FernUniversität in Hagen
Fakultät für Mathematik und Informatik

Bachelorarbeit zum Thema
Tango Baum

Andreas Windorfer
q8633657

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

Zusammenfassung

Bei binären Suchbäumen wird sich besonders für die Ausführungszeit seiner Operationen interessiert. Hier wird es speziell um die Ausführungszeit von Folgen von *access* Operationen gehen, 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 schnellte binäre Suchbaum. Um dies zu beweisen wurde die „Interleave Bound“, eine untere Schranke für die Ausführungszeit solcher Folgen verwendet. Bis dahin war für keinen binären Suchbaum ein Faktor kleiner als $\log_2(n)$ bewiesen und dieser wird von den balancierten binären Suchbäumen trivial erreicht. Der Tango Baum wird im Detail vorgestellt und zusätzlich noch einige Variationen zu ihm. Abschließend werden noch Laufzeittests zwischen dem Tango Baum und dem Splay Baum durchgeführt.

Inhaltsverzeichnis

1 Einleitung

Binäre Suchbäume werden vielfältig eingesetzt. Unter anderem zur Lösung des Wörterbuchproblems. 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. Vor allem diese Zugriffe müssen deshalb effizient sein. Das binäre Suchen bietet sich hierzu als Möglichkeit an. Die Schlüssel müssen dazu auf irgendeine Art sortiert vorliegen.
Beim Telefonbuch wäre das die alphabetische Sortierung. Somit kann man mit der Suche eines Namens in etwa der Mitte des Telefonbuches starten und nach einem Vergleich zumindest die Schlüssel einer Hälfte des Buches ausschließen. 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 Suchbaum ist es, dass weiter oben liegende Elemente schneller erreicht werden können als andere. Ist bekannt, wie häufig auf welchen Schlüssel zugegriffen wird, kann diese Eigenschaft ausgenutzt werden. Ein Element wird dann umso weiter oben platziert, je häufiger auf dieses zugegriffen wird.
Ein weiterer Versuch gute Laufzeiten 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 Zugriffshäufigkeiten. 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 Allgemeinem beschrieben. Außerdem werden Begriffe definiert, die in den nachfolgenden Kapiteln verwendet werden.

2.1 Definition binärer Suchbaum

Ein **Baum** T ist ein minimal zusammenhängender gerichteter Graph. Ein Baum ohne Knoten ist ein **leerer Baum**. In einem nicht leerem Baum gibt es genau einen Knoten ohne eingehende Kante, diesen bezeichnet man als **Wurzel**. Alle anderen Knoten haben genau eine eingehende Kante. 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 genau einen. Ein **Pfad** P ist eine Folge von Knoten $(v_0,v_1,...,v_n)$ mit, $\forall i \in \{1,2,...,n\}$: v_{i-1} *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 Wurzel eines **Teilbaumes** $T(v)$. Dieser entsteht in dem 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, alle anderen Knoten werden als **innere Knoten** bezeichnet.
Bei einem **binärem 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 bezeichnet man den Teilbaum mit Wurzel w als **linken Teilbaum** bzw. **rechten Teilbaum** von v.

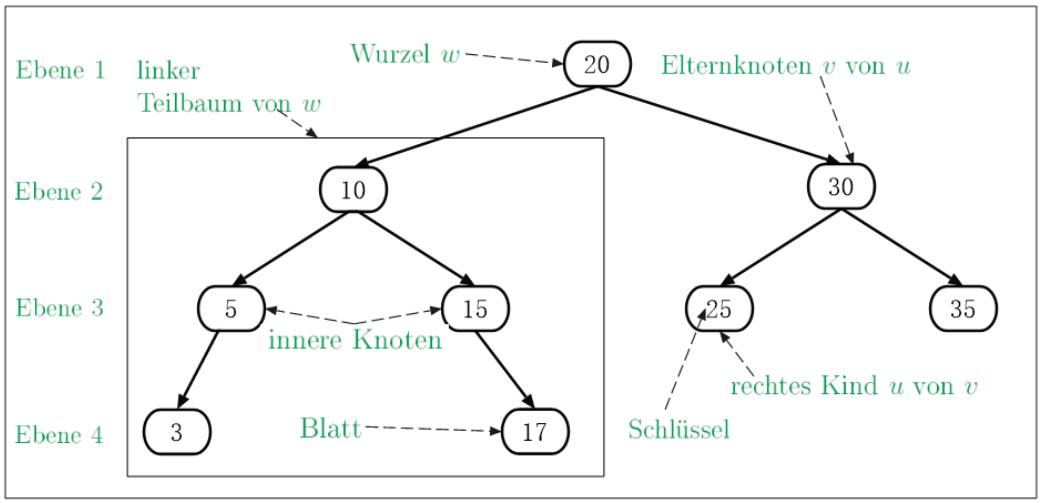


Abbildung 1: Ein binärer Suchbaum

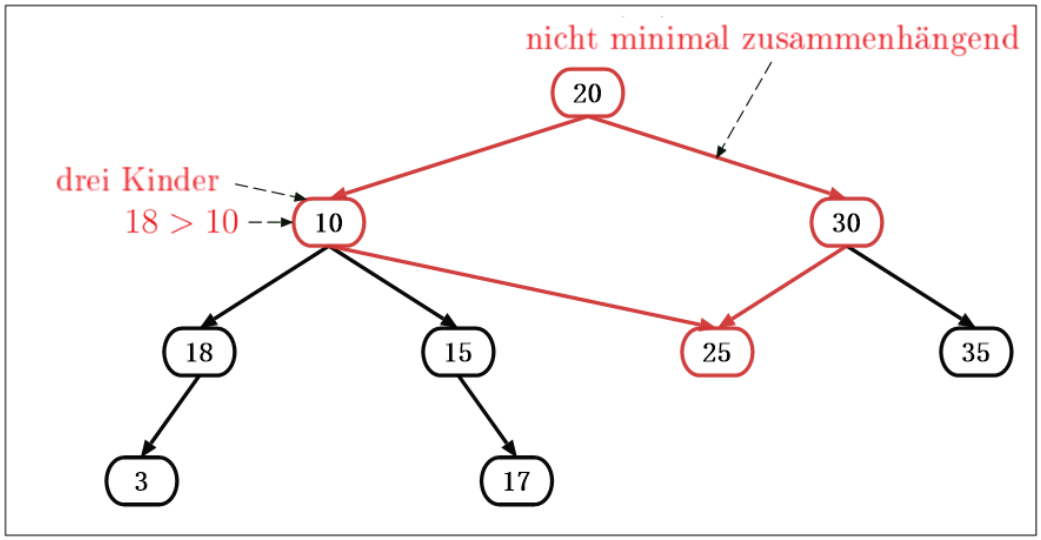


Abbildung 2: Kein binärer Suchbaum

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 binärem Suchbaum enthaltenen Schlüssel bezeichnen wir als seine **Schlüsselmenge**. Damit aus dem binären Baum ein binärer Suchbaum wird, benötigt man noch folgende Eigenschaft:

Für jeden Knoten im binären Suchbaum gilt, dass alle in seinem linken Teilbaum enthaltenen Schlüssel kleiner sind als der eigene Schlüssel. Alle im rechten Teilbaum enthaltenen Schlüssel sind größer als der eigene Schlüssel.

Anstatt binärer Suchbaum schreibt man häufig **BST** für Binary Search Tree. 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 nennt man **Geschwister**. Den Knoten in einem BST wird auch eine **Tiefe** und eine **Höhe** zugeteilt. Für einen Knoten v gilt, dass die Länge des Pfades von der Wurzel zu ihm seiner Tiefe entspricht. Sei l die maximale Länge eines von v aus startenden Pfades. Die Höhe $h(v)$ von v ist dann $l + 1$. Die Höhe der Wurzel entspricht der **Höhe des Baumes** $h(T)$, wobei ein leerer Baum Höhe 0 hat. Ein BST T mit Höhe h_T wird von oben nach unten in die **Ebenen** $1, 2..., h_T$ 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**.

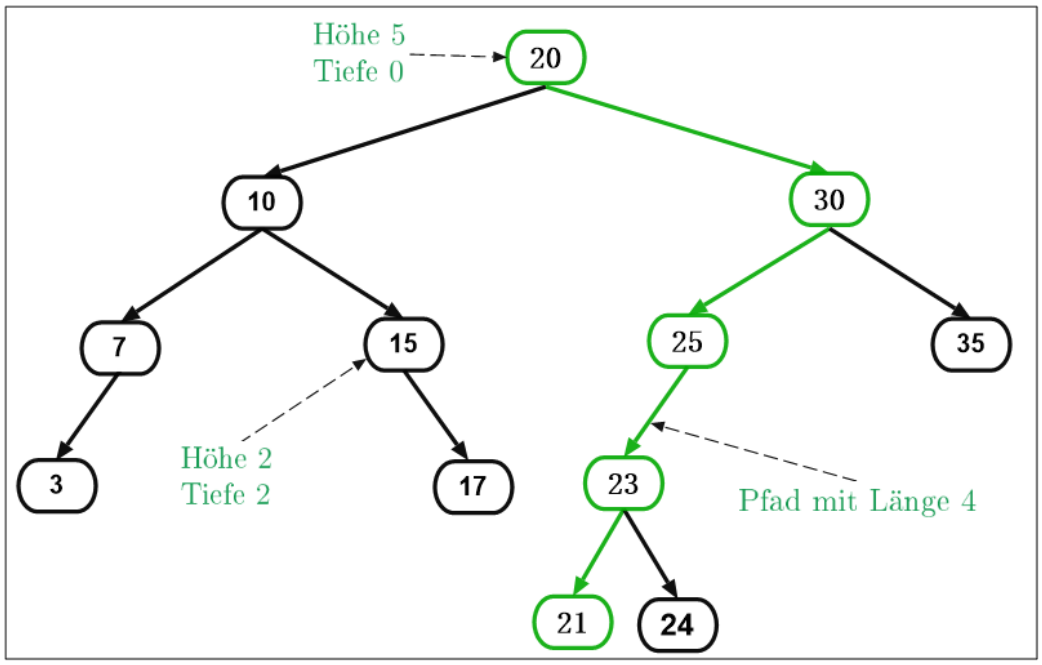


Abbildung 3: Ein weiterer binärer Suchbaum

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

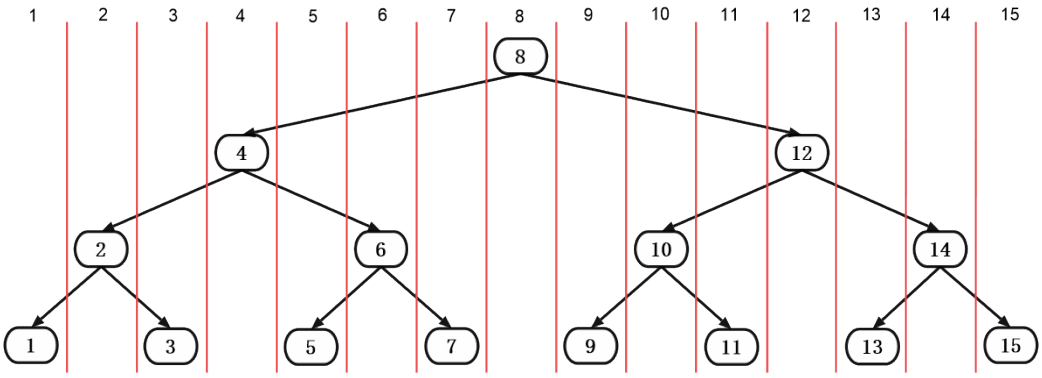


Abbildung 4: Schlüssel sind aufsteigend sortiert ablesbar.

