel k die Wurzel des Splay Baumes. Es gibt keine Invariante, welche eine bestimmte maximale Höhe garantiert. Splay Bäume können sogar zu Listen entarten. Amortisiert betrachtet verfügen sie dennoch über sehr gute Laufzeiteigenschaften.

6.1 Die *access* Operation beim Splay Baum.

Die wesentliche Arbeit leistet eine Hilfsoperation namens *splay*(key k). Nach deren Ausführung befindet sich der Knoten mit dem gesuchten Schlüssel k an der Wurzel und es wird nur noch eine Referenz auf ihn zurückgegeben.

splay Operation: Sei p der Zeiger der Operation in den BST. Zunächst wird eine gewöhnliche Suche ausgeführt, bis p auf den Knoten v mit dem Schlüssel k zeigt. Nun werden iterativ sechs Fälle unterschieden, bis v die Wurzel des Baumes darstellt. Zu jedem Fall gibt es einen der links-rechts-symmetrisch ist. Sei u der Elternknoten von v .

1. v ist das linke Kind der Wurzel (zig-Fall):
Es wird eine Rechtsrotation auf v ausgeführt. Nach dieser ist v die Wurzel des Splay Baumes und die Operation wird beendet.
2. v ist das rechte Kind der Wurzel (zag-Fall):
Symmetrischer Fall zu zig.
3. v ist ein linkes Kind und u ist ein linkes Kind (zig-zig-Fall):
Dieser Fall unterscheidet den Splay Baum von einem anderen BST (move-to-root) mit ganz unterschiedlichen Laufzeiteigenschaften. Es wird zuerst eine Rechtsrotation auf u ausgeführt und erst danach eine Rechtsrotation auf v . Bei move-to-root ist es genau umgekehrt.
4. v ist ein rechtes Kind und u ist ein rechtes Kind. (zag-zag-Fall):
Symmetrischer Fall zu zig-zig.
5. v ist ein linkes Kind und u ist ein rechtes Kind (zig-zag-Fall):
Es wird eine Rechtsrotation auf v ausgeführt. Im Anschluss wird eine Linksrotation auf u ausgeführt.
6. v ist ein rechtes Kind und u ist ein linkes Kind (zag-zig-Fall):
Symmetrischer Fall zu zig-zag.

Abbildung 37 zeigt drei der Fälle. Trotz der Einfachheit kann die Auswirkung einer einzelnen *splay* Operation groß sein. Abbildung 38 [1] zeigt eine solche Konstellation.

Die Laufzeit von *access* auf einem Splay Baum mit n Knoten ist $O(n)$.



Abbildung 37: Darstellung von zig, zig-zig und zig-zag.

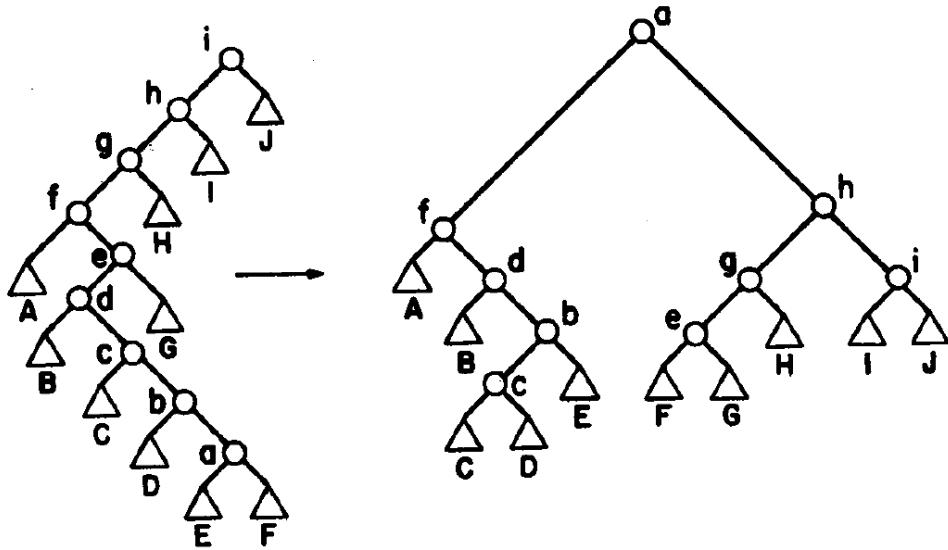


FIG. 4. Splaying at node a .

Abbildung 38: Eine einzige *splay* Operation. [1]

6.2 Amortisierte Laufzeitanalyse von *splay*.

Es wird die Potentialfunktionsmethode aus Kapitel 4.4 verwendet. Sei v ein Knoten im Splay Baum T . Eine Funktion $w(v)$ liefert zu jedem Knoten eine reelle Zahl > 0 , die **Gewicht** genannt wird. Eine Funktion $tw(v)$ bestimmt die Summe der Gewichte aller im Teilbaum mit der Wurzel v enthaltenen Knoten. Der **Rang** $r(v)$ ist definiert durch $r(v) = \log_2(tw(v))$. Sei V die Menge der Knoten von T . Als Potentialfunktion wird

$$\Phi = \sum_{v \in V} r(v)$$

verwendet.

Access Lemma 6.1. *Sei T ein Splay Baum mit n Knoten, Wurzel w und einem Knoten v mit dem Schlüssel k . Die amortisierten Kosten von $splay(k)$ sind maximal $3(r(w) - r(v)) + 1 = O(\log(tw(w)/tw(v))) = O(\log(n))$.*

Beweis. Es werden den Knoten fest zugeordnete Gewichte angenommen. Zunächst wird für *zig*, *zig-zig* und *zig-zag* gezeigt, dass die amortisierten Kosten nicht größer als $3(r(v)' - r(v)) + 1$ sind. Für die anderen drei Fälle folgt es dann aus der Symmetrie. Im Anschluss wird die gesamte Operation betrachtet. An Abbildung 37 ist zu erkennen, dass sich der Wert von $tw()$ bei den sechs Fällen nur an den Knoten v , dessen Elternknoten u und dem Elternknoten x von u verändern kann. Damit gilt:

$$\Phi' - \Phi = r(u)' + r(v)' + r(x)' - r(u) - r(v) - r(x)$$

zig: In diesem Fall existiert x nicht, damit gilt:

$\Phi' - \Phi = r(u)' + r(v)' - r(u) - r(v)$. Der Wert von $tw()$ für die Wurzel ist unabhängig vom Zustand des Splay Baumes, da an ihr alle im Baum vorhandenen Gewichte summiert werden. Deshalb muss $tw(v)' = tw(u)$ gelten. Daraus folgt $\Phi' - \Phi = r(u)' - r(v)$. Aus $r(v)' \geq r(u)'$ folgt

$\Phi' - \Phi \leq r(v)' - r(v) \leq 3(r(v)' - r(v))$. Addieren von 1 aufgrund der Rotation ergibt amortisierte Kosten $\leq 3(r(v)' - r(v)) + 1$.

zig-zig: Es müssen zwei Rotationen ausgeführt werden. Deshalb entstehen amortisierte Kosten von

$$\begin{aligned} & 2 + r(u)' + r(v)' + r(x)' - r(u) - r(v) - r(x) , \text{mit } r(x) = r(v)' \\ & = 2 + r(u)' + r(x)' - r(u) - r(v) , \text{mit } r(v)' \geq r(u)' \wedge r(u) \geq r(v) \\ & \leq 2 + r(v)' + r(x)' - 2r(v) \end{aligned}$$

Nun wird zunächst die Behauptung aufgestellt, dass dieser Ausdruck klein genug ist. Dies wird dann über Äquivalenzen gezeigt:

$$\begin{aligned}
& 2 + r(v)' + r(x)' - 2r(v) \leq 3(r(v)' - r(v)) \\
\Leftrightarrow & 2 \leq 2r(v)' - r(x)' - r(v) \\
\Leftrightarrow & -2 \geq -2r(v)' + r(x)' + r(v) \\
\Leftrightarrow & -2 \geq \log_2(tw(x')/tw(v')) + \log_2(tw(v)/tw(v'))
\end{aligned}$$

Dass die letzte Ungleichung gilt, kann an einer Eigenschaft des \log_2 abgeleitet werden. Für $a, b \in R$ mit $a, b > 0$ und $a + b \leq 1$ gilt $\log_2(a) + \log_2(b) \leq -2$. An Abbildung 37 ist zu erkennen, dass sich $tw(v)$ vom Ausgangszustand zum gestrichelt dargestellten Zwischenschritt hin nicht verändert. $tw(x')$ ist ebenfalls unverändert zum Zwischenschritt. Es kann also bei beiden Knoten mit den Werten aus dem Zwischenschritt gearbeitet werden. Bezeichne u^* den Knoten u im Zwischenschritt, dann gilt $tw(u^*) = tw(v')$.

Daraus folgt $tw(v') = tw(x') + tw(v) + w(u)$ und daraus $(tw(x') + tw(v))/tw(v') < 1$ und mit der Eigenschaft von \log_2 folgen die Ungleichungen.

zig-zag:

$$\begin{aligned}
& 2 + r(u)' + r(v)' + r(x)' - r(u) - r(v) - r(x) , \text{mit } r(x) = r(v)' \wedge r(v) \leq r(u) \\
\leq & 2 + r(u)' + r(x)' - 2r(v)
\end{aligned}$$

Nun wird wie bei zig-zig vorgegangen:

$$\begin{aligned}
& 2 + r(u)' + r(x)' - 2r(v) \leq 2(r(v)' - r(v)) \\
\Leftrightarrow & 2 \leq 2r(v)' - r(x)' - r(u)' \\
\Leftrightarrow & -2 \geq -2(v)' + r(x)' + r(u)' \\
\Leftrightarrow & -2 \geq \log_2(tw(x')/tw(v')) + \log_2(tw(u')/tw(v'))
\end{aligned}$$

Mit der \log_2 Eigenschaft aus zig-zig und Abbildung 37 folgt die Behauptung, wobei diesmal lediglich der Endzustand betrachtet werden muss. Zum Berechnen der Kosten der Gesamtoperation werden die Kosten der einzelnen Fallbehandlungen summiert. Dabei bildet sich eine Teleskopsumme, die möglicherweise, wenn ein zig, bzw. zag Fall enthalten ist, mit eins addiert werden muss. Daraus folgt das Lemma. \square

6.3 Dynamische Optimalitäts Vermutung

Der Splay Baum erfüllt die Working Set Property und die Dynamic Finger Property aus Abschnitt 4.5 und auch noch viele weitere in dieser Arbeit nicht

aufgeführte Eigenschaften. Der Beweis zur Dynamic Finger Property ist sehr aufwändig [9]. Dass der Splay Baum die Working Set Property besitzt, wurde bereits bei seiner Vorstellung gezeigt [1]. Dieser Beweis wird hier vorgestellt. Die anderen Eigenschaften (außer dynamisch optimal) aus Kapitel 4.5 folgen dann aus diesen beiden. Aufgrund dieser oberen Schranken der Laufzeit des Splay Baumes wurde die Vermutung aufgestellt, dass der Splay Baum dynamisch optimal ist. Bewiesen ist bisher nur, dass er $\log(n)$ -competitive ist. Dies folgt aus dem Access Lemma. Würde für eine solche obere Schranke gezeigt werden, dass der Splay Baum diese nicht einhalten kann, jedoch ein anderer BST schon, wäre die Vermutung zur dynamischen Optimalität widerlegt. Auch das ist bis heute nicht geschehen.

Wiederholung der Definition von a_i zu einer Zugriffsfolge $X = x_1, x_2, \dots, x_m$ aus Kapitel 4.5:

Für x_i sei $J_i = \{j \in \mathbb{N} | j < i \wedge x_j = x_i\}$. Sei $t_{xi} = \max(J_i)$, falls J_i nicht leer ist, ansonsten $t_{xi} = 0$. t_{xi} liefert also den Index des vorherigen Zugriffes auf x_i , falls ein solcher existiert. Sei $a_i = |\{x_j | t_{xi} < j \leq i\}|$.

Working Set Property 6.1. *Es sei T ein Splay Baum mit n Knoten. Sei $X = x_1, x_2, \dots, x_m$ eine für T erstellte Zugriffsfolge. Dann gilt für die amortisierten Kosten zum Ausführen von X :*

$$O(n \log(n) + m + \sum_{i=1}^m \log(a_i)).$$

Beweis. Den Knoten werden die Gewichte $1, 1/2^2, 1/3^2, \dots, 1/n^2$ zugeordnet. Diesmal ist die Zuordnung nicht fest. Nach jeder *access* Operation können sich Gewichte ändern. Sei i der kleinste Index mit $x_i = \text{key}(v)$ für einen Knoten v . Sei $b = |\{x_l | l < i\}|$, dann wird v zum Start das Gewicht $1/2^{b+1}$ zugeordnet. Auf Knoten, auf deren Schlüssel nicht zugegriffen wird, verteilen sich die kleinsten Gewichte beliebig. $tw(v)$ und Φ sind definiert wie beim Access Lemma.

Nach einer *access* (x_j) Operation werden die Gewichte neu zugeordnet. Der Knoten v_j mit dem Schlüssel x_j erhält das Gewicht $1/1$. Sei $1/k^2$ das Gewicht von v_j vor *access* (x_j). Sei u ein Knoten mit $w(u) = 1/k^*$ mit $k^* \in \{1, 4, \dots, (k-1)^2\}$ direkt vor *access* (x_j). Dann ist $1/(k^* + 1)^2$ das Gewicht von u nach *access* (x_j). Die Gewichte der restlichen Knoten bleiben unverändert. Zu beachten ist, dass nach einer solchen Neuzuordnung die Menge der im Baum enthaltenen Gewichte unverändert bleibt.

Diese Verteilung der Gewichte garantiert, dass direkt vor *access* (x_j), v_j ein Gewicht von $1/a_i^2$ hat, somit gilt $tw(v_j) \geq 1/a_i^2$. Der Wert von $tw()$ für die Wurzel von T ist $W = \sum_{i=1}^n 1/n^2 < 2 = O(1)$. Diese Werte in das Access Lemma eingesetzt, ergibt amortisierte Kosten von

$$O(\log(1/(1/a_i^2))) = O(\log(a_i^2)) = O(\log(a_i)).$$

Durch die nachfolgende Neuzuordnung der Gewichte kann Φ nur kleinere Werte annehmen. Denn nur das Gewicht der neuen Wurzel erhöht sich. $tw()$ ist für die Wurzel aber konstant.

Der Rang eines Knotens kann über die gesamte Zugriffsfolge nicht um mehr als $\log_2(n^2W) = O(\log(n))$ kleiner werden. Denn W ist der maximale Wert von $tw()$ und der minimale ist $1/n^2$. Daraus folgt eine maximale Verringerung des Potentials von T von $O(n \log(n))$. \square

7 Weitere dynamische Suchbäume

Hier werden kurz zwei Variationen zum Tango Baum vorgestellt. Zum einen der Zipper Baum [10]. Er ist ebenfalls $\log(\log(n))$ -competitive, garantiert aber auch $O(\log(n))$ im worst case, bei einer einzelnen *access* Operation. n steht wieder für die Anzahl der Knoten. Zum anderen der Multisplay Baum [11]. Bei diesem wird ein preferred path durch einen Splay Baum repräsentiert. Amortisiert betrachtet erreicht er $O(\log(n))$ bei *access* und ist $\log(\log(n))$ -competitive.

7.1 Zipper Baum

Der Zipper Baum basiert auf dem Tango Baum und nutzt auch preferred paths aus einem Referenzbaum P . Aufbau und Pflege der preferred paths in P unterscheiden sich nicht zu denen beim Tango Baum. Ihre Repräsentation im eigentlichen BST T macht den wesentlichen Unterschied zu einem Tango Baum aus. Abbildung 39 stellt eine solche beispielhaft dar. Sei v ein Knoten in T , dann ist in diesem Kapitel v^* der Knoten in P mit $key(v) = key(v^*)$. Die Repräsentation eines preferred paths $P_p = (p_1^*, p_2^*, \dots, p_m^*)$ in T stellt einen Hilfsbaum H dar, der in zwei Teile unterteilt ist, dem **zipper** und dem **bottom tree**. Der bottom tree ist ein balancierter BST, der genau die Schlüssel enthält, die in P_p enthalten sind, jedoch nicht im zipper. Der zipper besteht aus zwei Teilen, dem **top zipper** z_t und dem **bottom zipper** z_b . z_t und z_b dürfen jeweils maximal $\log_2(\log_2(n))$ Knoten enthalten. Gemeinsam enthalten sie zumindest $\log_2(\log_2(n))/2$ Knoten, wenn ein bottom tree existiert. Bei weniger als $\log_2(\log_2(n))/2$ Knoten im Hilfsbaum existiert kein bottom tree. Es wird im Folgenden angenommen, dass ein bottom tree existiert.

P_p wird in **zig Segmente** und **zag Segmente** unterteilt. Ein zig Segment in P_p ist ein längst möglicher Pfad in P , indem nur Knoten aus P_p enthalten sein dürfen, deren rechtes Kind ebenfalls in P_p enthalten ist. zag Segmente werden analog gebildet, wobei das linke Kind ebenfalls in P_p enthalten sein

muss, anstatt dem rechten. Dazu gibt es noch die Ausnahme, dass der Knoten mit der größten Tiefe in P_p dem Segment seines Elternknotens zugeordnet wird.

Sei S_{zig} die Folge der in zig Segmenten enthaltenen Knoten, aufsteigend sortiert nach der Tiefe und S_{zag} die Folge der in zag Segmenten enthaltenen Knoten, aufsteigend sortiert nach der Tiefe. Sei n_t , bzw. n_b die Anzahl der Knoten von z_t , bzw. z_b . z_t repräsentiert Knoten $P_t = (p_1^*, p_2^*, \dots, p_{n_t}^*)$ und z_b die Knoten $P_b = (p_{n_t+1}^*, p_{n_t+2}^*, \dots, p_{n_t+n_b}^*)$. Sei t_l^* der Knoten der größten Tiefe, der sowohl in P_t , als auch in S_{zig} enthalten ist. Sei t_r^* der Knoten mit der größten Tiefe, der sowohl in P_b , als auch in S_{zag} enthalten ist. t_l ist die Wurzel von H . t_r ist das rechte Kind von t_l . Der linke Teilbaum von t_l hat Listenform und enthält die Knoten, deren Schlüssel auch in S_{zig} enthalten ist, so dass kein Knoten in diesem Teilbaum ein linkes Kind hat. Der rechte Teilbaum von t_r hat Listenform und enthält die Knoten, deren Schlüssel auch in S_{zag} enthalten ist, so dass kein Knoten in diesem Teilbaum ein rechtes Kind hat. z_b wird analog aus P_b erzeugt und seien b_l und b_r die Knoten in z_b entsprechend zu t_l und t_r in z_t . b_l ist das linke Kind von t_r . Die Wurzel des bottom trees ist das linke Kind von b_r . Dass die Links-Rechts-Beziehung eingehalten wird ergibt sich aus dem Aufbau der zig und zag Segmente. Es ist leicht zu erkennen, dass die Wurzel des bottom trees in konstanter Zeit erreicht werden kann.

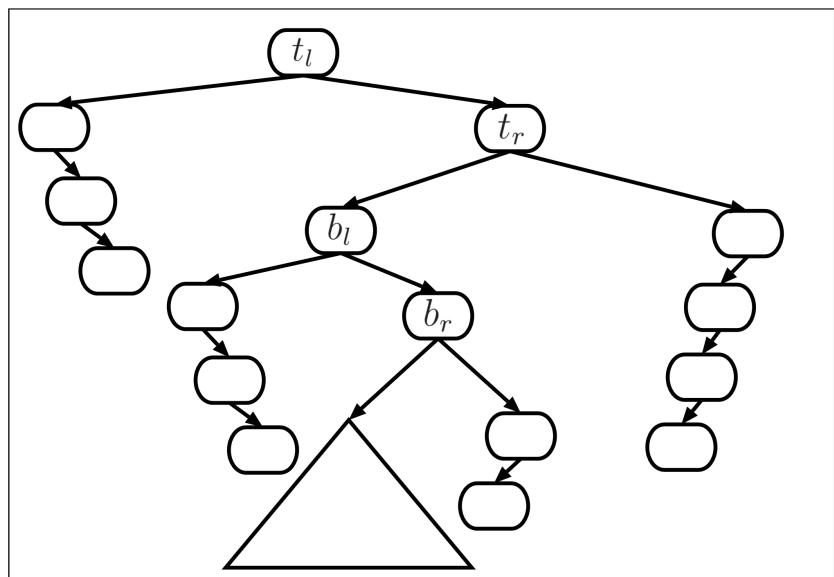


Abbildung 39: Pfadrepräsentation beim Zipper Baum, basiert auf einer Abbildung aus [10].



Abbildung 40: zig Segmente sind grün dargestellt. zag Segmente blau.

Besonderheiten bei *access*: Sei l die Anzahl der Knoten des top zipper. Beim Suchen nach einem Schlüssel in einem Hilfsbaum H wird dessen top zipper soweit wie notwendig in einen Pfad gewandelt, der (p_1, p_2, \dots, p_l) entspricht. Ist der top zipper vollständig in einen Pfad umgewandelt, wird der Vorgang beim bottom zipper fortgesetzt. Dieser hat dann die Stellung des top zipper. Außerdem wird dann ein **Extraktionsprozess** angestoßen, der $\log(\log(n))$ Knoten aus dem bottom tree auslagert, um einen neuen bottom zipper zu erzeugen. Für dessen Laufzeit ist entscheidend, dass die Wurzel des bottom trees in konstanter Zeit erreicht werden kann. Ein Extraktionsprozess ist in Abbildung 41 dargestellt. l' ist der größte Schlüssel, der kleiner ist, als die Schlüssel der zu extrahierenden Knoten. r' ist entsprechend der kleinste Schlüssel, der größer ist, als diese Schlüssel.

Der nächste Knoten aus dem zipper kann dem Pfad in konstanter Zeit hinzugefügt werden. Ist der gesuchte Knoten p_v im zipper enthalten, entstehen um p_v^* in P zu erreichen, asymptotisch betrachtet die gleichen Kosten, wie in H . Ist in H der zipper aufgebracht, sind bereits Kosten von $O(\log(\log(n)))$ entstanden. Befindet sich der gesuchte Knoten im bottom tree entstehen in H ebenfalls Kosten von $O(\log(\log(n)))$. Diese müssen auch in P entstehen, da dann $v > \log_2(\log_2(n))/2$ gelten muss. Deshalb entstehen asymptotisch

betrachtet in H und innerhalb P die gleichen Kosten. In P ist jeder Knoten in $O(\log(n))$ Zeit erreichbar, deshalb ist dies auch für T der Fall.

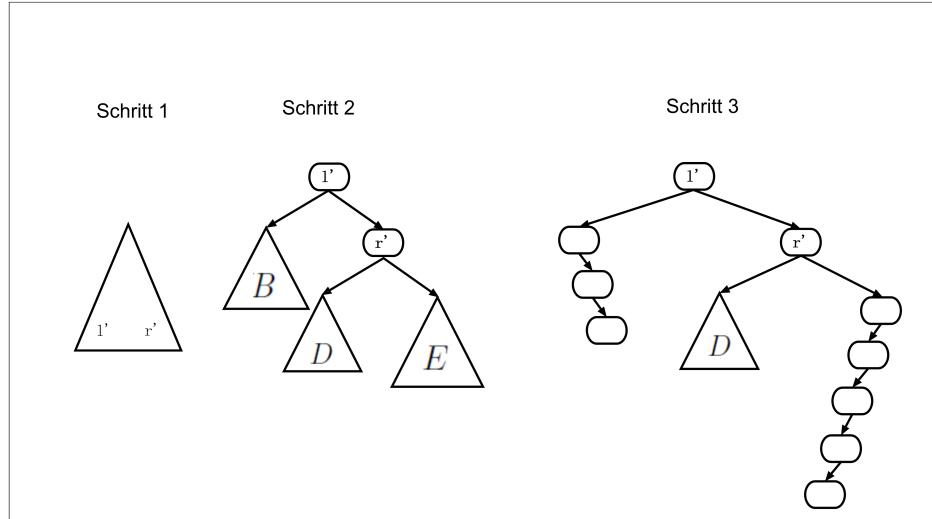


Abbildung 41: Extraktionsprozess beim Zipper Baum.

7.2 Multisplay Baum

Ein preferred path wird hier durch einen Splay Baum dargestellt, um dessen Laufzeiteigenschaften nutzen zu können. Da der Splay Baum kein balancierter BST ist, gibt es zusätzliche mögliche Zustände im Vergleich zu einem Tango Baum mit der gleichen Knotenzahl. Der Hilfsbaum mit der Wurzel des Gesamtbaumes in Abbildung 43, ist beispielsweise zu einer Liste entartet. Zu den genannten Eigenschaften bezüglich der Laufzeit sind Beweise in [11] enthalten. Der Multisplay Baum erfüllt die Working Set Property [12].



Abbildung 42: Referenzbaum mit grün gezeichneten preferred paths.



Abbildung 43: Beispielhafter Multisplay Baum zum Referenzbaum aus Abbildung 42.