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

Algorithmus *inorder (Node v)*

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

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 Knotenanzahl $n+1$ und Wurzel w. Sowohl für den linken, als auch für den rechten Teilbaum von w gilt, dass die Anzahl enthaltener Knoten $\leq n$ ist. Als erstes wird der linke Teilbaum von w korrekt ausgegeben, dann der Schlüssel von w selbst und zuletzt der rechte Teilbaum von w. Damit wurde auch für den Gesamtbaum die richtige Ausgabe erzeugt.
Als **Vorgänger** eines Knoten v, mit Schlüssel k_v wird der Knoten mit dem größten im BST enthaltenem Schlüssel k für den gilt $k < k_v$ 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 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 enthaltenem Schlüssel k für den gilt $k > k_v$ 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 linkem Teilbaum liegt. Abbildung 5 zeigt Vorgänger und Nachfolger eines Knotens.
Als **Vorfahre** eines Knotens v werden alle Knoten auf dem Pfad von der Wurzel zu v, inklusive v selbst, bezeichnet.

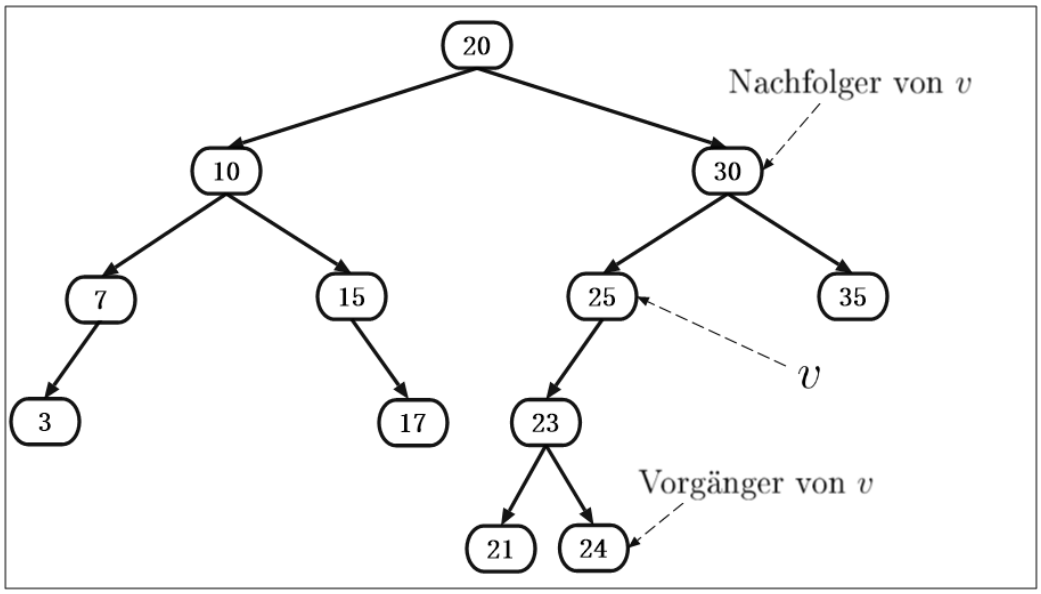


Abbildung 5: Darstellung von Vorgänger und Nachfolger.

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

Rotationen können verwendet werden, um lokale Änderungen an der Struktur eines BST durchzuführen, ohne eine der geforderten Eigenschaften zu verletzen. Es wird zwischen der Linksrotation und der Rechtsrotation unterschieden. Hier wird zunächst auf die in Abbildung 6 dargestellte Linksrotation eingegangen. Sei x der Knoten auf dem eine Linksrotation durchgeführt wird. Sei z der Elternknoten von x. z muss existieren, ansonsten darf auf x keine Rotation durchgeführt werden. Sei B der linke Teilbaum von x. Nach der Rotation ist x' linkes bzw. rechtes Kind von dem Knoten, an dem z linkes bzw. rechtes Kind war. z' ist linkes Kind von x'. Die Wurzel von B' ist rechtes 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, drei Zeiger umzusetzen. Zu beachten ist, dass die Höhen von x' und der Knoten in dessen, ansonsten unverändertem, rechtem Teilbaum jeweils um eins größer sind, als die von x und den Knoten in dessen rechtem Teilbaum. Die Höhe der Knoten im Teilbaum mit Wurzel z' sind jeweils um eins kleiner als vor der Rotation. Abbildung 7 zeigt die symmetrische Rechtsrotation. 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.

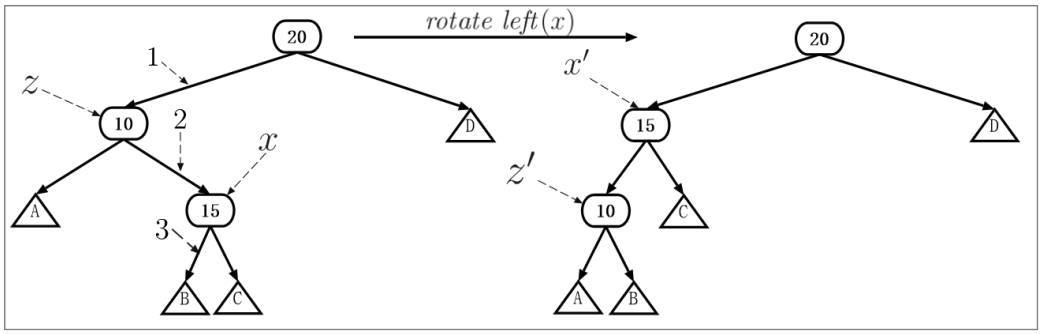


Abbildung 6: Linksrotation auf Knoten x.

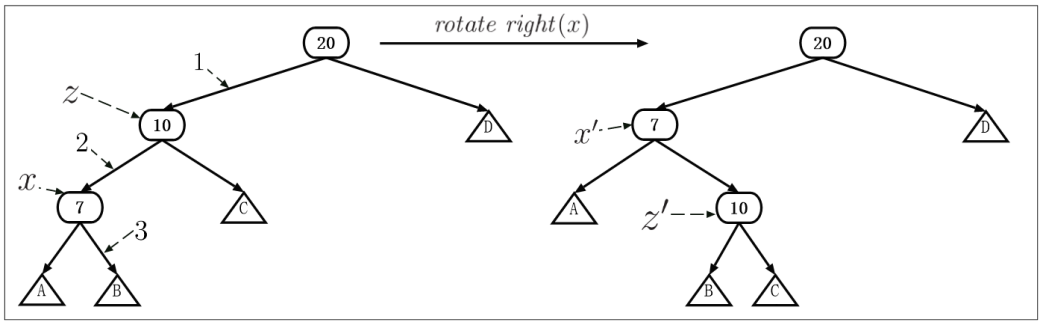


Abbildung 7: Rechtsrotation auf Knoten x.

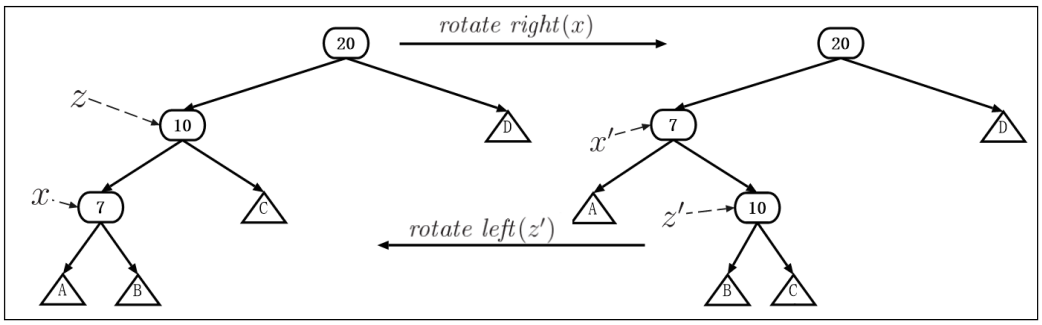


Abbildung 8: Gegenseitiges aufheben von Rotationen

Grundoperationen *search*, *insert* und *delete* Hier geht es nur um die Standardvarianten eines BST. Später werden Varianten gezeigt die von diesem Verhalten zum Teil deutlich abweichen. Innerhalb Operationen wird häufig von einem Knoten aus direkt auf dessen Elternknoten zugegriffen, so dass sich im Baum auch nach oben hin bewegt werden 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 *search(key k)* gibt eine Referenz auf den Knoten im BST zurück, dessen Schlüssel mit k übereinstimmt. Die Operation startet an der Wurzel und vergleicht den darin enthaltenen Schlüssel mit dem Gesuchten. Ist der gesuchte Schlüssel kleiner, muss er sich im linken Teilbaum des betrachtetem Knoten befinden und die Suche wird bei dessen Wurzel fortgesetzt. Ist der Schlüssel größer, muss er sich im rechten Teilbaum befinden und die Suche wird bei dessen Wurzel fortgesetzt. Dieses Verhalten iteriert solange bis der gesuchte Schlüssel gefunden ist, oder der Teilbaum bei dem die Suche fortgesetzt werden müsste, leer ist. Ist das Letztere der Fall, ist der gesuchte Schlüssel im Baum nicht vorhanden und es wird eine leere Referenz zurückgegeben. In keinem Fall kommt es zu einer Veränderung des BST.
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 Schlüssel k an der Position von T_2 eingefügt. Durch den neuen Knoten wird keine BST Eigenschaft verletzt. Durch Ersetzen eines leeren Teilbaumes, durch einen Knoten bleibt es bei einem binären Baum. Das Verhalten von *insert* stellt sicher, dass k nur in linken Teilbäumen von Knoten mit Schlüssel $> k$ bzw. in rechten Teilbäumen von Knoten mit Schlüssel $< k$ enthalten ist.

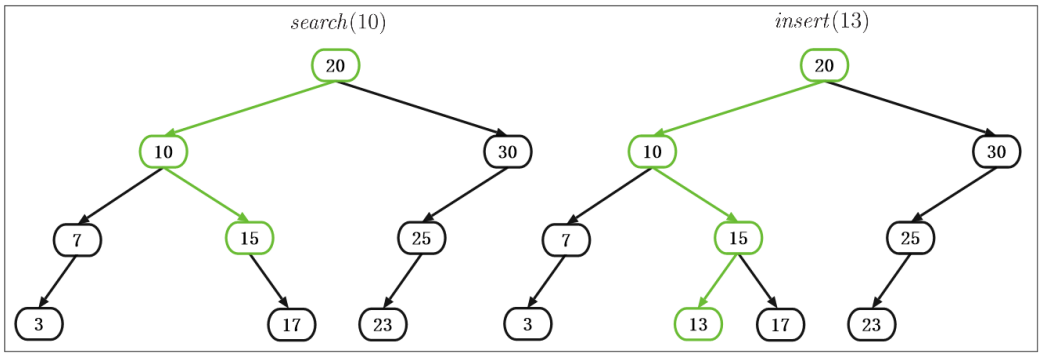


Abbildung 9: Links zeigt eine Suche nach dem Schlüssel 15. Rechts das Einfügen des Schlüssels 13

Auch bei *delete(key k)* wird sich zunächst wie beim *search(key k)* verhalten. Ist k im BST nicht vorhanden wird abgebrochen und der BST bleibt un-

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

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

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

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

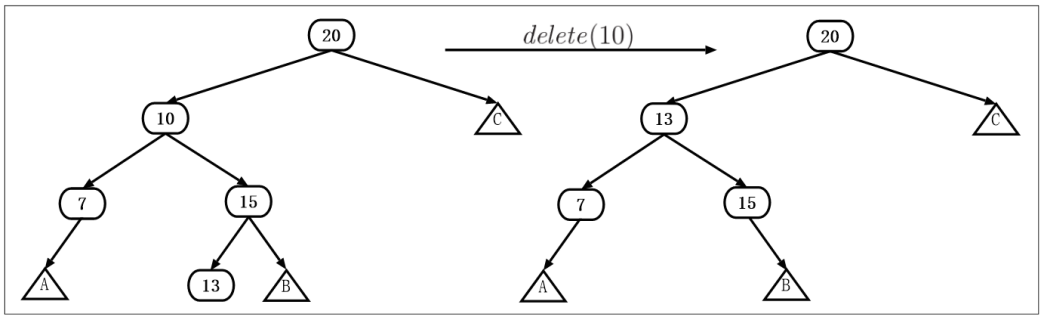


Abbildung 10: Löschen des Schlüssels 10

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

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

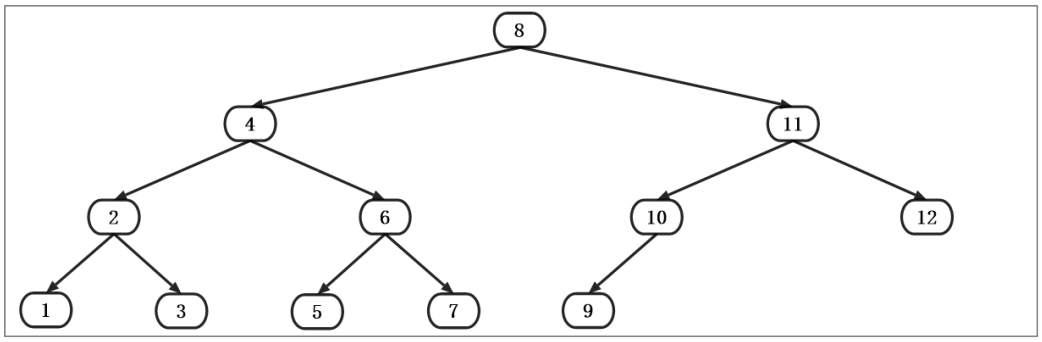


Abbildung 11: Kompletter BST mit 12 Knoten

Lemma 2.1. *Die Höhe eines vollständig balancierten BST 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 maximale Anzahl an Knoten jeder Ebene aufaddiert wird.