Die *access* Operation beim Multisplay Baum: Zu beachten ist, dass jede BST Darstellung auch eine Splay Baum Darstellung ist. Anders als beim Tango oder Zipper Baum muss ein neu erzeugter Hilfsbaum also nicht so angepasst werden, dass er weitere Invarianten einhält. Nach einer *access*(k) Operation ist der Knoten v_k mit dem Schlüssel k die Wurzel des Multisplay Baumes T . Zunächst wird eine gewöhnliche Suche in T durchgeführt, bis der Zeiger p der Operation auf v_k zeigt. Ist v_k gefunden, werden die Pfadrepräsentationen aktualisiert. Hierzu muss der Hilfsbaum, der die Wurzel von T enthält neu erzeugt werden. *cut* und *join* werden beim Multisplay Baum zu *switch* zusammengefasst. Zum Erzeugen des neuen Hilfsbaumes mit der Wurzel des Gesamtbaumes sind so viele *switch* Operationen notwendig, wie es Wechsel bei preferred children gab.

8 Dokumentation zur Implementierung

In diesem Kapitel wird die Implementierung zum Tango Baum beschrieben. Implementiert wurde der Tango Baum mit dem Rot-Schwarz-Baum als Hilfsdatenstruktur. Außerdem wurde der Splay Baum implementiert, um Laufzeittests zwischen diesem und dem Tango Baum durchführen zu können. Bedient werden kann das Programm über eine einfach gehaltene graphische Oberfläche. Das Programm wurde mit Java 8 übersetzt und als IDE wurde Apache NetBeans 12.0 verwendet.

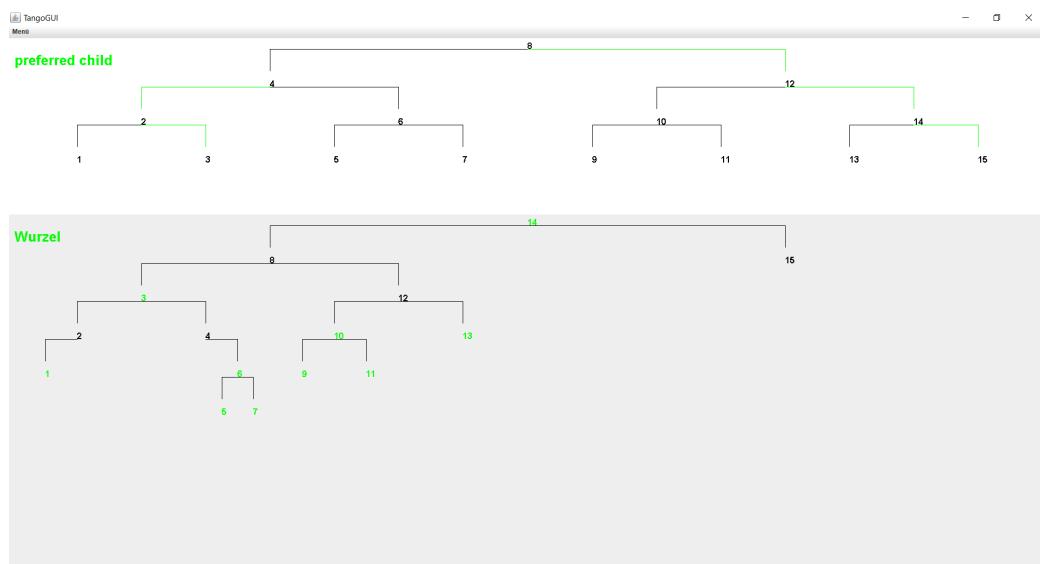


Abbildung 44: Screenshot des Hauptfensters der GUI. Im oberen Teil befindet sich der Referenzbaum. Im unteren Teil der dazugehörige Tango Baum.

Abbildung 44 zeigt das Hauptfenster. Oben ist ein Referenzbaum zu einem Tango Baum mit 15 Knoten dargestellt, unten der Tango Baum. Preferred children und die Wurzeln von Hilfsbäumen sind grün dargestellt. Diese Ansicht dient dazu, die Beziehung zwischen dem Referenzbaum und dem Tango Baum darzustellen.

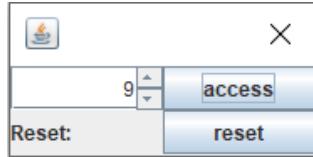


Abbildung 45: Mit diesem Fenster können *access* Operationen auf dem Tango Baum des Hauptfensters angestoßen werden.

Mit dem Menüpunkt „access“ wird das Fenster aus Abbildung 45 geöffnet. Mit diesem werden *access* Operationen angestoßen. Außerdem können die Bäume damit zurückgesetzt werden.

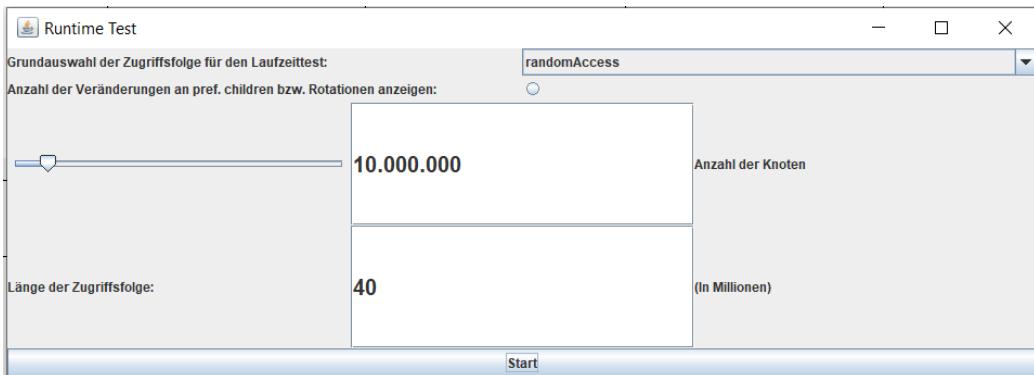


Abbildung 46: Mit diesem Fenster kann ein Laufzeittest ausgewählt, parametriert und angestoßen werden.

Mit dem Menüpunkt „RuntimeTest“ wird das Fenster aus Abbildung 46 geöffnet. Mit diesem können Laufzeittests zwischen dem Tango Baum und dem Splay Baum durchgeführt werden. Auf die Parameter und den Aufbau der Zugriffsfolgen wird im Abschnitt zu den Laufzeittests eingegangen. Ungültige Werte, also z. B. eine negative Anzahl an Knoten, werden von den Eingabeelementen nicht angenommen, sondern mit dem letzten gültigen Wert überschrieben.



Abbildung 47: Mit diesem Fenster wird das Resultat eines Laufzeittests dargestellt.

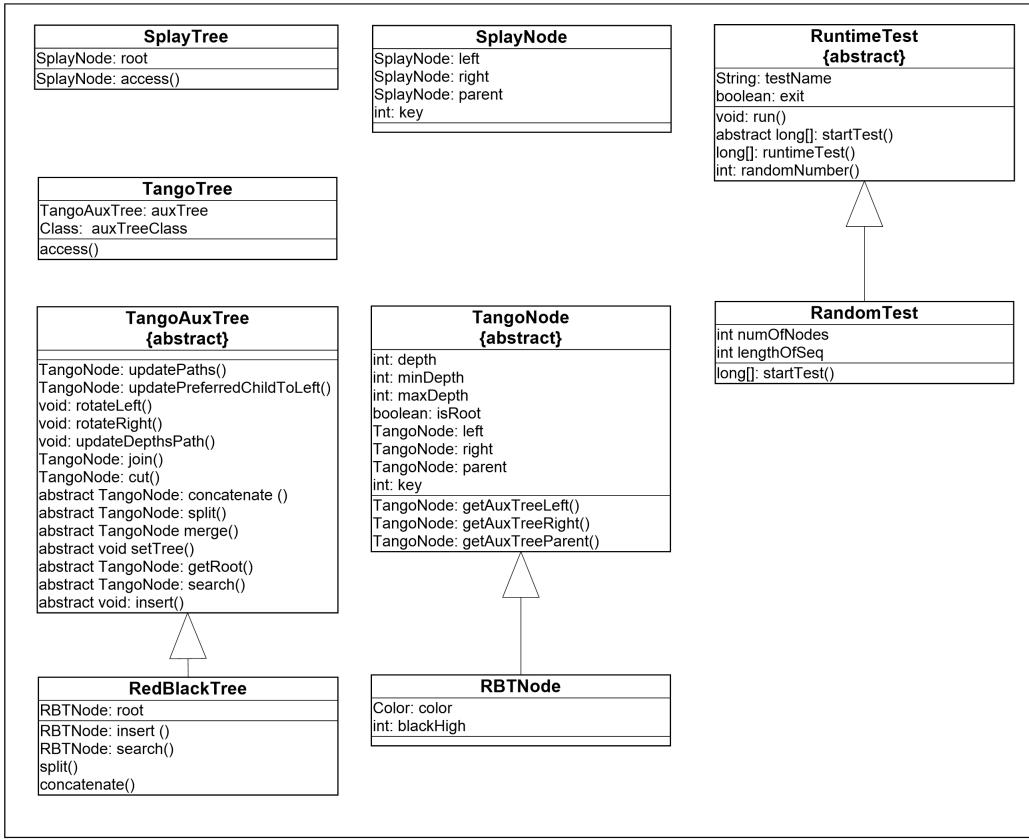


Abbildung 48: Wesentliche Klassen der Implementierung. Methoden zum direkten Lesen, bzw. Schreiben von Attributen sind nicht dargestellt.

8.1 Beschreibung der Klassen

Abbildung 48 stellt das Klassendiagramm zur Implementierung dar. Die einzelnen Klassen werden nun noch genauer beschrieben.

SplayTree und SplayNode: Der Splay Baum startet genau wie der Tango Baum vollständig balanciert, obwohl sich dies bei längeren Zugriffssfolgen praktisch nicht auswirken sollte. Ansonsten gibt es keine Besonderheiten. `access` verhält sich genauso wie im Kapitel zum Splay Baum beschrieben.

TangoNode: Der TangoNode enthält bereits alle zwingend notwendigen Attribute eines Knotens im Tango Baum.

TangoAuxTree: Klassen, deren Objekte als Hilfsbaum im Tango Baum eingesetzt werden sollen, müssen diese Klasse erweitern. `setTree` wird benö-

tigt, da von der Klasse TangoTree abgeleitete Objekte ihre BST Struktur nur über das Attribut „auxTree“ erreichen. Gibt es eine Veränderung an der Wurzel des Tango Baumes, wird die BST Struktur von „auxTree“ neu gesetzt. Da ein Objekt das einen BST repräsentiert die eigentliche BST Struktur im Normalfall nur über ein Attribut *root* erreicht, sollte diese zusätzliche Anforderung einfach zu erfüllen sein. *updateDepthsPath* pflegt die Attribute „minDepth“ und „maxDepth“ der TangoNode.

TangoTree: „auxTree“ macht die Wurzel des Tango Baumes erreichbar. Außerdem können über dieses Attribut die *split* und *concatenate* Operationen aufgerufen werden. „auxTreeClass“ entspricht der Klasse, deren Objekte als Hilfsbäume eingesetzt werden. Diese wird dem Constructor übergeben. Somit kann der RBT einfach durch eine andere geeignete Struktur ersetzt werden.

RedBlackTree und RBTreeNode: Diese erweitern die abstrakten Klassen TangoAuxTree und TangoNode. RedBlackTree verhält sich wie im Kapitel zum Rot-Schwarz-Baum beschrieben.

RuntimeTest: Klassen deren Objekte einen Laufzeittest mit dem Tango Baum und dem Splay Baum durchführen sollen, müssen diese Klasse erweitern. Das Attribut „exit“ wird über die GUI gesetzt, worauf hin ein gestarteter Test abgebrochen wird. Die Methode *runtimeTest* führt den eigentlichen Test aus und gibt ein Array mit der maximalen Länge vier zurück. Dieses enthält dann die Ausführungszeiten des Tango Baumes und des Splay Baumes, sowie die Anzahl der Veränderungen bei preferred Children im Referenzbaum zum Tango Baum und die Anzahl der vom Splay Baum ausgeführten Rotationen. Um Programmabbrüchen aufgrund zu wenig Speicher vorzubeugen, wurde bei den Projekteigenschaften die Option „ -Xmx4096m“ gesetzt, Abbildung 49. Dies wirkt sich jedoch nur beim Starten der Applikation aus der IDE heraus aus. Wird die Applikation ohne IDE gestartet, muss die Standardeinstellung der VM ggf. angepasst werden. Für Windows Systeme ist dazu der Abgabe bereits eine Batch-Datei angefügt, bei der 4096 MB voreingestellt sind.