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

h ist minimal wenn gilt:

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

Einsetzen:

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

□

3 Rot-Schwarz-Baum

Der Tango Baum verwendet intern eine Hilfsdatenstruktur. Der Rot-Schwarz-Baum gehört zur Gruppe der **balancierten BST** und erfüllt alle Eigenschaften, um ihn als Hilfsdatenstruktur im Tango Baum verwenden zu können.

Genau das ist auch die Rolle des Rot-Schwarz-Baumes in dieser Ausarbeitung. Bei balancierten BST 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 RBT. Fehlende Kinder von Knoten mit gewöhnlichem 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 $(v_0, v_1, ..., v_n)$ ein Pfad von einem Knoten v_0 zu einem Blatt v_n. Die Anzahl der schwarzen Knoten innerhalb $(v_1, ..., v_n)$ 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 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 Schwarz-Höhe 0 hat. Die Schwarz-Höhe eines Knoten x ist genau dann eindeutig, wenn er Eigenschaft 5 nicht verletzt. Hält x 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 leeren Baum verwenden werden könnte. Auch diese Konstellation erfüllt aber die Eigenschaften, so dass sie kein Problem darstellt.

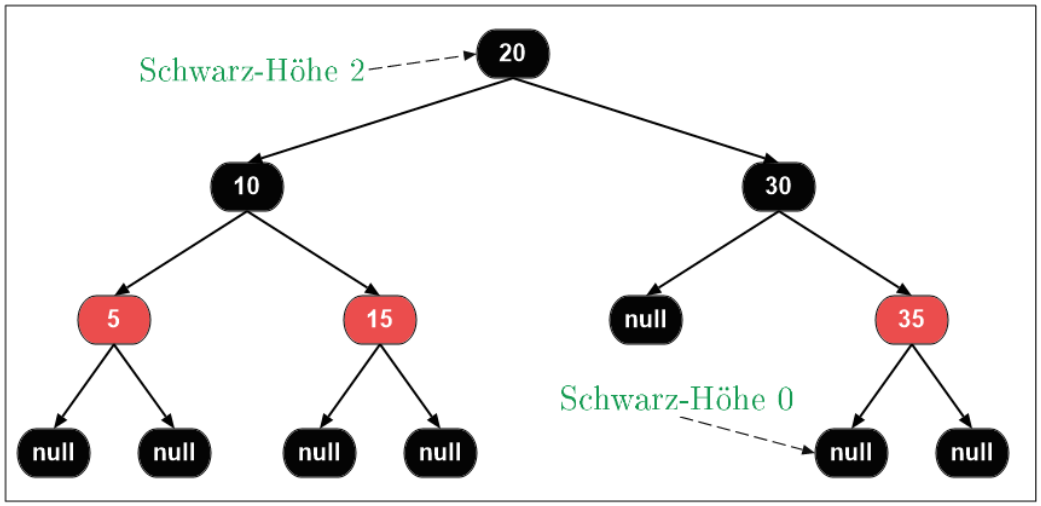


Abbildung 12: Rot-Schwarz-Baum ohne Verletzung von Eigenschaften.

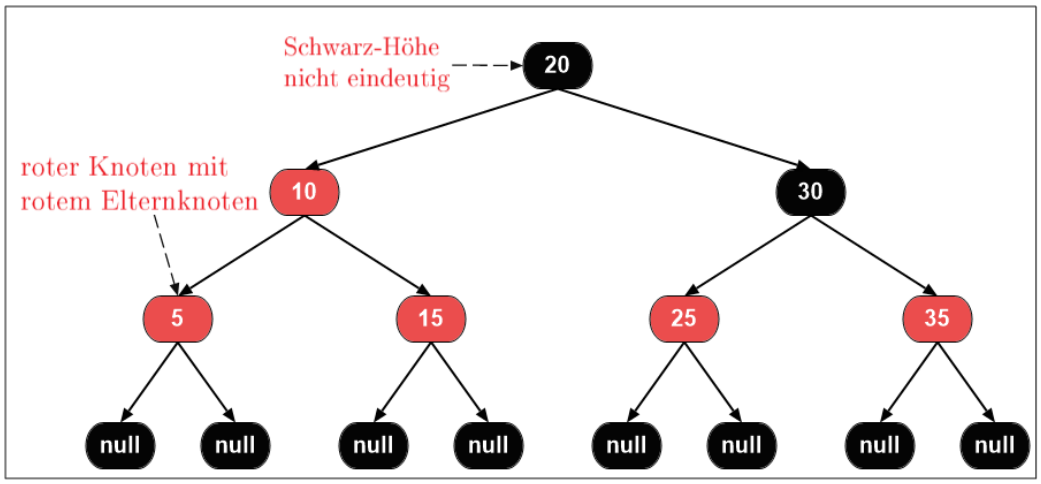


Abbildung 13: Rot-Schwarz-Baum bei dem Eigenschaft vier und fünf verletzt sind.

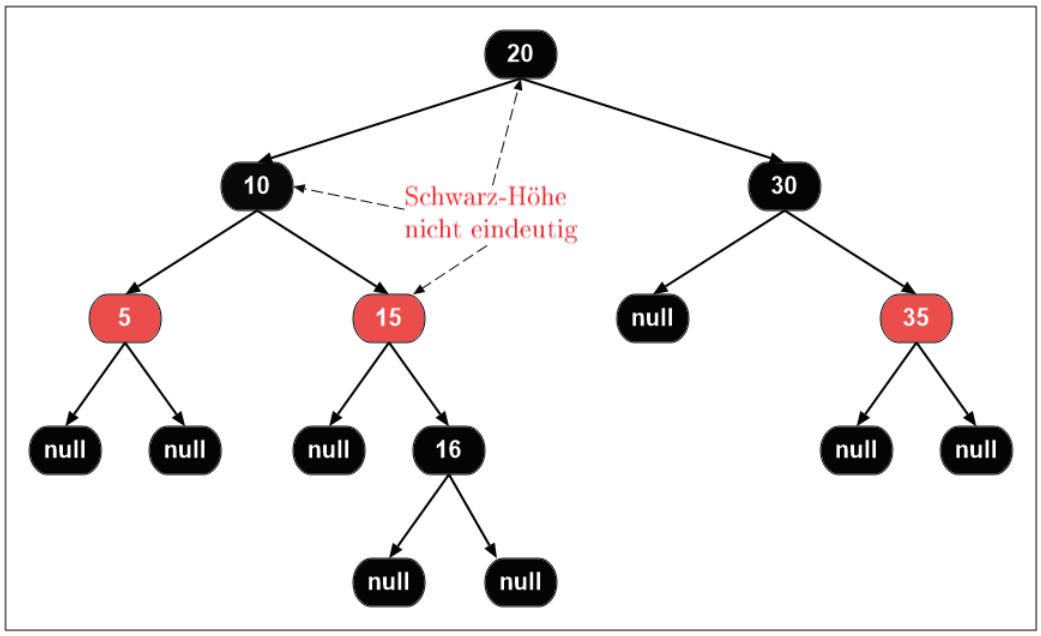


Abbildung 14: Rot-Schwarz-Baum bei dem Eigenschaft fünf verletzt ist.

3.1 Grundoperationen

Suchen im Rot-Schwarz-Baum Die Suche unterscheidet sich nur in einem Punkt von der in 2.2 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 zurückzugeben. Die Operation verändert den RBT nicht.

Einfügen in den Rot-Schwarz-Baum *insert* wird für eine Hilfsdatenstruktur zum 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. Ein neu erzeugter roter Knoten v_k mit Schlüssel k und Schwarz-Höhe 1 nimmt den Platz von b ein. k werden Sonderknoten als linkes und rechtes Kind angefügt. 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 den leeren Baum eingefügt, so ist der neu eingefügte rote Knoten die Wurzel, was eine Verletzung darstellt. Waren bereits Knoten 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. Der neue Knoten 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. Die Schwarz-Höhe keines anderen Knotens hat sich verändert, denn den Platz eines schwarzen Knoten mit Schwarz-Höhe 0 nimmt nun 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 verwenden werden, dass die vorherigen Fallbedingungen nicht erfüllt sind. 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 RBT:** Dieser Fall wird behandelt in dem die Wurzel schwarz gefärbt wird. Man muss noch zeigen, dass es durch das Umfärben zu keiner anderen Verletzung gekommen ist.

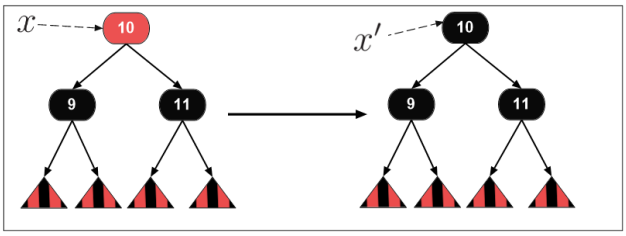


Abbildung 15: *insertFixup*. Dargestellt ist Fall 1

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, ansonsten wäre Eigenschaft vier an zwei Knoten verletzt.

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 umgefärbte 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. z' Schwarz-Höhe steigt um eins gegenüber 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.

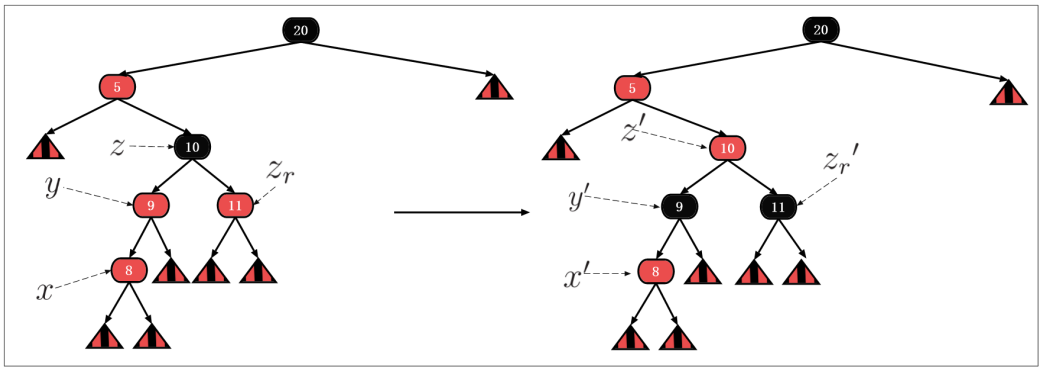


Abbildung 16: *insertFixup.* Dargestellt ist Fall 2

Fall 3: x ist ein linkes Kind. 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 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.

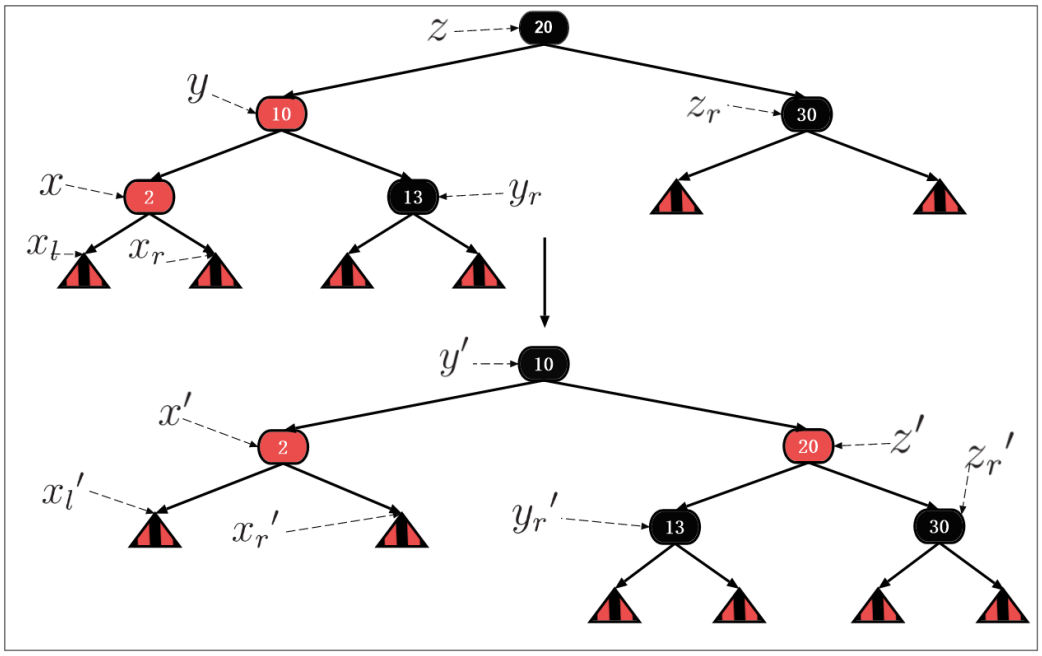


Abbildung 17: *insertFixup*. Dargestellt ist Fall 3

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 Teilbäumen 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 linkes 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'. Links an z' hängt nun ein unveränderter Teilbaum, dessen Wurzel zuvor Geschwisterknoten von y war. Dieser ist nach Fallunterscheidung ein schwarzer Knoten. Links hängt ebenfalls ein unveränderter Teilbaum, dessen Wurzel zuvor das rechte Kind von y war. Das rechte Kind von y muss schwarz sein, ansonsten wäre Eigenschaft vier an zwei Knoten verletzt gewesen. Im Teilbaum mit Wurzel y gibt es also keine aufeinanderfolgenden roten Knoten. 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 ${x_l}'$, ${x_r}'$, ${y_r}'$ 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 $bh(x') = bh(z') = bh(z)$. y' Kinder sind die roten Knoten x' und z'. Da beide Kinder rot sind gilt $bh(y') = bh(x') = bh(z)$. Somit sind alle Schwarz-Höhen im betrachteten Teilbaum eindeutig. Die neue Wurzel des Teilbaumes y' hat die gleiche Schwarz-Höhe und die gleiche Farbe wie die vorherige Wurzel 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. 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.

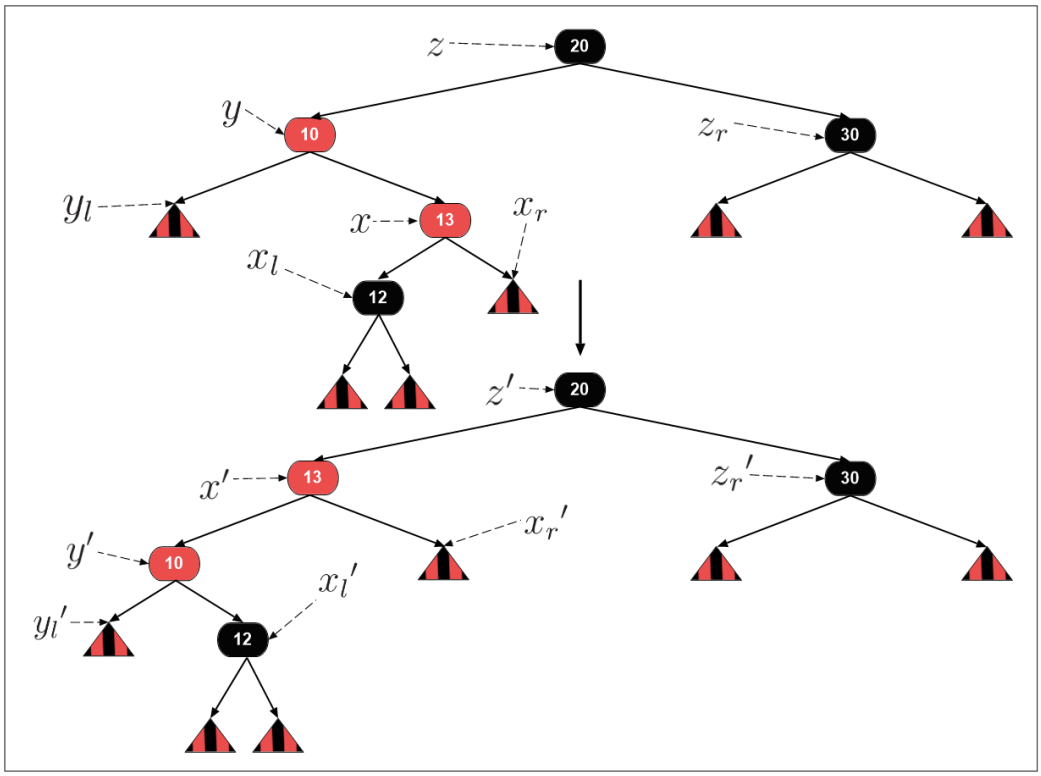


Abbildung 18: *insertFixup*. Dargestellt ist Fall 4

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. 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. Da x und y rot sind, müssen z, x_l und x_r schwarz sein. Nach der Rotation ist y' linkes Kind von x'. x' ist Kind vom schwarzen z'. Die weiteren Kinder der Knoten x' und y' sind ${x_l}'$, ${x_r}'$ und ${y_l}'$. Diese müssen schwarz sein, ansonsten hätte es in ursprünglichen Baum an mehr als einem

Knoten eine Verletzung von Eigenschaft vier gegeben. Durch die Rotation verbleibt es also bei einer Verletzung der Eigenschaft vier in der gleiche 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 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 roten Knotens 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. y ist ein rechtes Kind:
Links-Rechts-Symmetrisch zu Fall 3
Fall 6: x ist ein linkes Kind.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 $h/2$ mal ausgewählt werden, bevor x oder y an der Wurzel liegt. Nach einer Iteration bei der nicht Fall 2 ausgewählt wird, terminiert *insertFixup*. Der Aufwand innerhalb jeder Fallbehandlung ist $O(1)$. Für die Gesamtlaufzeit 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 Hilfsdatenstruktur für den Tango Baum keine solche Operation benötigt, entfällt die Beschreibung. In [4] ist eine detaillierte Beschreibung enthalten.

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

Lemma 3.1. *Für die Höhe h eines RBT 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 ein einzelner Sonderknoten. Induktionsschritt mit Höhe $h + 1$:
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 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 Induktionsnahme mindestens $2^{bh(w)-1} - 1$ innere Knoten. Aufaddieren ergibt die Behauptung.

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

Daraus folgt:

$$\Rightarrow log_2(m + 1) \geq bh(w)$$

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. Daraus folgt.

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

□

search und *insert* haben also 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 $x_1, x_2, ...x_m$ wird also zunächst *access(x_1)* ausgeführt, dann *access(x_2)* usw. m ist die Länge von X. Bei BST wird bezüglich Zugriffsfolgen zwischen online und offline Varianten unterschieden. Bei **offline BST** ist die Zugriffsfolge zu Beginn bereits bekannt, somit kann ein Startzustand gewählt werden, der die Kosten minimiert. Beim **online BST** ist die Zugriffsfolge zu Beginn nicht bekannt. Bei einer worst case Laufzeitanalyse muss somit von dem Startzustand ausgegangen werden, bei dem die Kosten am höchsten sind. In dieser Arbeit werden *access* Operation betrachtet die folgende Eigenschaften einhalten:

1. Die Operation verfügt über genau einen Zeiger p in den BST. Dieser wird zu Beginn so initialisiert, dass er auf die Wurzel zeigt. Terminiert die Operation muss p auf den Knoten mit 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 durchgeführten Einzelschritte während 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. In [5] wurde gezeigt, dass der Zustand eines BST mit maximal $2n - 2$ Rotationen in jeden anderen BST mit der gleichen Schlüsselmenge überführt werden kann. Da bei der Berechnung der Kosten für $OPT(X)$, m ebenfalls als Summand vorkommt, können die zusätzlichen Kosten der online Varianten, für $m > n$ asymptotisch betrachtet vernachlässigt werden.
Als **dynamisch optimal** wird ein BST bezeichnet, wenn er eine beliebige Zugriffsfolge X in $O(OPT(X))$ Zeit ausführen kann. Ein BST der jede Zugriffsfolge in $O(c \cdot OPT(X))$ Zeit ausführt, wird als **c-competitive** bezeichnet. Es konnte bis heute für keinen BST bewiesen werden, dass er dynamisch optimal ist. Es wurden aber mehrere untere Schranken für $OPT(X)$ gefunden. Eine davon wird nun vorgestellt.

4.2 Erste untere Schranke von Wilber

Robert Wilber hat in [6] zwei Methoden zur Berechnung unterer Schranken für die Laufzeit von Zugriffsfolgen bei BST vorgestellt. Hier wird auf die Erste davon eingegangen. Im folgenden werden offline BST betrachtet, bei denen während einer *access(k)* Operation, der Knoten mit Schlüssel k, durch Rotationen zur Wurzel des BST gemacht wird. Ein solcher BST wird als **standad offline BST** bezeichnet. Asymptotisch betrachtet entsteht hierdurch kein Verlust der Allgemeinheit. Sei v_p der Knoten auf den p zum Zeitpunkt t direkt vor der Terminierung von *access* zeigt. Sei d die Tiefe von v_p . Dann sind mindestens Kosten $d + 1$ entstanden. Mit d Rotationen kann v_p zur Wurzel gemacht werden und mit d weiteren Rotationen kann der Zustand zum Zeitpunkt t wieder hergestellt werden. Für einen BST T mit Schlüsselmenge K_T und einer Zugriffsfolge X notieren wir die minimalen Kosten eines wie eben vorgestellt arbeitenden BST mit $W(X, T)$. Im folgenden wird angenommen, dass $K = \{i \in \mathbb{N} | i \in [j, k] \ \ mit\ j, k \in \mathbb{N}\}$ gilt. Dadurch entsteht kein Verlust der Allgemeinheit, denn anderenfalls könnte man die Schlüsselmenge einfach aufsteigend sortiert mit j startend durchnummerieren. Eine Rotation wird innerhalb dieses Kapitels mit (i, j) notiert. i ist dabei der Schlüssel des Knotens v auf dem die Rotation ausgeführt wird. j ist der Schlüssel des Elternknoten von v, vor Ausführung der Rotation. Aus einer Folge von Rotationen $R = (i_1, j_1), (i_2, j_2), .., (i_m, j_m)$ erhält man die Folge $R_x^y = (i_{1'}, j_{1'}), (i_{2'}, j_{2'}), .., (i_{m'}, j_{m'})$, in dem man aus R jede Rotation entfernt

bei der $i \notin [x, y] \vee j \notin [x, y]$ gilt. Ähnlich erhält man aus X die Zugriffsfolge X_x^y, in dem aus X alle Schlüssel k entfernt werden, für die $k < x \vee k > y$ gilt.

lower bound tree Ein lower bound tree Y zu T ist ein BST, der genau $2|K| - 1$ Knoten enthält. Seine $|K|$ Blätter enthalten die Schlüssel aus K. Die $|K| - 1$ inneren Knoten enthalten die Schlüssel aus der Menge $\{r \in R | \exists i, j \in K \colon (i + 1 = j \wedge r = i + 0,5)\}$. Y kann immer erstellt werden indem zunächst ein BST Y_i mit den inneren Knoten von Y erzeugt wird. Ein Blatt wird dann an der Position angefügt, an der die Standardvariante von *insert* angewendet auf Y_i ihren Schlüssel einfügen würde. Dass hierbei für zwei Blätter mit Schlüssel k_1, k_2 die gleiche Position gewählt wird ist ausgeschlossen, da es einen inneren Knoten mit Schlüssel k_i so geben muss dass $(k_1 < k_i < k_2) \vee (k_1 > k_i > k_2)$ gilt. An der Konstruktionsanleitung ist zu erkennen, dass zu den meisten BST mehrere mögliche lower bound trees existieren. Abbildung 19 zeigt eine beispielhafte Konstellation.