RandomTest: Diese Klasse erweitert die Klasse RuntimeTest. Insgesamt sind in dem Projekt sechs solcher Klassen vorhanden. Prinzipiell unterscheiden diese sich jedoch nicht, weshalb hier nur exemplarisch auf eine dieser Klassen eingegangen wird. Die Methode *startTest* erzeugt eine Zugriffsfolge, bei der jeder enthaltene Schlüssel durch einen Pseudozufallsgenerator ausgewählt wurde. Der Test selbst wird dann von der bereits in der Klasse

RuntimeTest implementierten Methode *runtimeTest* ausgeführt.

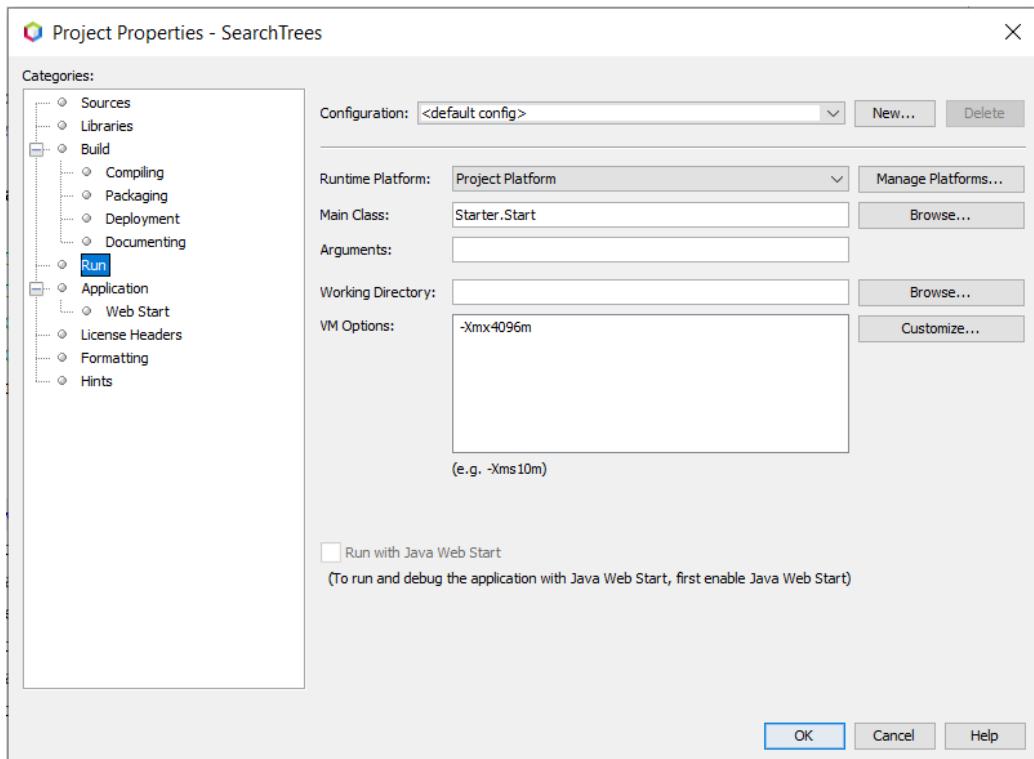


Abbildung 49: Den zur Ausführung verwendbaren Speicher auf 4096MB erweitern.

9 Laufzeittests mit dem Tango Baum und dem Splay Baum.

Es werden Laufzeittests zu unterschiedlichen Arten von Zugriffsfolgen durchgeführt. Unter anderem solche, die auf eine der Eigenschaften Static Finger, Working Set oder Dynamic Finger aus Abschnitt 4.5 zugeschnitten sind. Hierzu werden dann Zugriffsfolgen verwendet, bei denen die jeweilige Eigenschaft für die Laufzeit besonders relevant ist. Der Tango Baum erfüllt die aufgeführten Eigenschaften zwar nicht, jedoch ist er $\log(\log(n))$ -competitive. Ein dynamisch optimaler BST muss diese Eigenschaften erfüllen. Dadurch könnte auch der Tango Baum von den speziellen Eigenschaften der Zugriffsfolgen indirekt profitieren.

9.1 Verwendete Zugriffsfolgen zu den Laufzeittests.

Hier werden die verwendeten Zugriffsfolgen zunächst kurz vorgestellt. Auf Details zum Aufbau wird dann im Abschnitt zu dem jeweiligen Test eingegangen. n entspricht der Anzahl der Knoten und m der Länge der Zugriffsfolge. Als Schlüsselmenge wird $K = \{1, 2, \dots, n\}$ verwendet. Existiert keine Abhängigkeit von m zu n , so haben die Zugriffsfolgen die Länge 40 Millionen.

1. Zufällig erzeugte Zugriffsfolgen.

2. Die Bit Reversal Permutation:

Es werden mehrere solcher in Abschnitt 4.3 beschriebenen Zugriffsfolgen mit unterschiedlich langen Binärdarstellungen der Schlüssel verwendet.

3. Zugriffsfolgen zur Static Finger Property:

Hier wird ein Schlüssel $k_F \in K$ gewählt. Die Häufigkeit des Vorkommens eines Schlüssels $k \in K$ in X ist von $|k - k_F|$ abhängig. Schlüssel zu denen dieser Wert klein ist, sind häufiger enthalten als andere. Es werden solche Zugriffsfolgen für Schlüsselmengen mit unterschiedlicher Kardinalität erstellt und die Ergebnisse der beiden Datenstrukturen verglichen. Zusätzlich werden noch Zugriffsfolgen X_1, X_2, \dots, X_7 verwendet, so dass die Static Finger Property mit steigendem Index der Zugriffsfolge relevanter wird. Bei diesen sieben Folgen ist der Wert für n konstant.

4. Zugriffsfolgen zur Dynamic Finger Property:

Hier werden Zugriffsfolgen verwendet, so dass $|x_{i+1} - x_i|$ für alle $i \in \{1, 2, \dots, m-1\}$ einen festen parametrierbaren Wert ergibt. Zusätzlich werden noch Zugriffsfolgen X_1, X_2, \dots, X_7 verwendet, so dass die Dynamic Finger Property mit steigendem Index der Zugriffsfolge weniger relevant wird.

5. Zugriffsfolgen zur Working Set Property:

In den hier verwendeten Zugriffsfolgen wiederholen sich die Schlüssel mit einem festen parametrierbaren Abstand. Um Einflüsse der anderen aufgeführten Eigenschaften niedrig zu halten, werden die verwendeten Schlüssel betragsmäßig über die gesamte Schlüsselmenge verteilt. Außerdem gilt für zwei aufeinanderfolgende Schlüssel k_1 und k_2 in den meisten Fällen $|k_1 - k_2| = n/2$. Zusätzlich werden noch Zugriffsfolgen X_1, X_2, \dots, X_7 verwendet, mit denen analog zur Dynamic Finger Property vorgegangen wird.

6. Alternierend die Schlüssel 1 und n .
7. Aufsteigend sortiert:
Jeder Schlüssel wird nur einmal verwendet.
8. Absteigend sortiert:
Jeder Schlüssel wird nur einmal verwendet.
9. Alternierend aufsteigend und absteigend sortiert:
Jeder Schlüssel wird zweimal verwendet.

9.2 Durchführung der Laufzeittests.

Hier werden die verwendeten Zugriffsfolgen im Detail beschrieben. Außerdem werden die Ergebnisse der Laufzeittests vorgestellt.

9.2.1 Zufällig erzeugte Zugriffsfolgen

Ein Pseudozufallsgenerator erstellt die hier verwendeten Zugriffsfolgen. Für jedes $i \in \{1, 2, \dots, m\}$ und $k \in K$ ist die Wahrscheinlichkeit, dass $x_i = k$ gilt, gleich $1/n$. Es werden jeweils zehn solche Zugriffsfolgen pro verwendeter Schlüsselmenge getestet.

Als erstes werden die Zeiten zu den Zugriffsfolgen dargestellt, bei denen der Tango Baum die kürzesten Laufzeiten erreichte:

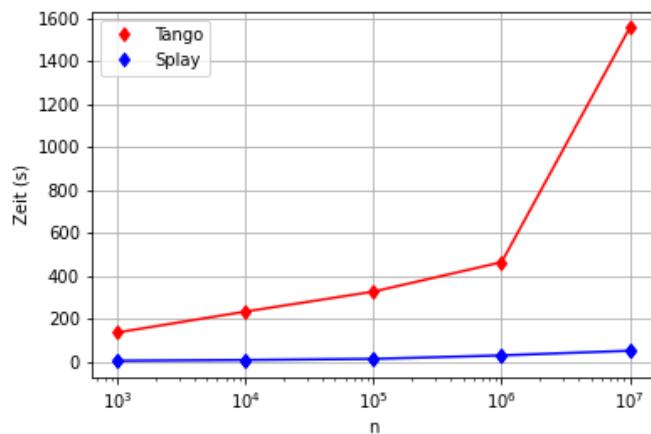


Abbildung 50: Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Tango Baumes.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	138, 2	6, 5
10^4	235, 6	9, 9
10^5	328, 6	15, 1
10^6	465, 7	31, 4
10^7	1557, 3	53

Tabelle 2: Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Tango Baumes.

Hier werden die Zeiten zu den Zugriffsfolgen dargestellt, bei denen der Splay Baum die kürzesten Laufzeiten erreichte:

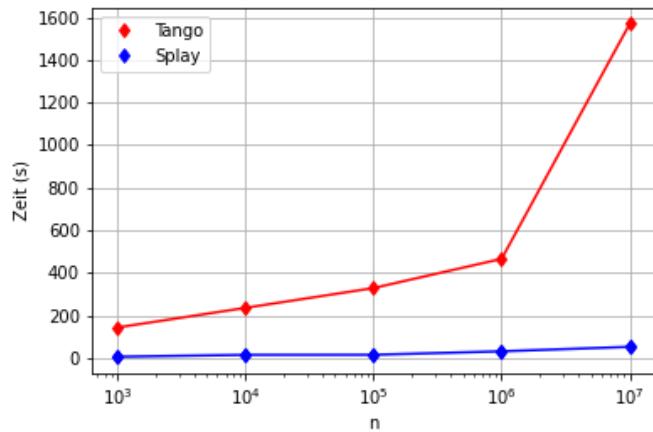


Abbildung 51: Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Splay Baumes.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	143, 4	6
10^4	235, 6	9, 9
10^5	328, 6	15, 1
10^6	465, 7	31, 4
10^7	1569, 3	52, 5

Tabelle 3: Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Splay Baumes.

Hier werden die durchschnittlichen Werte über jeweils zehn Zugriffsfolgen dargestellt:

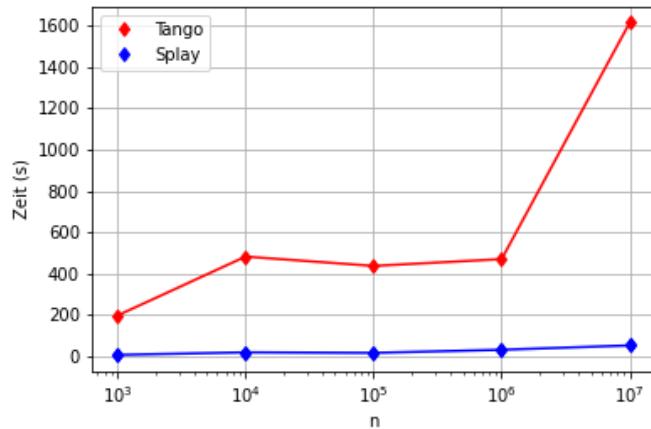


Abbildung 52: Durchschnittliche Laufzeiten zu jeweils zehn zufällig erzeugten Zugriffsfolgen.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	198, 3	8
10^4	483, 1	20.3
10^5	438	17, 7
10^6	470, 8	32, 7
10^7	1613, 5	53, 7

Tabelle 4: Durchschnittliche Laufzeiten zu jeweils zehn zufällig erzeugten Zugriffsfolgen.

Beide erreichen bei diesen Zugriffsfolgen im Vergleich zu den meisten noch Folgenden hohe Werte bei der Laufzeit. Da es sich um dynamische BSTs handelt, ist dies auch zu erwarten. Der Zustand der dynamischen BSTs wird im Wesentlichen durch die bereits ausgeführten *access* Operationen bestimmt. Bei den hier verwendeten Zugriffsfolgen gibt es keine Abhängigkeiten zwischen dem Parameter einer *access* Operation, zu denen von bereits ausgeführter *access* Operationen. Somit kann durch die Anpassung des Zustandes kein genereller Vorteil generiert werden.

Jetzt wird noch auf die deutlichen Differenzen zwischen den Werten des Tango Baumes und des Splay Baumes eingegangen. Die Laufzeit des Tango Bau-

mes ist im Wesentlichen von der Gesamtanzahl der Veränderungen bei preferred children n_T in seinem Referenzbaum abhängig. Die Laufzeit des Splay Baumes ist im Wesentlichen von der Anzahl der ausgeführten Rotationen n_S abhängig. Die Tabelle 5 stellt die durchschnittlichen Werte von n_T und n_S über jeweils zehn Zugriffsfolgen gegenüber.

n	n_T	n_S	n_T/n_S
10^3	167.247.550	456.650.373	0,37
10^4	233.092.762	649.066.234	0,36
10^5	299.550.021	842.015.806	0,36
10^6	367.122.698	1.034.934.019	0,35
10^7	439.981.567	1.226.490.084	0,36

Tabelle 5: Die Spalte n_T enthält die durchschnittliche Anzahl der Veränderungen bei preferred children im Referenzbaum zum Tango Baum über zehn Zugriffsfolgen. Die Spalte n_S enthält die durchschnittliche Anzahl der Rotationen beim Splay Baum über zehn Zugriffsfolgen.

In der Spalte $a = n_T/n_S$ gibt es keinen Wert $< 1/3$. Durchschnittlich muss der Splay Baum über die jeweils zehn Zugriffsfolgen, also maximal drei Rotationen pro Veränderung an einem preferred child durchführen. Der Tango Baum muss pro Veränderung an einem preferred child eine *cut* und eine *join* Operation ausführen. Dies führt im allgemeinen Fall zu vier *split* und vier *concatenate* Operationen, die ausgeführt werden müssen. Für $n = 1000$ hat der Referenzbaum des Tango Baumes bereits zehn Ebenen, was in den meisten Fällen dazu führen wird, dass der Hilfsbaum mit der Wurzel des Tango Baumes neun oder zehn Knoten enthält. Es kann davon ausgegangen werden, dass das Ausführen der jeweils vier *split* und *concatenate* Operationen deutlich aufwändiger ist, als das Ausführen der drei Rotationen, zumal aufgrund der *insertFixup* Operation auch beim Tango Baum mehrere Rotationen pro Veränderung bei einem preferred child möglich sind. Abbildung 53 stellt den Zusammenhang von n_T und n_S zu n graphisch dar.

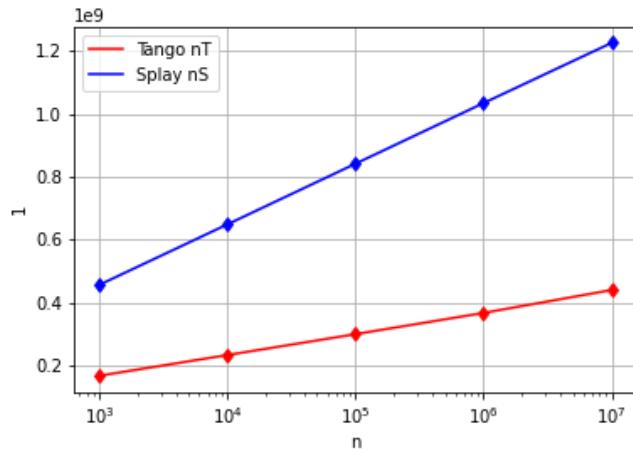


Abbildung 53: Hier sind n_T und n_S graphisch dargestellt.

Der Aufwand, der nach einer Veränderung bei einem preferred child im Referenzbaum notwendig ist, um die Pfadrepräsentationen zu aktualisieren und die rote Kennlinie in Abbildung 53 lassen annehmen, dass der Tango Baum für Schlüsselmengen konstruiert wurde, deren Kardinalitäten deutlich größer sind, als die hier darstellbaren.

Allerdings ist n_T/n_S bei den hier durchgeführten Tests in etwa konstant. Für die Kosten zum Aktualisieren einer Pfadrepräsentation im Tango Baum gilt $O(\log(\log(n)))$ und die Kosten zum Ausführen einer Rotation sind unabhängig von n . Kombiniert mit der bewiesenen $\log(\log(n))$ -competitiveness des Tango Baumes ergibt sich hier ein (vorsichtiger) Hinweis auf die dynamische Optimalität des Splay Baumes.

9.2.2 Bit Reversal Permutations

Auf der X-Achse wird die Länge der Binärdarstellung der Schlüssel l dargestellt. In der Spalte 2^l kann jeweils die Länge der Zugriffsfolge und die Anzahl der Knoten abgelesen werden.

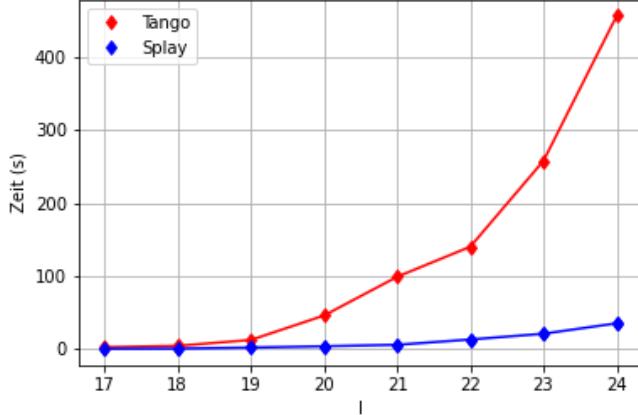


Abbildung 54: Ergebnisse zu den Laufzeittests zur Bit Reversal Permutation.

l	2^l	Zeit Tango in (s)	Zeit Splay in (s)
17	131.072	2,2	0,1
18	262.144	4,1	0,3
19	524.288	12,1	1,7
20	1.048.576	45,7	3,5
21	2.097.152	98,9	5,5
22	4.194.304	140,3	12,8
23	8.388.608	258,2	20,7
24	16.777.216	457,3	35

Tabelle 6: Ergebnisse zu den Laufzeittests zur Bit Reversal Permutation.

Wie in Abschnitt 4.3 gezeigt, gilt bei diesen Folgen für die Laufzeit eines beliebigen BSTs, $\Omega(n \log(n))$. Der Splay Baum erfüllt die Balanced Property, weshalb für seine Laufzeit auch $O(n + n \log(n))$ gelten muss. Die Ergebnisse bestätigen, dass es sich um aufwändige Zugriffsfolgen für BSTs handelt. Vergleichbare Werte traten nur bei den zufällig erzeugten Zugriffsfolgen auf.

9.2.3 Static Finger Property

Sei $a = \lfloor n/2 \rfloor$. a ist der Parameter bei 2 Prozent der *access* Operationen. Auf $a+1$ und $a-1$ entfallen dann 1 Prozent (gemeinsam 2 Prozent) der restlichen *access* Operationen. Dieses Vorgehen iteriert bis ein Prozent der Anzahl der

verbleibenden *access* Operationen weniger als 1 ergibt. Es wird dann nochmals die Anzahl von Zugriffen auf a hinzugefügt, die benötigt wird, um die gewünschte Länge der Zugriffsfolge zu erreichen. Die Anordnung der Schlüssel innerhalb der Zugriffsfolge erfolgt über einen Pseudozufallsgenerator. Da ein Pseudozufallsgenerator verwendet wird, werden zu jeder Schlüsselmenge wieder zehn Zugriffsfolgen erstellt. Anders als bei Abschnitt 9.2.1 erreichten beide BSTs ihre kürzeste Laufzeit jeweils bei der gleichen Zugriffsfolge, so dass diese in einem Diagramm dargestellt werden.

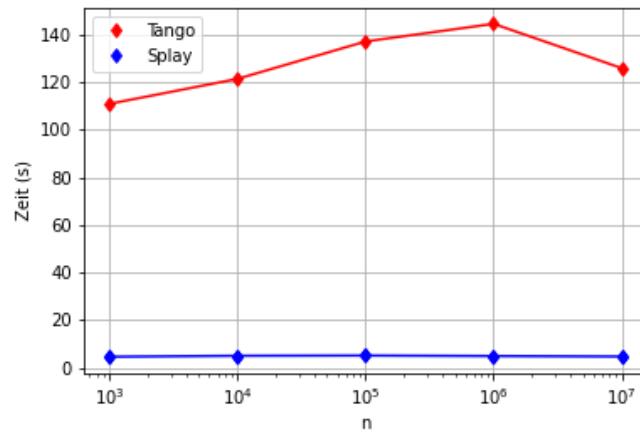


Abbildung 55: Ergebnisse zu den Zugriffsfolgen zur Static Finger Property mit den kürzesten Laufzeiten beider BSTs.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	110,8	4,6
10^4	121,3	5
10^5	137	5,1
10^6	144,4	4,9
10^7	125,8	4,7

Tabelle 7: Ergebnisse zu den Zugriffsfolgen zur Static Finger Property mit den kürzesten Laufzeiten beider BSTs.

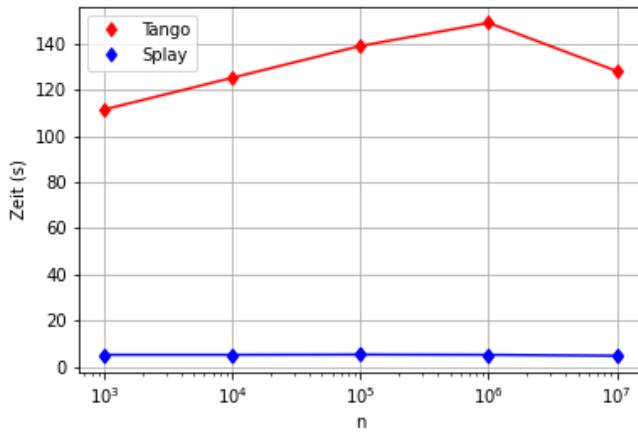


Abbildung 56: Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	111,5	5,1
10^4	125,3	5,1
10^5	139,1	5,2
10^6	149,1	5,1
10^7	128,2	4,7

Tabelle 8: Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property.

Sei X eine der verwendeten Zugriffsfolgen und L die Menge der in X enthaltenen Schlüssel. L hat eine Kardinalität von 968.

Der Splay Baum erfüllt die Static Finger Property und zusätzlich führt die eingeschränkte Kardinalität von L gegenüber K zu den niedrigen Laufzeiten. Auch der Tango Baum erreicht hier verglichen mit den weiter oben beschriebenen Laufzeittests kurze Laufzeiten. Die zu Beginn des Kapitels angeführte Vermutung, dass der Tango Baum aufgrund seiner $\log(\log(n))$ -competitiveness ebenfalls von den speziellen Eigenschaften solcher Zugriffsfolgen profitiert, bestätigt sich in diesem Fall. Auch hier kann zusätzlich die Kardinalität von L zur Erklärung verwendet werden. Bei den größeren Instanzen wird sich das preferred child bei vielen Knoten im Referenzbaum zum Tango Baum entweder gar nicht oder nur bei wenigen *access* Operationen ändern.

Nun werden Zugriffsfolgen für $n = 10.000.000$ verwendet. Es werden verschiedene Prozentsätze p zum Berechnen der jeweiligen Anzahl des Vorkommens eines Schlüssels in der Zugriffsfolge verwendet. Pro Prozentsatz werden wieder die Durchschnittswerte zu zehn Zugriffsfolgen dargestellt.

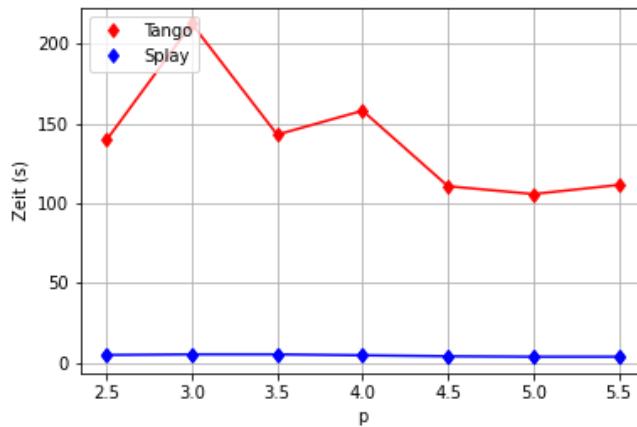


Abbildung 57: Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property mit variablem p .

p in %	Zeit Tango in (s)	Zeit Splay in (s)
2, 5	139, 6	5
3	212, 4	5, 4
3, 5	143	5, 4
4	158	4, 8
4, 5	110, 7	4, 2
5	105, 8	3, 9
5, 5	111, 5	3, 9

Tabelle 9: Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property mit variablem p .