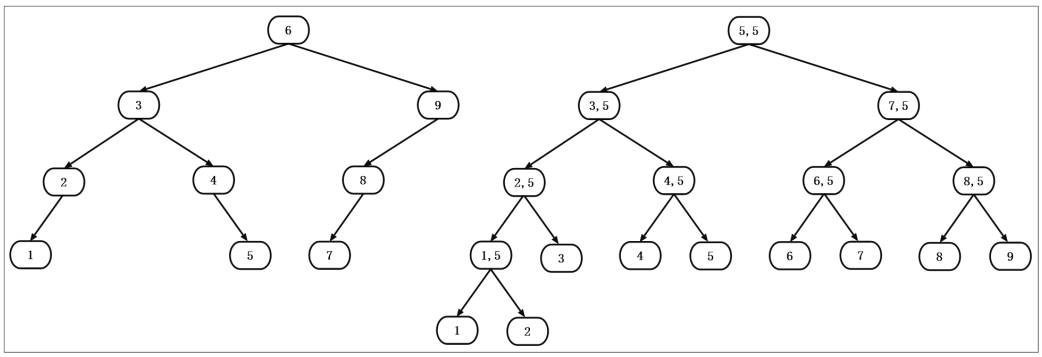


Abbildung 19: Rechts ist ein möglicher lower bound tree zum linken BST dargestellt.

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

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

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

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

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

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

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

5. Ist P_r der leere Pfad weiter mit 7.

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

7. Füge w zu U hinzu, weiter mit 1

Das Verfahren muss terminieren, da die Anzahl der Knoten von T endlich ist. So konstruiert muss T_l^r ein BST sein. Ein Beispiel stellt Abbildung 20 dar.

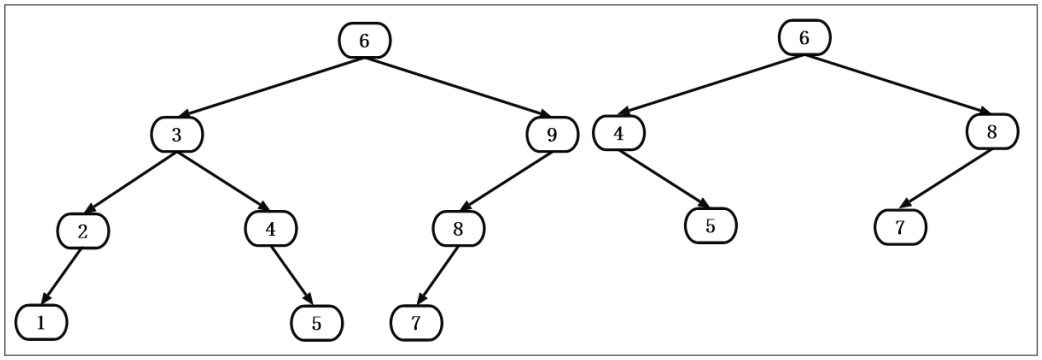


Abbildung 20: Links ein BST T. Rechts ein davon abgeleiteter BST T_4^8 .

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

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

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

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

Nun kommen wir zum Lemma:

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

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

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 $key(u)$ oder $key(v)$ in $[l, r]$ liegt.

1. u ist das linke Kind von v und $key(u) < l$:
   Sei w ein Knoten aus T_l^r und w' einer aus ${T'}_l^r$, mit $key(w) = key(w')$ und $key(w) \in [l, r]$. Es muss gezeigt werden, dass wenn w ein linkes bzw. rechtes Kind mit Schlüssel k hat, dann gilt dies auch für w'. Da $key(u) < l \leq key(w)$ gilt, kann weder u noch v im rechten Teilbaum von w liegen. Somit ist bezüglich der rechten Kinder nichts zu zeigen. Sei P_l der linke innere Pfad von w. Ist v nicht in P_l enthalten und gilt $v \neq w$ dann gilt $P_l = P_l{}'$. Sei $w = v$, dann gilt $P_l = u \circ P_l{}'$, vergleiche Abbildung 7, und da $key(u) < l$, bleibt das linke Kind von w unverändert. Nun sei v in P_l enthalten. Dann unterscheiden sich P_l und $P_l{}'$ dadurch, dass ein Knoten mit $key(u)$ in P_l' enthalten ist. Mit $u < l$ gilt aber, dass sich w und w' bezüglich des Schlüssels ihres linken Kindes nicht unterscheiden.

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

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

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

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

1. Sei u_r das rechte Kind von u. Sei $u, u_r, v_1, v_2, .., v_n$ der linke innere Pfad von v, dann ist $({u_r}', {v_1}', {v_2}', .., {v_n}')$ der linke innere Pfad von v'. Es gilt $l \leq key\,(u) < key\,(u_r) < key\,(v) \leq r$. Damit ist ${u_r}'^*$ das linke Kind von v'^*.

2. Sei $v_1, v_2, .., v_n$ der rechte innere Pfad von u, dann ist $(v', {v_1}', {v_2}', .., {v_n}')$ der rechte innere Pfad von u'. Damit v'^* ist das rechte Kind von u'^*.

3. Ist v das linke bzw. rechte Kind eines Knoten z mit $key(z) \in [r, l]$, dann sei $v, v_1, v_2, .., v_n$ der linke bzw. rechte innere Pfad von z. Dann ist $(u', v', {v_1}', {v_2}', .., {v_n}')'$ der linke bzw. rechte innere Pfad von z'. Dann u'^* das linke bzw. rechte Kind von z'^*.

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

□

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

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

bzw. rechte Teilbaum von w. Es ist zu beachten, dass Y^1 ein lower bound tree zu $T_1^{k_w}$ ist und Y^2 einer zu $T_{k_w}^{\infty}$. $T_{i1}^{k_w}$ wird im folgenden als $T_i{}^1$ bezeichnet und $T_{ik_w}^{\infty}$ als $T_i{}^2$. Da $n \geq 2$ muss w ein innerer Knoten sein. Sei $R^1 = r^1{}_1, r^1{}_2, .., r^1{}_{l^1} = R_1^{k_w}$ und $R^2 = r_1^2, r_2^2, .., r_{l^2}^2 = R_{k_w}^{\infty}$. Mit M wird die Folge bezeichnet, die entsteht, wenn aus R alle Rotationen entfernt werden, die in R^1 oder R^2 enthalten sind. Sei l_M die Länge von M. Es muss $l = l^1 + l^2 + l_M$ gelten, da keine Rotation sowohl in R^1 als auch in R^2 enthalten sein kann. X_1 ist die Folge die entsteht wenn aus X alle Schlüssel $k > k_w$ entfernt werden. X_2 entsteht durch entfernen aller Schlüssel $k < k_w$ aus X. Für $j \in \{1, 2\}$, sei U^j die Menge der inneren Knoten von Y^j. Sei $T_0^{j*}, T_1^{j*}, .., T_{l^t}^{j*}$ die entstehende Folge, wenn aus $T_0^j, T_1^j, .., T_l^j$ die T_t^j entfernt werden für die $T_{t-1}^j = T_t^j$ gilt.
Mit Lemma 4.1 kann T_t^{j*} durch Ausführung der Rotation r_t^j auf T_{t-1}^{j*} abgeleitet werden. Dadurch folgt durch dieses Lemma, dass wenn ein Knoten mit Schlüssel $k < w$ bzw. $k > w$ die Wurzel von T_t ist dann muss die Wurzel von T_t^1 bzw. T_t^2 auch Schlüssel k haben. R^j bringt also der Reihe nach, die Knoten mit den Schlüsseln aus X^j an die Wurzel von T^j und X^j kann als Zugriffsfolge für T^j aufgefasst werden. Da die Knotenzahl in T^j kleiner n sein muss gilt mit der Induktionsvoraussetzung $l_j \geq \sum_{u \in U^j} score(u)$.
Sei $\sigma = key(w) \circ X$. Sei a eine w-Transition. Nun wird angenommen dass $\sigma_{a-1} < key(w) \wedge \sigma_a > key(w)$. Der andere Fall kann davon problemlos abgeleitet werden. Sei y der Knoten in T mit $key(y) = \sigma_{a-1}$ und z der Knoten in T mit $key(z) = \sigma_a$. Nach *access(σ_{a-1})* ist y die Wurzel von T. z muss sich im rechten Teilbaum von y befinden. Nach *access(σ_a)* ist z die Wurzel von T. y muss sich im linken Teilbaum von z befinden. Somit muss während *access(σ_a)* die Rotation $(key(z), key(y))$ ausgeführt worden sein. $(key(z), key(y))$ muss in M enthalten sein. Für jede w-Transition ist also mindestens eine Rotation in M enthalten, also gilt $l_M \geq score\,(w)$.
Zusammengefasst ergibt sich:

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

□

Daraus folgt direkt $OPT\,(X) = \Omega\,(_X(T, Y, X))$ für beliebige BST T.