Die Laufzeiten des Splay Baumes werden erwartungsgemäß mit steigendem p kürzer. Beim Tango Baum gibt es auch steigende Werte bei der Laufzeit, trotz steigenden Werten für p . Tendenziell werden aber auch seine Laufzeiten mit steigendem p kürzer. Hier muss auch beachtet werden, dass wieder ein Pseudozufallsgenerator verwendet wurde.

9.2.4 Dynamic Finger Property

Im ersten Teil ist die Anzahl der Knoten wieder variabel und es werden Zugriffsfolgen der Form $1, 3, 5, \dots, n-1, n-3, n-5, \dots, 1, 3, 5, \dots, n-1, \dots$ verwendet. Der Betrag a der Differenz zwischen zwei aufeinanderfolgenden Schlüsseln ist also 2.

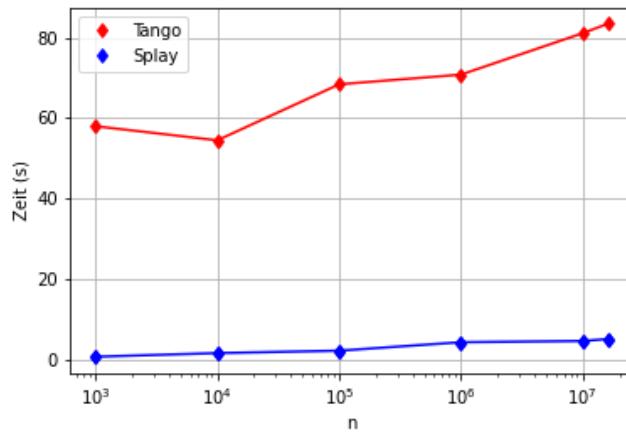


Abbildung 58: Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit konstantem a .

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	58	0,8
10^4	54,5	1,7
10^5	68,4	2,3
10^6	70,8	4,4
10^7	81,1	4,7
$1,6 * 10^7$	83,3	5,2

Tabelle 10: Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit konstantem a .

Der Splay Baum erfüllt die Dynamic Finger Property und erreicht wie zu erwarten war, sehr kurze Laufzeiten. Auch der Tango Baum profitiert wieder von dem speziellem Aufbau der Zugriffsfolgen, wie an einem Vergleich der Laufzeiten zu den zufällig erzeugten Zugriffsfolgen erkennbar ist.

Nun ist $n = 10.000.000$ fest und es werden unterschiedliche Werte für a verwendet.

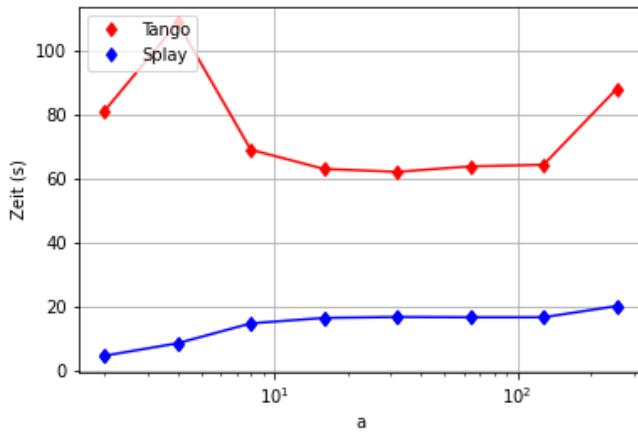


Abbildung 59: Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit variablem a .

a	Zeit Tango in (s)	Zeit Splay in (s)
2	81, 1	4, 7
4	108, 6	8, 6
8	69	14, 8
16	63	16, 5
32	62, 1	16, 8
64	63, 8	16, 7
128	64, 3	16, 7
256	88	20, 2

Tabelle 11: Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit variablem a .

Es ist zu erkennen, dass die Zeiten des Splay Baumes ansteigen, obwohl n und m konstant sind. Die Zeiten des Tango Baumes zeigen keine eindeutige Tendenz, bleiben insgesamt jedoch vergleichsweise niedrig. Die Messwerte legen die Vermutung nahe, dass der Tango Baum von der konstanten Betragsdifferenz zweier aufeinanderfolgender Schlüssel profitiert. Es kann jedoch nicht abgeleitet werden, dass ein niedrigerer Wert bei a auch zu kürzeren Laufzeiten führt.

9.2.5 Working Set Property

Die Schlüssel dieser Zugriffsfolgen wiederholen sich mit einem festem parametrierbaren Abstand a . Sei $X = x_1, x_2, \dots, x_m$ eine verwendete Zugriffsfolge und $i \in \{1, 2, \dots, m\}$. Sei k der Schlüssel mit $k = x_i$. Dann ist entweder $i - a$ der größte Index kleiner i , mit $x_{i-a} = k$ oder i ist der kleinste Index mit $x_i = k$. Es werden die Teilmengen t_0, t_1, \dots, t_b mit $b = \lfloor n/a \rfloor$ der Schlüsselmenge gebildet. Sei $l \in \{0, 1, \dots, b\}$, $c = \lfloor a/2 \rfloor$ und $d = \lfloor n/2 \rfloor$, dann gilt:

$$t_l = \{l * c + 1, l * c + 2, \dots, l * c + c\} \cup \{l * c + 1 + d, l * c + 2 + d, \dots, l * c + c + d\}.$$

Jede Teilmenge enthält also $2c$ Elemente. Sei $e_1, e_2, \dots, e_c, e_{c+1}, e_{c+2}, \dots, e_{c+c}$ die Folge aller in t_l enthaltenen Elemente, aufsteigend sortiert nach dem Betrag. Aus t_l wird eine Zugriffsfolge s_l gebildet, indem $e_1, e_{c+1}, e_2, e_{c+2}, \dots, e_c, e_{c+c}$ so oft wiederholt wird, bis s_l die Länge $\lfloor m/b \rfloor$ hat. Die Gesamtzugriffsfolge ist dann $s_0 \circ s_1 \circ \dots \circ s_l$.

Durch diesen Aufbau wird erreicht, dass zwar die Working Set Property für die Laufzeit relevant wird, die bereits betrachteten Eigenschaften jedoch nicht.

Im ersten Teil wird nun $a = 8$ gesetzt.

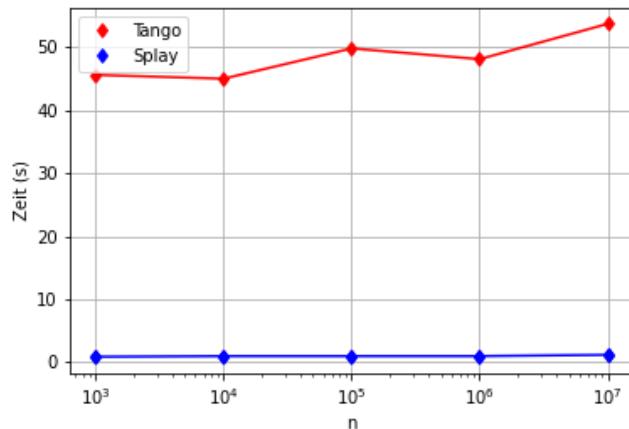


Abbildung 60: Die Ergebnisse zum Laufzeittest zur Working Set Property mit konstantem a .

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	46,6	0,9
10^4	45	1
10^5	49,8	1
10^6	48,1	1
10^7	53,7	1,2

Tabelle 12: Die Ergebnisse zum Laufzeittest zur Working Set Property mit konstantem a .

Beide BSTs erreichen hier ihre bisher kürzesten Laufzeiten. Zudem werden die Laufzeiten bei steigendem n nur etwas länger, trotz logarithmischer Skala zu n . Der Splay Baum erfüllt die Working Set Property, wodurch die kurzen Laufzeiten zu erklären sind. Nun wird eine mögliche Erklärung zum langsamsten Ansteigen der Laufzeiten des Tango Baumes angegeben:

Seien k_1, k_2 und k_3 im Tango Baum enthaltene Schlüssel, so dass $k_1 + 1 = k_3$ und $k_2 \gg k_1$ gilt. Es wird aufeinanderfolgend $access(k_1)$, $access(k_2)$ und $access(k_3)$ ausgeführt. Betrachtet wird nun der Referenzbaum P zum Tango Baum und $access(k_3)$. Sei $P_k = (p_1, p_2, \dots, p_l)$ der Pfad von der Wurzel von P zu dem Knoten v_3 mit dem Schlüssel k_3 . Sei v_v der gemeinsame Vorfahre von v und dem Knoten v_2 mit dem Schlüssel k_2 mit der größten Tiefe. Bei keinem Knoten mit einer kleineren Tiefe als der von v_v kann sich eine Veränderung beim preferred child ergeben. Sei u ein Knoten in P_k mit größerer Tiefe als v_v , sowie $key(u) \neq k_1 \wedge key(u) \neq k_2$. Es muss entweder $key(u) > k_3 > k_1$ oder $k_3 > k_1 > key(u)$ gelten und somit müssen k_1 und k_3 im selben Teilbaum von u enthalten sein. So kann erklärt werden, dass verlängerte Pfade im Referenzbaum hier nur zu wenigen zusätzlichen Veränderungen bei preferred children führen.

Im zweiten Teil wird $n = 10.000.000$ gesetzt und a ist variabel.

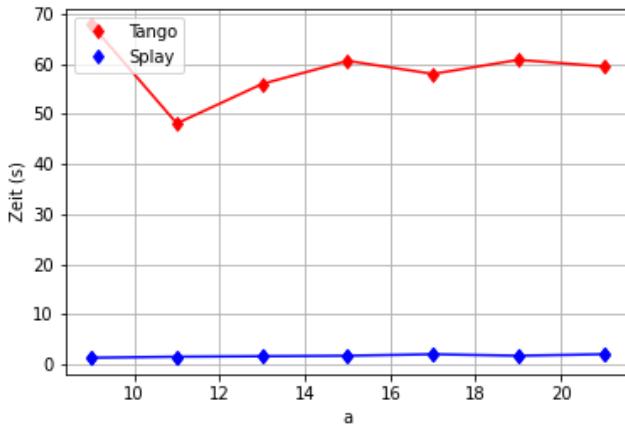


Abbildung 61: Die Ergebnisse zum Laufzeittest zur Working Set Property mit variablem a .

a	Zeit Tango in (s)	Zeit Splay in (s)
9	67,8	1,3
11	48,1	1,4
13	56	1,6
15	60,6	1,6
17	58	1,7
19	60,8	1,8
21	59,5	2

Tabelle 13: Die Ergebnisse zum Laufzeittest zur Working Set Property mit variablem a .

Die Laufzeiten des Splay Baumes nehmen mit steigendem a etwas größere Werte an, bleiben jedoch insgesamt kurz. Bei den Laufzeiten des Tango Baumes ist keine klare Tendenz auszumachen. Das bestätigt die Vermutung, dass der niedrige Betrag der Differenz der Parameter zweier *access* Operationen, zwischen deren Ausführung nur eine weitere *access* Operation liegt, wesentlich für die kurzen Laufzeiten ist.

9.2.6 Weitere Laufzeittests

Alternierend die Schlüssel 1 und n :

Umgesetzt werden diese Zugriffsfolgen durch solche zur Dynamic Finger Property, bei denen $a = n - 1$ für den Abstand a gilt. Relevant für die Laufzeit wird hier jedoch die Working Set Property.

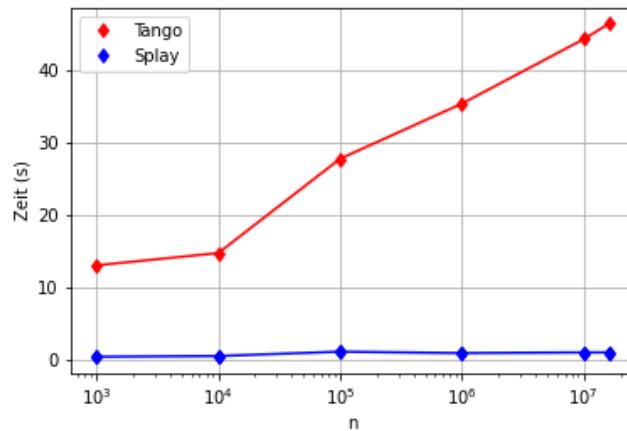


Abbildung 6.2: Laufzeittests, bei denen alternierend der kleinste und der größte Schlüssel verwendet wird.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	13	0,4
10^4	14,7	0,5
10^5	27,7	1,1
10^6	35,3	0,9
10^7	44,2	1
$1,6 * 10^7$	46,3	1

Tabelle 14: Laufzeittests, bei denen alternierend der kleinste und der größte Schlüssel verwendet wird.

Beide BSTs erreichen hier ihre bisher kürzesten Laufzeiten. Dies war auch zu erwarten. Im Referenzbaum zum Tango Baum wird es ab der dritten *access* Operation, pro Operation nur noch zu einem Wechsel des preferred child an der Wurzel kommen. Beim Splay Baum ist der Knoten mit dem

größten Schlüssel nach der zweiten Operation an der Wurzel. Der Knoten mit dem kleinsten Schlüssel war nach der ersten Operation an der Wurzel. Somit sind ab der dritten Operation jeweils nur noch maximal zwei Rotationen notwendig. Zwei Rotationen sind der Fall, wenn bei der vorherigen Operation zum Abschluss zig-zig oder zag-zag angewendet wurde, siehe Abbildung 37.

Aufsteigend sortiert mit insgesamt n Zugriffen:

Zu beachten ist hier, dass die Dynamic Finger Property relevant wird.

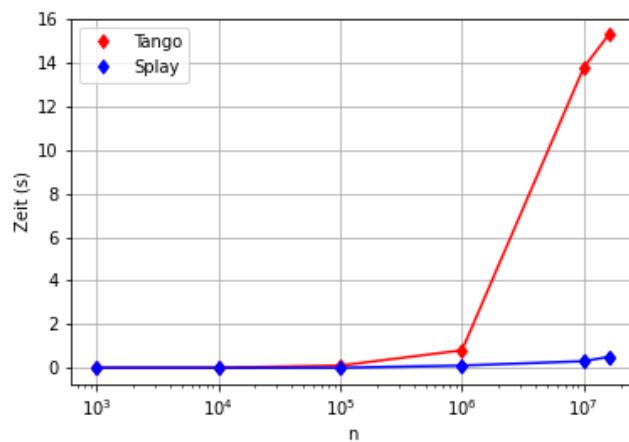


Abbildung 63: Laufzeittests, bei denen aufsteigend sortierte Zugriffsfolgen verwendet werden.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	≈ 0	≈ 0
10^4	0, 1	≈ 0
10^5	0, 2	≈ 0
10^6	1	0, 1
10^7	12.3	0, 3
$1,6 * 10^7$	16.8	0, 4

Tabelle 15: Laufzeittests, bei denen aufsteigend sortierte Zugriffsfolgen verwendet werden.

Beide BSTs erreichten bei den Zugriffsfolgen zur Dynamic Finger Property vergleichsweise kurze Laufzeiten. Hier kommt hinzu, dass die Länge der Zugriffsfolgen abhängig von der Anzahl der Knoten kürzer ist. Das Ergebnis sind erwartbar kurze Laufzeiten.

Absteigend sortiert mit insgesamt n Zugriffen:

Auch hier wird die Dynamic Finger Property relevant.

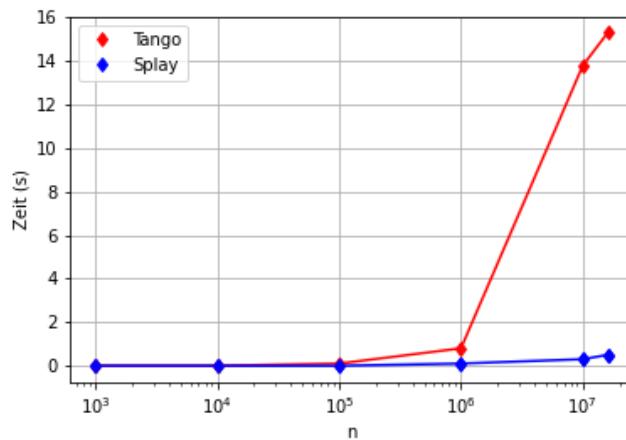


Abbildung 64: Laufzeittests, bei denen absteigend sortierte Zugriffsfolgen verwendet werden.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	≈ 0	≈ 0
10^4	≈ 0	≈ 0
10^5	0, 1	≈ 0
10^6	0, 8	0, 1
10^7	13, 8	0, 3
$1,6 \cdot 10^7$	15, 3	0, 5

Tabelle 16: Laufzeittests, bei denen absteigend sortierte Zugriffsfolgen verwendet werden.

Die Laufzeiten sind ähnlich zu denen der aufsteigend sortierten Zugriffsfolgen und diese können auf die gleiche Art und Weise begründet werden.

Alternierend aufsteigend und absteigend sortiert:

Hier werden Zugriffsfolgen mit dem Aufbau $1, n, 2, n - 1, \dots, n - 1, 2, n, 1$ verwendet. Die Länge der Zugriffsfolgen ist also $2n$.

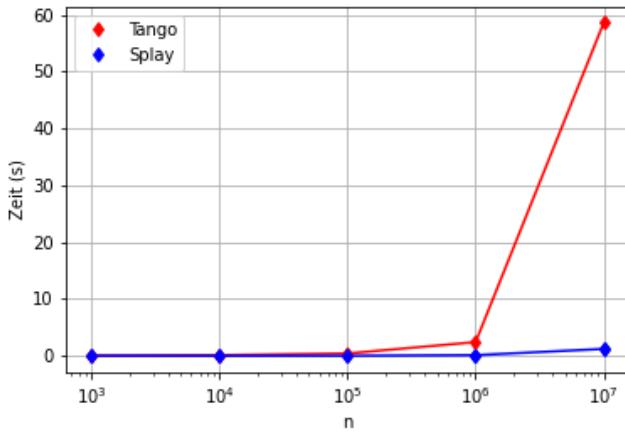


Abbildung 65: Laufzeittests, bei denen alternierend aufsteigend und absteigend sortierte Zugriffsfolgen verwendet werden.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	≈ 0	≈ 0
10^4	0,1	≈ 0
10^5	0,6	≈ 0
10^6	9,2	0,1
10^7	58,6	1,2

Tabelle 17: Laufzeittests, bei denen alternierend aufsteigend und absteigend sortierte Zugriffsfolgen verwendet werden.

Die Laufzeiten steigen gegenüber den aufsteigend bzw. absteigend sortierten Zugriffsfolgen um mehr als das Doppelte an, vor allem bei den großen Instanzen. Eine Erklärung dafür ist, dass hier die Dynamic Finger Property lediglich beim mittleren Teil der Zugriffsfolge relevant wird. Und auch beim mittleren Teil bleibt die Betragsdifferenz zweier aufeinanderfolgender Schlüssel nicht konstant. Bei den Tests zur Dynamic Finger Property, profitierte der Tango Baum eher von einer solchen konstanten Betragsdifferenz, als von niedrigen Betragsdifferenzen.

9.3 Fazit zu den Laufzeittests

Hier werden die Ergebnisse der Laufzeittests eingeordnet und mit den bekannten theoretischen oberen Laufzeitschranken verglichen.

Zum einen wurden hier Laufzeiten mit einem realem System ermittelt. Deshalb gibt es Einflüsse auf die Laufzeiten die unabhängig von den BSTs sind, wie z.B. die Speicherverwaltung. Auffällig ist z. B., dass die Laufzeit des Tango Baumes bei den Tests mit zufällig erzeugten Zugriffsfolgen, bei der größten Knotenanzahl unverhältnismäßig stark ansteigt. Diese Konstellation ist auch eine mit der höchsten Speicherauslastung, da der Tango Baum einen höheren Speicherbedarf als der Splay Baum hat und hier die gesamte Zugriffsfolge im Speicher gehalten wird. Zusätzlich verstärkt die logarithmische x-Skala diesen Eindruck. Denn auch bei diesen Konstellationen führte eine Verzehnfachung der Knotenmenge lediglich zu einer in etwa 3,5-fachen Laufzeit. Ein Merkmal der Laufzeittests war auch, dass die Laufzeiten auch bei identischen Konstellationen über mehrere Versuche etwas schwankten. Dies macht zum Beispiel das exakte Einordnen der zum Teil sehr kurzen Laufzeiten des Splay Baumes schwierig.

Die theoretischen Laufzeitschranken sind mit der O-Notation angegeben. Konkrete Kennlinien dazu könnten mit konstanten Faktoren in der Form angepasst werden und mit konstanten Summanden auf der y-Skala verschoben werden.

Bei den zufällig erzeugten Zugriffsfolgen und den Bit Reversal Permutations verhalten sich die jeweils niedrigsten oberen Schranken logarithmisch. Für die Bit Reversal Permutations wurde dies in Abschnitt 4.3 gezeigt und bei einer Zugriffsfolge die zufällig erzeugt wurde, kann auch eine solche erzeugt worden sein, bei der jeweils auf einen in der untersten Baumebene enthaltenen Schlüssel zugegriffen wird. Beim Splay Baum ist aufgrund der logarithmischen x-Skala und dem Balanced Property in etwa eine Gerade als Kennlinie zu erwarten. Beim Tango Baum ist von einer Kennlinie auszugehen die $\log(n) \log(\log(n))$ abbildet. Die ermittelten Daten spiegeln dies mit Ausnahme der Kennlinie des Tango Baumes zu den durchschnittlichen Werten, bei den zufällig erzeugten Zugriffsfolgen, auch in etwa wider. Diese andere Kennlinie hat zwei deutliche Anstiege und dazwischen schwankende Werte. Seine beiden Kennlinien zu den jeweils kürzesten Laufzeiten kommen dem erwarteten Verhalten deutlich näher. Es müsste hier mit größeren Anzahlen von Knoten weiter getestet werden, um erkennen zu können ob sich die Kennlinie auch bei den durchschnittlichen Werten noch anpasst.

Bei den Tests mit variablem n zu den drei speziellen Eigenschaften des Splay Baumes wurden jeweils Zugriffsfolgen gewählt, bei denen der wesentliche Summand zum Bilden der jeweiligen theoretischen Schranke konstant oder annähernd konstant bei einem niedrigen Wert bleibt. Hier ist jedoch zu beachten, dass jeweils auch entweder n oder $n \log(n)$ als Summand innerhalb der O-Notation enthalten ist. Die gemessenen Laufzeiten zu diesen Zugriffsfolgen sind insgesamt sehr niedrig und der Bereich der eingesetzten

Werte für n ist groß. Deshalb ist auch beim Splay Baum nicht unbedingt von konstanten Laufzeiten auszugehen. Beim Tango Baum ist aufgrund der $\log(\log(n))$ -competitiveness von nur etwas stärkeren Reaktionen der Laufzeit auszugehen.

Die ermittelten Werte entsprechen bei der Working Set Property und der Static Finger Property den Erwartungen. Bei der Dynamic Finger Property steigen die ermittelten Werte des Splay Baumes an, was in diesem speziellen Fall, wie bereits angemerkt, kein Widerspruch sein muss. Passend dazu ist, dass auch die Laufzeiten des Tango Baumes merklich länger werden.

Bei den Laufzeittests zu den speziellen Eigenschaften des Splay Baumes mit konstantem n und variablem p , bzw. a , gibt es bei den oberen Schranken innerhalb der O-Notation jeweils nur einen Summanden der nicht konstant bleibt. Bei der Working Set Property ist dies $O(\sum_{i=1}^m \log(a_i))$. Dieser Ausdruck kann bei unserem speziellen Aufbau der Zugriffsfolgen zu

$O(n \log(1) + (m-n) \log(a))$ umgeformt werden, wobei die Bezeichner aus dem jeweiligen Abschnitt zum Laufzeittest verwendet werden. Somit ist von logarithmisch steigenden Zeiten beim Splay Baum auszugehen. Bei der Dynamic Finger Property ist $O(\sum_{i=2}^m \log(|x_{i-1} - x_i| + 1))$ der variable Summand. Dieser kann bei unserem Aufbau der Zugriffsfolgen zu $O((m-1) \log(a+1))$ umgeformt werden. Somit ist von logarithmisch steigenden Zeiten beim Splay Baum auszugehen. Beim Tango Baum ist ein ähnliches Verhalten etwas verstärkt zu erwarten. Zu beiden Propertys zeigen die BSTs in etwa das erwartete Verhalten. Bei der Static Finger Property ist $s = \sum_{i=1}^m \log(k_f - x_i + 1)$ der variable Summand innerhalb der O-Notation. Hier ist es schwieriger das Verhalten abzuleiten, deshalb wird der Summand in Abbildung 66 graphisch dargestellt.

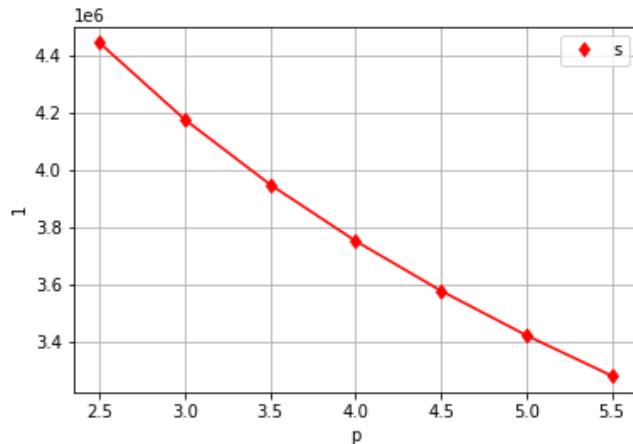


Abbildung 66: Werte des variablen Summanden innerhalb der O-Notation zur Static Finger Property, zu den Laufzeittests mit variablem p . Es wurde der Logarithmus zur Basis 2 verwendet. Es wurde eine Anzahl der Knoten von 10 Millionen verwendet und Zugriffsfolgen mit der Länge 40 Millionen.