4.3 Bit reversal permutation

In diesem Abschnitt wird gezeigt, dass es Zugriffsfolgen mit Länge m gibt, so dass für die Laufzeit beliebiger BST T $\Omega\,(m \log n)$ gilt, mit n ist die Anzahl der Knoten von T. Hier werden speziell die Zugriffsfolgen betrachtet, die als

bit reversal permutation bezeichnet werden. Dafür wird die erste untere Schranke der von Wilber verwendet und ein Beweis ist ebenfalls in [6] enthalten.
Nun wird zunächst der Aufbau einer solchen Zugriffsfolge eingegangen. Sei $l \in \mathbb{N}$ und $i \in \{0, 1, .., l-1\}$. Eine Folge $b_{l-1}, b_{l-2}, .., b_0$ mit $b_i \in \{0, 1\}$, kann als Zahl zur Basis 2 interpretiert werden. T enthält alle Schlüssel die als solche Folge dargestellt werden können. Die Schlüsselmenge von T ist deshalb $K_l = \{0, 1, .., 2^l - 1\}$. Die Funktion $br_l(k)\colon K \to K$ ist wie folgt definiert. Sei $b_{l-1}, b_{l-2}, .., b_0$ die Binärdarstellung von k, dann gilt

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

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

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

Beweis. Sei U die Menge der inneren Knoten von Y. Mit Satz 4.1 reicht es aus

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

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

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

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

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

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

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

Zusammenfassen ergibt

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

□

Die Schlüsselmenge wurde beim Korollar auf $K_l = \{0, 1, .., 2^l - 1\}$ festgelegt. Vielleicht wäre es aber mit einer anderen Schlüsselmenge K möglich X schneller auszuführen ? In jedem Fall müsste $K_l \subsetneqq K$ gelten. Sei R die Folge von Rotationen, die beim Ausführen von X bei einem BST T mit Schlüsselmenge K entsteht. Sei $y = 2^l - 1$ Mit Lemma 4.1 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 Anschnitt werden die Kosten von amortisierten Laufzeitanalysen verwendet. Deshalb wird diese hier nun vorgestellt. Sei $i \in \{0, .., m\}$. Bei der **amortisierten Laufzeitanalyse** wird eine Folge von m Operationen betrachtet. Hierbei kann es sich m mal um die gleiche Operation handeln, oder auch um verschiedene. Die **tatsächlichen Kosten** t_i stehen für die Kosten zum ausführen der i-ten Operation. Durch aufaddieren der tatsächlichen Kosten jeder einzelnen Operation erhält man **tatsächlichen Gesamtkosten**. Stehen für die Laufzeit der Operationen jeweils nur obere Schraken zur Verfügung, kann man mit diesen genau so vorgehen, um eine obere Schranke für die Gesamtlaufzeit zu erhalten. So erzeugte obere Schranken können jedoch unnötig hoch sein. Die Idee bei einer amortisierten Analyse ist es, eingesparte Zeit durch schnell ausgeführte Operationen, den langsameren Operationen zur Verfügung zu stellen. Dabei wird insbesondere der aktuelle Zustand der zugrunde liegenden Datenstruktur vor und nach einer Operation betrachtet. Es gibt drei Methoden zur amortisierten Analyse, bei BST wird in der Regel die **Potentialfunktionmethode** verwendet.

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

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

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

Folgendes gilt für die Summe der t_i:

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

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

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

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

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

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

4.5 Eigenschaften eines dynamisch optimalen BST

Im folgendem werden einige obere Laufzeitschranken für 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 $K = \{1, 2, .., n\}$ angenommen. Wenn nicht anders angegeben wird $X = x_1, x_2, .., x_m$ als Zugriffsfolge verwendet. Es wird $m \geq n$ angenommen.

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

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

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

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

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

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

Zeit ausführt. Der Name kommt daher, dass es sich hierbei um eine untere Schranke für die Ausführungszeit von X bei statischen BST handelt, siehe [7]. Solche ändern ihre Struktur während *access* nicht.

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

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

gilt.

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

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

gilt.
Abbildung 21 zeigt Implikationen zwischen den Eigenschaften und basiert auf einer Abbildung aus 21.

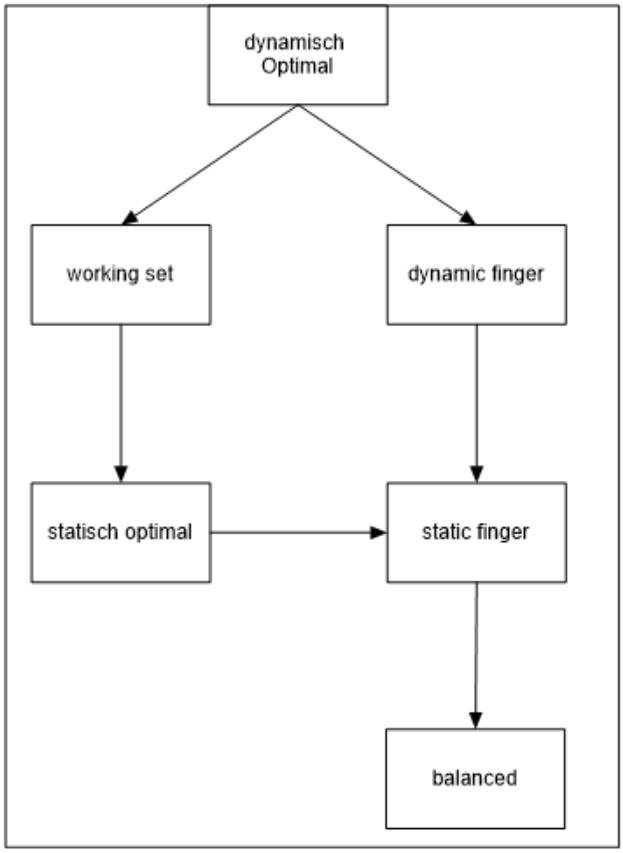


Abbildung 21: Implikationen zwischen den Eigenschaften, abgeleitet aus einer Abbildung aus [8]

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 in [2], von Demaine, Harmon, Iacono und Patrascu beschrieben, 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 Interleave Lower Bound

Sei $X = x_1, x_2, ., x_m$ eine Zugriffsfolge und sei $K = \{k \in \mathbb{N} | k \textit{ ist in } X \textit{ enthalten}\}$. 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 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 Wurzel v und r der größte. Sei $X_l^r = x_{1'}, x_{2'}, .., x_{m'}$ wie in Abschnitt 4.2 definiert. $i \in \{2, 3, .., m'\}$ 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 *inScore(u)* die Funktion die zu dem Knoten $u \in U$ die Anzahl der Interleaves durch u zurückgibt. Die Funktion $IB(X)$ ist definiert durch:

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

Sei T_0 der BST mit Schlüsselmenge K auf der X ausgeführt wird. Für $i \in \{1, 2, .., m\}$ sei T_i der BST, der entsteht nachdem $access(x_i)$ auf T_{i-1} ausgeführt wurde. Zu $u \in U$ und $j \in \{0, 1, .., m\}$ gibt es einen **transition point** v in T_j. v ist ein Knoten mit 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_i ist kein Knoten mit Eigenschaft 1 und 2 enthalten, der eine kleinere Tiefe als v hat.

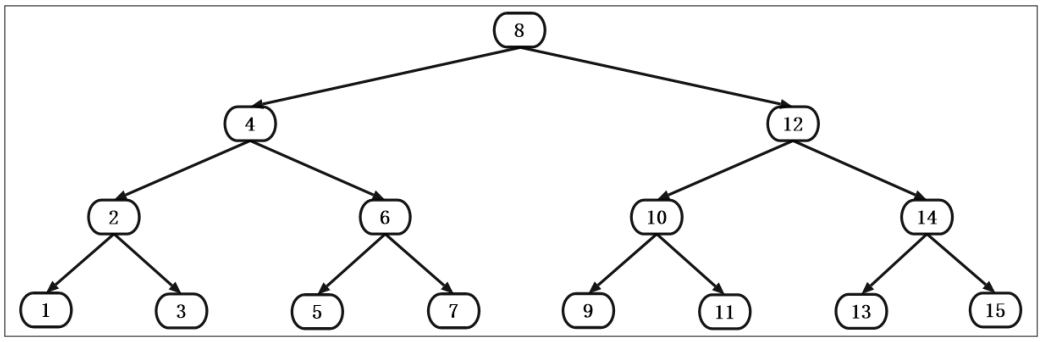


Abbildung 22: Der lower bound tree zur Zugriffsfolge $1, 2, .., 15$

Im Beweis dieses Abschnittes wird gezeigt das $OPT(X) \geq \frac{IB(X)}{2} - n$ gilt, wobei n die Anzahl der Knoten im lower bound tree ist. Dafür werden jedoch noch drei Lemmas zu den Eigenschaften von Y benötigt.

Lemma 5.1. *Sei $X = x_1, x_2, .., x_m$ eine Zugriffsfolge und Y ein zu X erstellter lower bound tree mit Schlüsselmenge K. Sei T_0 der BST mit Schlüsselmenge K auf dem X ausgeführt wird. Für $i \in \{1, 2, .., m\}$ sei T_i der BST der durch Ausführen von access(x_i) auf $T_{(i-1)}$ entsteht. Sei U die Menge der Knoten von Y, bei denen die rechte Region nicht leer ist. Dann gibt es zu jedem Knoten $u \in U$ und $j \in \{0, 1, .., m\}$ 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 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 4.2. Sei w der gemeinsame Vorfahre aller Schlüssel aus der linken und der rechten Region von u in T_l^r 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 größten 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 $P_u = v_0, v_1, .., v_r$ 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 $v_0, v_1, .., v_r$ 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 Knoten nicht verändern kann, solange v nicht wenigstens einmal der berührte Knoten war. In den zwei verbleibenden Lemmas und dem Satz seien T_j, X, Y, U und u wie in Lemma 5.1 definiert.

Lemma 5.2. *Sei v der transition point zu u in T_j. Sei $l \in N$, mit $j < l \leq m$. Gilt für alle x_i, mit $i \in [j, l]$, während der Ausführung von access(x_i), v war nicht wenigstens einmal der berührte Knoten, dann ist v während der gesamten Ausführungszeit von $access(x_j), access(x_{j+1}), .., access(x_l)$ 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 $access(x_j), access(x_{j+1}), .., access(x_l)$ der gemeinsame Vorfahre der Schlüssel aus der rechten Region von u, mit der größten Tiefe. Knoten mit Schlüssel in der linken Region von u könnten berührt werden. Zu einem Ausführungszeitpunkt t kann deshalb ein Knoten $v_{lt} \neq v_l$ mit einem Schlüssel aus der linken Region von u der gemeinsame Vorfahre der Knoten mit diesen Schlüsseln mit der größten Tiefe sein. Da v_r nicht berührt wird kann zu keinem Zeitpunkt v_l im Teilbaum mit 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 Wurzel 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 $key(v_1)$ nicht im Teilbaum mit Wurzel u_2 enthalten, so kann v_1 nicht der transition point von u_2 sein.

2. $key(v_1)$ ist im Teilbaum mit Wurzel u_2 enthalten und $key(v_1)$ ist in der linken Region von u_1 enthalten:
   Da u_1 Vorfahre von u_2 ist, müssen alle Schlüssel im Teilbaum mit 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. $key(v)$ muss somit der Schlüssel von w_l bzw. w_r sein, je nachdem wessen Tiefe kleiner ist. Denn andererseits könnte man einen Pfad von der Wurzel von T_j zu v angeben der zwei Knoten aus der linken Region von u_1 enthält, dass ist jedoch ein Widerspruch dazu, dass $key(v_1)$ in der linken Region von u_1 enthalten ist und v_1 zudem der transition point für u_1 ist.
   w ist entweder der Knoten w_l oder w_r je nachdem wessen Tiefe größer ist, somit gilt $v \neq w$.

3. $key(v_1)$ ist im Teilbaum mit Wurzel u_2 enthalten und $key(v_1)$ ist in der rechten Region von u_1 enthalten:
   Symmetrisch zu Fall 2.

□

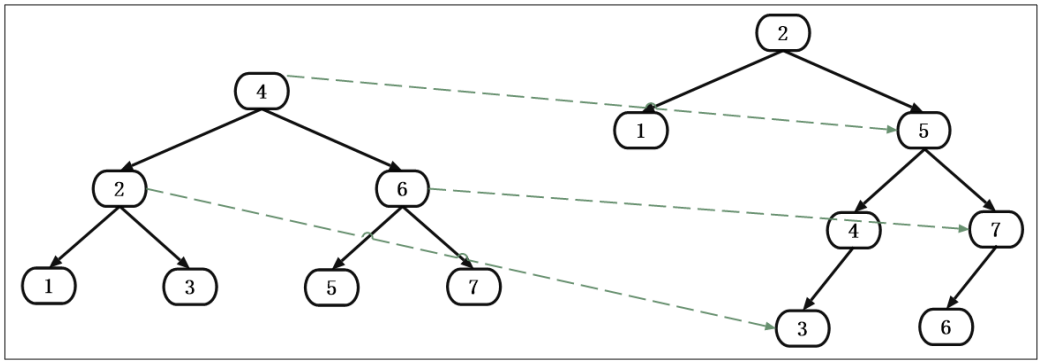


Abbildung 23: Transition point Zuordnung. Links ein lower bound tree, rechts ein möglicher T_j.