Das Abfallen der Kennlinie aus Abbildung 66 flacht langsam ab. Die Laufzeiten des Splay Baumes spiegeln dieses Verhalten in etwa wider, obwohl die Laufzeit von $p = 2,5\%$ auf $p = 3,5\%$ sogar leicht ansteigt. Die Laufzeiten des Splay Baumes sind bei diesen Konstellationen jedoch sehr niedrig, so dass externe Einflüsse hier relativ groß werden könnten. Die Laufzeiten des Tango Baumes werden tendenziell kürzer, steigen bei drei Konstellationen jedoch sogar an. Hier werden jedoch Schlüssel aus einer Teilmenge der eigentlichen Schlüsselmenge verwendet. Die Elemente dieser Teilmenge bilden auch noch ein Intervall bezogen auf den Betrag der Schlüssel. Da auch noch ein Pseudozufallsgenerator verwendet wird, könnte es sein, dass sich die anderen beiden Propertys ebenfalls auswirken und dass bei verschiedenen Folgen in unterschiedlichem Ausmaß.

Anhang: Daten des Systems zu den Laufzeittests

Bildschirm	
Bildschirmdiagonale in Zentimeter	39,6 cm
Bildschirmdiagonale in Zoll	15,6 "
Bildschirmauflösung in Pixel	1920 x 1080
Auflösungsstandard	Full HD
Bildschirmtechnologie	WLED-Hintergrundbeleuchtung
Verstellbarkeit Bildschirm	drehbar
Betrachtungswinkel	360 °

Speicher	
Typ Festplatte	SSD HDD
Anzahl installierter Festplatten	2
Speicherkapazität Festplatte gesamt	1256 GB
Speicherkapazität Festplatte HDD	1000 GB
Speicherkapazität Festplatte SSD	256 GB
Größe Arbeitsspeicher (RAM)	8 GB
Typ Arbeitsspeicher	DDR4-SRAM
Taktung Arbeitsspeicher	2400 MHz

Prozessor	
Prozessorhersteller	Intel
Prozessorserie	Core i5
Prozessornummer	8265U
Prozessorbauart	Quad-Core
Anzahl Prozessorkerne	4
Taktfrequenz Prozessor	1,6 GHz
Turbo-Taktfrequenz Prozessor maximal	3,9 GHz

Betriebssystem / Software	
Betriebssystem	Microsoft Windows 10 Home (64Bit)
Installationsart Betriebssystem	vorinstalliert
Software (vorinstalliert)	HP apps: HP 3D DriveGuard; HP CoolSense; HP JumpStart; HP Support Assistant; Mitgelieferte Software: McAfee LiveSafe™; Testversion für Kunden des neuen Microsoft Office 365 für 1 Monat; Vorinstallierte Software: Netflix

10 Allgemeines Fazit und Ausblick

Der Tango Baum hat in den Praxistests im Vergleich zum Splay Baum wenig überzeugend abgeschnitten. Auch der Aufwand zur Implementierung ist für einen BST recht hoch. Mit ihm wurde die Idee, auf Grundlage der Interleave Lower Bound einen $\log(\log(n))$ -competitive BST zu konstruieren jedoch als Erstes umgesetzt und somit ein wichtiger Beitrag zur Forschung zur dynamischen Optimalität geleistet. Außerdem könnten hier noch weitere interessante Variationen, mit zusätzlichen guten Laufzeiteigenschaften folgen.

Mittlerweile hat sich zum Thema „dynamische Optimalität“ viel Literatur angesammelt. Hier wurde nur auf einen sehr kleinen Ausschnitt eingegangen. Zum Beispiel gibt es weitere untere Schranken für die Ausführungszeit von BSTs. Außerdem gibt es eine interessante geometrische Sicht auf BSTs. Von dieser konnten wichtige Aussagen über BSTs und andere Verfahren abgeleitet werden. Es gibt auch deutlich mehr obere Laufzeitschranken für Zugriffssfolgen mit speziellen Eigenschaften, als hier vorgestellt wurden.

Ob es irgendwann gelingen wird die dynamische Optimalität des Splay Baumes oder irgendeiner anderen Variante eines BSTs zu beweisen, ist offen. Für den Einsatz in der Praxis, müssen dann aber auch die Summanden und Faktoren berücksichtigt werden, die innerhalb der O -Notation vernachlässigt werden.

Abbildungsverzeichnis

1	Ein binärer Suchbaum.	7
2	Kein binärer Suchbaum.	7
3	Ein weiterer binärer Suchbaum	9
4	Die Schlüssel sind aufsteigend sortiert ablesbar.	9
5	Darstellung von Vorgänger und Nachfolger.	11
6	Linksrotation auf Knoten x.	12
7	Rechtsrotation auf Knoten x.	13
8	Gegenseitiges Aufheben von Rotationen.	13
9	Links zeigt eine Suche nach dem Schlüssel 15. Rechts das Einfügen des Schlüssels 13.	14
10	Löschen des Schlüssels 10	15
11	Kompletter BST mit 12 Knoten	16
12	RBT ohne Verletzung von Eigenschaften.	18
13	Kein RBT, da die Eigenschaften vier und fünf verletzt sind.	19
14	Kein RBT, da die Eigenschaft fünf verletzt ist.	19
15	<i>insertFixup</i> . Dargestellt ist der Fall 1.	21
16	<i>insertFixup</i> . Dargestellt ist der Fall 2.	23
17	<i>insertFixup</i> . Dargestellt ist der Fall 3.	24
18	<i>insertFixup</i> . Dargestellt ist der Fall 4.	26
19	Rechts ist ein möglicher Lower Bound Tree zum linken BST dargestellt.	31
20	Links ein BST T , rechts ein davon abgeleiteter BST T_4^8	33
21	Implikationen zwischen den Eigenschaften, abgeleitet aus einer Abbildung aus [7].	43
22	Der Lower Bound Tree zur Zugriffsfolge 1, 2, .., 15.	45
23	Transition point Zuordnung. Links ein Lower Bound Tree, rechts ein möglicher T_j .	47
24	Die preferred children im Referenzbaum werden durch die grünen Pfeile markiert.	49
25	Die Hilfsbäume zu den preferred paths aus dem Beispiel.	50
26	Der Tango Baum zu dem Beispiel.	51
27	Die preferred children nachdem <i>access(9)</i> ausgeführt wurde.	52
28	Der Tango Baum nachdem <i>access(9)</i> ausgeführt wurde.	52
29	Ablauf von <i>cut(d)</i> . Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.	56
30	Ablauf von <i>cut(d)</i> bei fehlenden r' . Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.	57

31	Ablauf von $join(H_1, H_2)$. Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert	59
32	Ablauf von $join(H_1, H_2)$ bei fehlendem r' . Die Abbildung basiert auf einer aus [2]. Die Wurzeln von Hilfsbäumen sind mit einem Kreis markiert.	59
33	Darstellung von Schritt 3 und 4 der <i>access</i> Operation.	61
34	Beispielhaftes <i>concatenate</i> zweier RBTs unterschiedlicher Schwarz-Höhe, nach Schritt 1.	63
35	Beispielhaftes <i>concatenate</i> zweier RBTs gleicher Schwarz-Höhe, nach Schritt 1.	64
36	Beispielhaftes <i>split</i> eines RBTs mit Parameter v . Die Symbole in den Knoten beziehen sich auf die Schlüssel der Knoten.	70
37	Darstellung von zig, zig-zig und zig-zag.	72
38	Eine einzige <i>splay</i> Operation. [1]	72
39	Pfadrepräsentation beim Zipper Baum, basiert auf einer Abbildung aus [10].	77
40	zig Segmente sind grün dargestellt. zag Segmente blau.	78
41	Extraktionsprozess beim Zipper Baum.	79
42	Referenzbaum mit grün gezeichneten preferred paths.	80
43	Beispielhafter Multisplay Baum zum Referenzbaum aus Abbildung 42.	80
44	Screenshot des Hauptfensters der GUI. Im oberen Teil befindet sich der Referenzbaum. Im unteren Teil der dazugehörige Tango Baum.	81
45	Mit diesem Fenster können <i>access</i> Operationen auf dem Tango Baum des Hauptfensters angestoßen werden.	82
46	Mit diesem Fenster kann ein Laufzeittest ausgewählt, parametriert und angestoßen werden.	82
47	Mit diesem Fenster wird das Resultat eines Laufzeittests dargestellt.	82
48	Wesentliche Klassen der Implementierung. Methoden zum direkten Lesen, bzw. Schreiben von Attributen sind nicht dargestellt.	83
49	Den zur Ausführung verwendbaren Speicher auf 4096MB erweitern.	85
50	Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Tango Baumes.	87
51	Ergebnisse zu den zufällig erzeugten Zugriffsfolgen, mit den kürzesten Laufzeiten des Splay Baumes.	88

52	Durchschnittliche Laufzeiten zu jeweils zehn zufällig erzeugten Zugriffsfolgen.	89
53	Hier sind n_T und n_S graphisch dargestellt.	91
54	Ergebnisse zu den Laufzeittests zur Bit Reversal Permutation.	92
55	Ergebnisse zu den Zugriffsfolgen zur Static Finger Property mit den kürzesten Laufzeiten beider BSTs.	93
56	Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property.	94
57	Durchschnittliche Laufzeiten über jeweils zehn Zugriffsfolgen zur Static Finger Property mit variablem p	95
58	Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit konstantem a	96
59	Die Ergebnisse zum Laufzeittest zur Dynamic Finger Property mit variablem a	97
60	Die Ergebnisse zum Laufzeittest zur Working Set Property mit konstantem a	98
61	Die Ergebnisse zum Laufzeittest zur Working Set Property mit variablem a	100
62	Laufzeittests, bei denen alternierend der kleinste und der größte Schlüssel verwendet wird.	101
63	Laufzeittests, bei denen aufsteigend sortierte Zugriffsfolgen verwendet werden.	102
64	Laufzeittests, bei denen absteigend sortierte Zugriffsfolgen verwendet werden.	103
65	Laufzeittests, bei denen alternierend aufsteigend und absteigend sortierte Zugriffsfolgen verwendet werden.	104
66	Werte des variablen Summanden innerhalb der O-Notation zur Static Finger Property, zu den Laufzeittests mit variablem p . Es wurde der Logarithmus zur Basis 2 verwendet. Es wurde eine Anzahl der Knoten von 10 Millionen verwendet und Zugriffsfolgen mit der Länge 40 Millionen.	107

Selbstständigkeitserklärung

Name : Andreas Windorfer
Matrikel-Nummer: q8633657
Fach: Informatik
Modul: Bachelorarbeit
Thema: Tango Bäume

Ich erkläre, dass ich die Abschlussarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Datum, Ort

Unterschrift

Literatur

- [1] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, July 1985.
- [2] Erik D. Demaine, Dion. Harmon, John. Iacono, and Mihai. Patrascu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [4] Karel Culik and Derick Wood. A note on some tree similarity measures. *Information Processing Letters*, 15(1):39 – 42, 1982.
- [5] Robert Wilber. Lower bounds for accessing binary search trees with rotations. *SIAM Journal on Computing*, 18(1):56–67, 1989.
- [6] Norman Abramson. *Information theory and coding*. McGraw-Hill electronic sciences series. McGraw-Hill, New York, NY, 1963.
- [7] Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn, and Thatchaphol Saranurak. The landscape of bounds for binary search trees. *CoRR*, abs/1603.04892, 2016.
- [8] Robert Endre Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, USA, 1983.
- [9] Richard Cole, Bud Mishra, Jeanette Schmidt, and Alan Siegel. On the dynamic finger conjecture for splay trees. part i: Splay sorting log n-block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.
- [10] Prosenjit Bose, Karim Douïeb, Vida Dujmović, and Rolf Fagerberg. An $\tilde{o}(\log \log n)$ -competitive binary search tree with optimal worst-case access times. *Algorithm Theory - SWAT 2010*, page 38–49, 2010.
- [11] Daniel Dominic Sleator and Chengwen Chris Wang. Dynamic optimality and multi-splay trees. Technical report, 2004.
- [12] Jonathan Derryberry, Daniel Sleator, and Chengwen Chris Wang. Properties of multi-splay trees, 2009.