Satz 5.1. *Sei $X = x_0, x_1, .., x_m$ 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. Durch Lemma 5.3 kann die Anzahl der Berührungen für jedes $y \in Y$ einzeln bestimmt werden, diese müssen dann lediglich noch aufaddiert werden. Sei l, r, v_r und v_l wie in Lemma 5.1 zu y definiert, so dass entweder v_l oder v_r der transition point zu y sein muss, je nachdem welcher der beiden Knoten die größere Tiefe hat. Sei $X_l^{r\prime} = x_{i_0}, x_{i_1}, .., x_{i_p}$ 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 y wie x_{k-1}. Damit gilt $inScore(y) = p$. Nun wird angenommen, dass die x_{i_j} mit j ist gerade in der rechten Region von y liegen, und die x_{i_j} mit j ist ungerade in der linken Region. Der andere Fall kann wieder direkt abgeleitet werden. Sei $q \in \mathbb{N}$ mit $1 \leq q \leq \lfloor p/2 \rfloor$. $access\left(x_{i_{2q-1}}\right)$ muss v_l berühren und $access\left(x_{i_{2q}}\right)$ muss v_r berühren. Sei k_1 der Schlüssel des transition point von y zu Beginn von $access\left(x_{i_{2q-1}}\right)$ und k_2 der Schlüssel des transition point von y zu Beginn von $access\left(x_{i_{2q}}\right)$. Gilt $k_1 = k_2$ so muss der transition point von y in $access\left(x_{i_{2q}}\right)$ berührt worden sein. Gilt $k_1 \neq k_2$ so muss der transition point von y, nach Lemma 5.2, in $access\left(x_{i_{2q-1}}\right)$ berührt worden sein. Aus der Konstruktion von $X_l^{r\prime}$ folgen daraus mindestens $\lfloor p/2 \rfloor \geq p/2-1$ Berührungen des transition point von y. Sei U wie in Lemma 5.1 definiert. Aufaddieren über alle $u \in U$ ergibt bei den Werten der *inScore* Funktion die Interleave Bound und bei den Berührungen von transition points zumindest $IB(X)/2 - |U| \geq IB(X)/2 - n$.

□

5.2 Aufbau des Tango Baums

Wie bereits erwähnt besteht ein Tango Baum T mit Schlüsselmenge K aus Hilfsbäumen. Eine Anforderung an einen Hilfsbaum mit n Knoten ist, dass für seine Höhe $h = O(\log n)$ gilt. T bietet lediglich eine *access* Operation an. Ist T also erst einmal für K erzeugt, ist seine Schlüsselmenge unveränderlich. Sei P der lower bound tree aus Abschnitt 5.1 mit 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 Wurzel p enthalten Schlüssel als Parameter ausgeführt, so hat p kein preferred child. Ansonsten sei $access(k)$ die zuletzt ausgeführte Ope-

ration mit einem Schlüssel der im Teilbaum mit Wurzel p enthalten ist. Liegt k in der linken Region von p, 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 kann man bereits die Kopplung zur interleave lower bound erkennen. 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. Man erkennt sofort, dass der Parameter der letzten *access* Operation 8, 4, 2 oder 1 gewesen sein muss, da man von der Wurzel aus über preferred childs zu den Knoten mit diesen Schlüsseln gelangen kann. 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 childs lassen sich die **preferred path** erstellen. Sei v ein Knoten in P, der nicht preferred child eines anderen Knoten aus P ist. Dann ist der preferred path zu v, der längst mögliche Pfad $(v_0, v_1, .., v_l)$, mit $v_0 = v$ und $\forall i \in \{1, 2, .., l\}$: v_i *ist preferred child von* v_{i-1}.

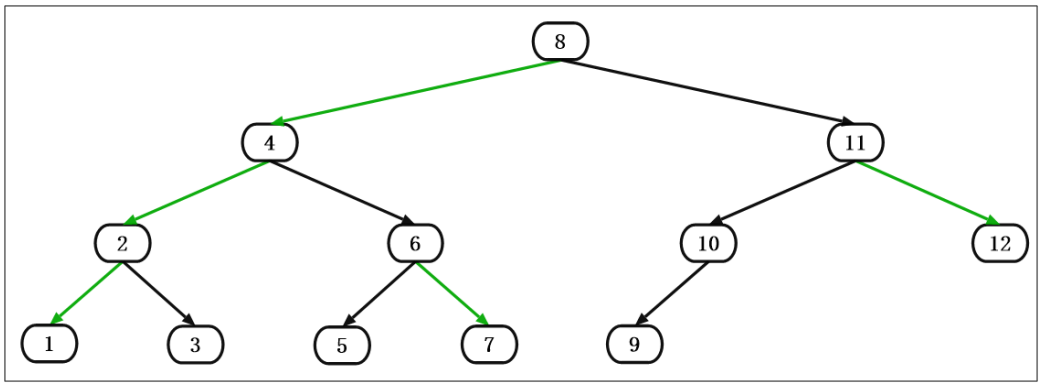


Abbildung 24: Die preferred childs werden durch die grünen Pfeile markiert.

Nun werden die preferred path des BST aus Abbildung 24 angegeben, wobei der Schlüssel jeweils als Bezeichner für den ihn enthaltenden Knoten verwen-

det 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 preferred child eines Knoten sein kann und Knoten die kein preferred child sind als Startknoten eines Pfades dienen, muss jeder Knoten in genau einem preferred Pfad 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.

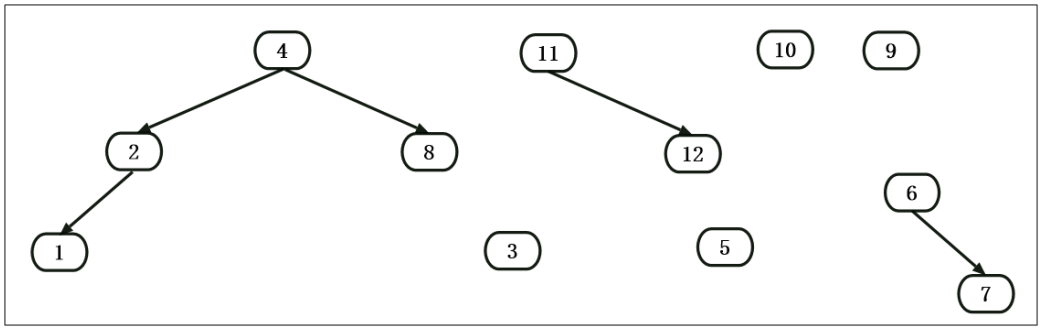


Abbildung 25: Hilfsbäume zu den preferred path aus dem Beispiel.

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 $key\,(v) = key\,(u)$ nicht preferred child seines Elternknotens ist. Sei h_2 der Hilfsbaum, der den Schlüssel $key\,(u)$ 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 Standartvariante 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 mit 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 Wurzel u enthalten. Sei l der kleinste Schlüssel in diesem Teilbaum und r der Größte. In P kann es außerhalb des Teilbaumes mit Wurzel u keinen Schlüssel k mit $l \leq k \leq r$ geben. h_2 kann nur Schlüssel enthalten die in P aber nicht im Teilbaum mit Wurzel u enthalten sind. Ein Vorgänger von a in h_2 muss einen Schlüssel haben der kleiner als l ist. Ein Nachfolger von a in h_2 muss einen Schlüssel haben, der größer als r ist. Im Tango Baum kann es also keine Verletzung der Links-Rechts-Beziehung geben.

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

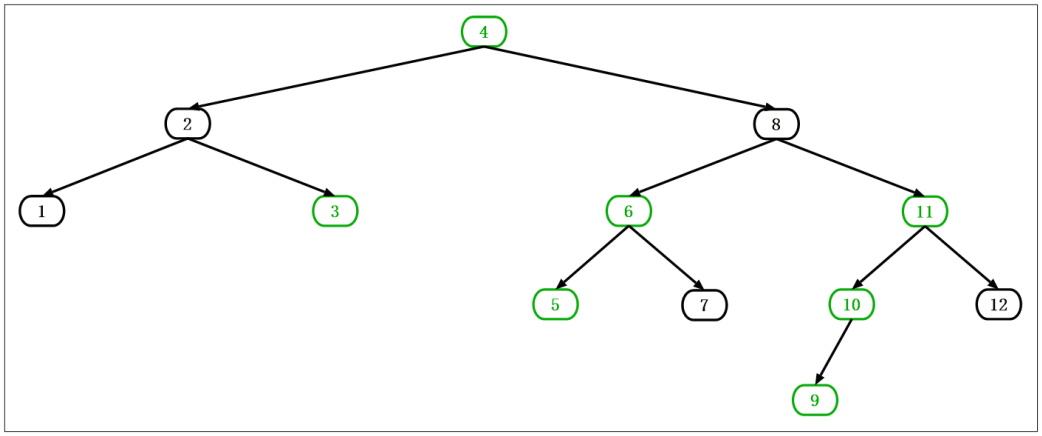


Abbildung 26: Tango Baum zu dem Beispiel.

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

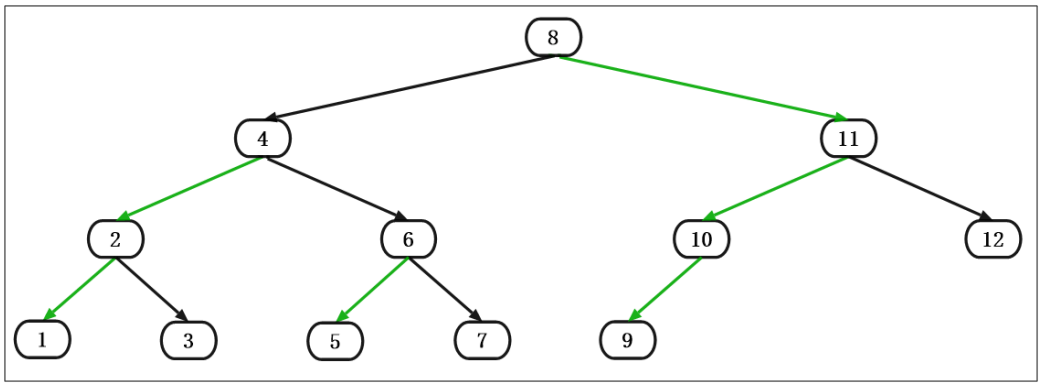


Abbildung 27: Preferred childs nach *access*(9).

Abbildung 28 zeigt einen möglichen Zustand von T'.

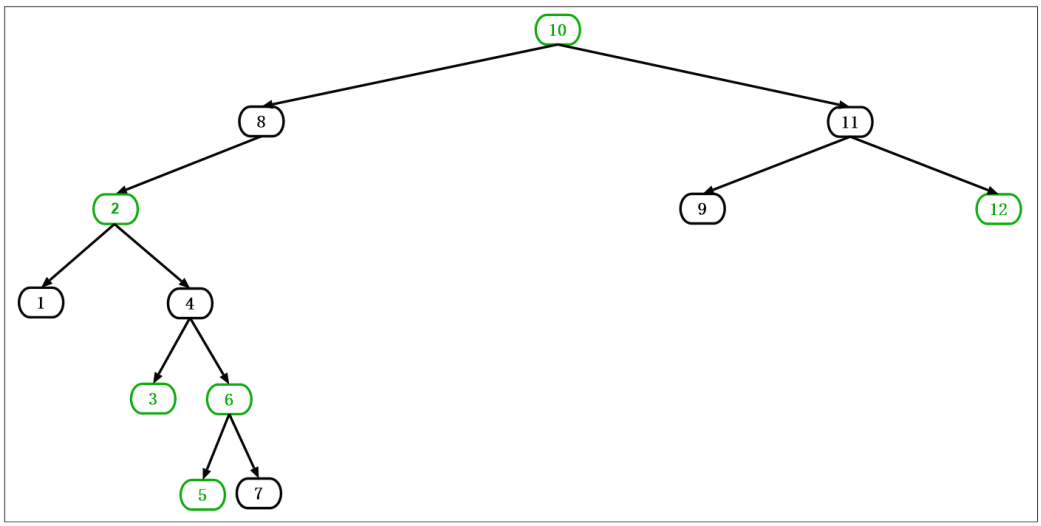


Abbildung 28: Tango Baum nach *access*(9).

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 Wert *true* hat, wenn v die Wurzel eines Hilfsbaumes ist. In einer Konstante *depth* wird die Tiefe des Knoten mit Schlüssel *key* (v) in P gespeichert. Außerdem gibt es noch Variablen *minDepth* und *maxDepth*. Sei

v im Hilfsbaum H enthalten und sei H_v der Teilbaum mit 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 in H_v, dann entspricht *min* dem Wert der *minDepth* Variable von v. Sei *max* der größte Wert aller *depth* Konstanten in H_v, dann entspricht *max* dem Wert der *maxDepth* Variable von v. Auf die Variablen und Konstanten eines Knoten 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* H_1*, key* k*, HB* H_2*)* 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 und K_2 die von H_2. Die Operation kann verwenden, dass für $k_1 \in K_1$ und $k_2 \in K_2$, $k_1 < k < k_2$ gilt. Es gibt drei Fälle. 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 Hilfsbaum H mit Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ zurück.

   (b) $w_1.isRoot = true$:
   Die Operation gibt die Wurzel eines Hilfsbaum H mit Schlüsselmenge $K_2 \cup \{k\}$ zurück. An H ist ein Hilfsbaum H_3 mit Schlüsselmenge K_2 angefügt. *isRoot* der Wurzel von H_3 hat den Wert *true*.

   (c) $w_2.isRoot = true$:
   Die Operation gibt die Wurzel eines Hilfsbaum H mit Schlüsselmenge $K_1 \cup \{k\}$ zurück. An H ist ein Hilfsbaum H_3 mit Schlüsselmenge K_1 angefügt. *isRoot* der Wurzel von H_3 hat den Wert *true*.

   Bei allen Fällen hat *isRoot* 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 einen Knoten mit Schlüssel k existiert. Sei K die Schlüsselmenge von H. Die Operation gibt einen Knoten v mit Schlüssel k zurück. Das linke Kind von v muss die Wurzel eines Hilfsbaumes mit Schlüsselmenge $K_l = \{i \in K \mid i < k\}$ sein. Das rechte Kind von v muss die Wurzel eines Hilfsbaumes mit Schlüsselmenge $K_r = \{i \in K \mid i > k\}$ sein. 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 A_1 und A_2. Es dürfen nur Werte für d übergeben werden zu denen es in A einen Knoten v mit $v.depth = d$ gibt. Wobei die Knoten in A bei denen $depth \leq d$ gilt in A_1 enthalten sind und die mit $depth > d$ in A_2. Die Rückgabe ist die Wurzel eines Hilfsbaumes 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 Knoten l, l', r und r' in A gesucht. l ist der kleinste Schlüssel eines Knoten v_l mit $v_l.depth > d$ in A . r ist der größte Schlüssel eines Knoten v_r mit $v_r.depth > d$ in A . 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. Man startet mit dem Zeiger p an der Wurzel von A. Zeigt p nicht auf v_l, muss es im linken Teilbaum von p einen Knoten v mit $v.depth > d$ geben, und das ist an der *maxDepth* Variable des linken Kindes von p direkt abfragbar. Ist v_l erreicht , kann l' über eine Suche des Vorgängers von v'_l zu gefunden werden. Die Suche nach r und r' verläuft analog.
A besteht aus Schlüsseln 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 $[l, r]$ liegen in P entweder im linken oder im rechten Teilbaum des Knotens mit Schlüssel k.
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.
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 rechtes Kind von v_l.
5. Setze die *isRoot* Variable der Wurzel von D auf *true*.
6. Führe $F = concatenate(D,\ r',\ E)$ aus.
7. Führe $G = concatenate(B,\ l',\ F)$ aus.
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 Baum. Abbildung 29 demonstriert den Ablauf nochmals und Abbildung 30 zeigt einen verkürzten Ablauf bei fehlendem r'

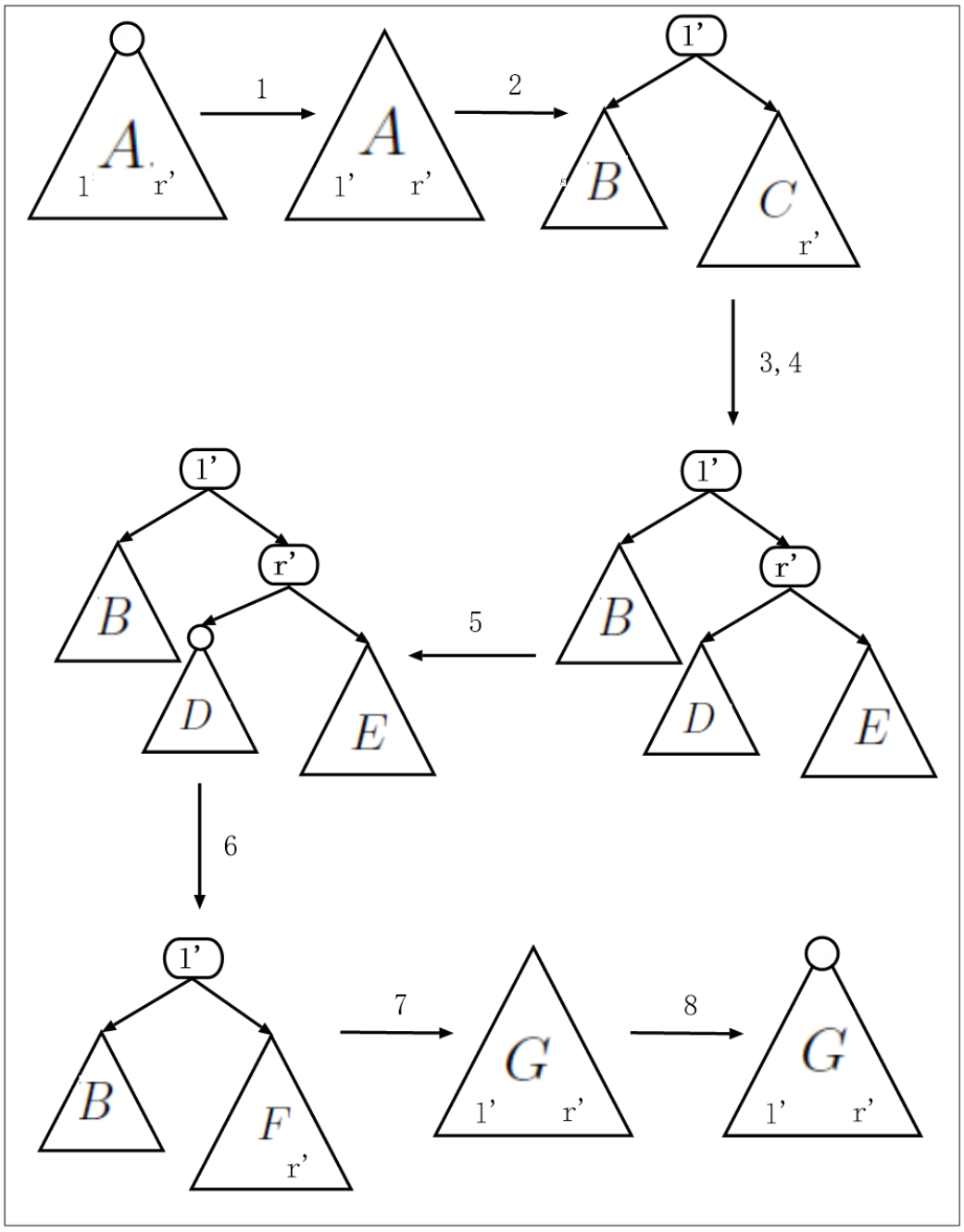


Abbildung 29: Ablauf von *cut(d)*. Die Abbildung basiert auf einer aus [2], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

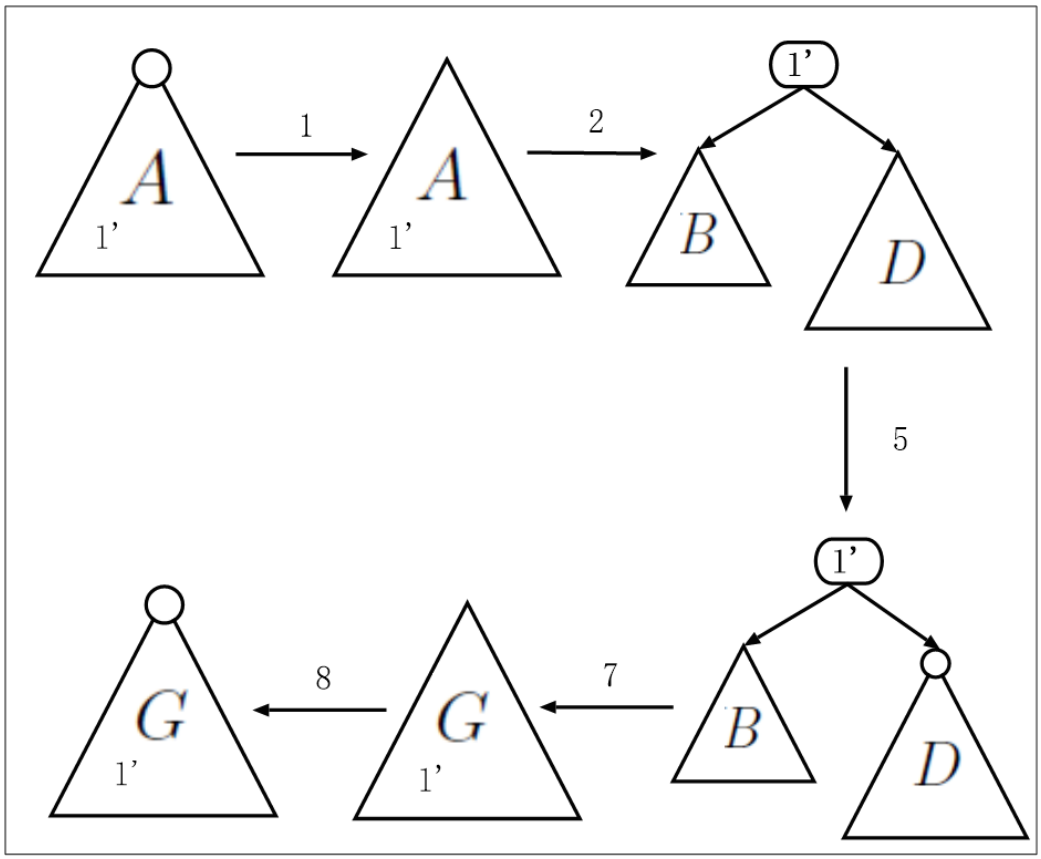


Abbildung 30: Ablauf von *cut(d)* bei fehlenden r'. Die Abbildung basiert auf einer aus [2], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

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 Gesamtlaufzeit $O(\log(n))$.

join Operation *join(HB H_1, HB H_2)* fügt die Hilfsbäume H_1 und H_2 zu einem Hilfsbaum H zusammen. Auch H muss einem preferred path repräsentieren. An die Parameter werden deshalb Anforderungen gestellt. 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 $r' > r$. Wird in H_1 ein Schlüssel aus H_2 gesucht so muss l' bzw. r' zurückgegeben werden. 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 d 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 rechtes Kind von v_l. Setze die Wurzel von H_2 als linkes Kind von v_r.
5. Führe $F = concatenate(H_2,\ r',\ E)$ aus.
6. Führe $H = concatenate(B,\ l',\ F)$ aus.
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 Baum. Abbildung 31 demonstriert den Ablauf nochmals und Abbildung 32 zeigt einen verkürzten Ablauf bei fehlendem r'

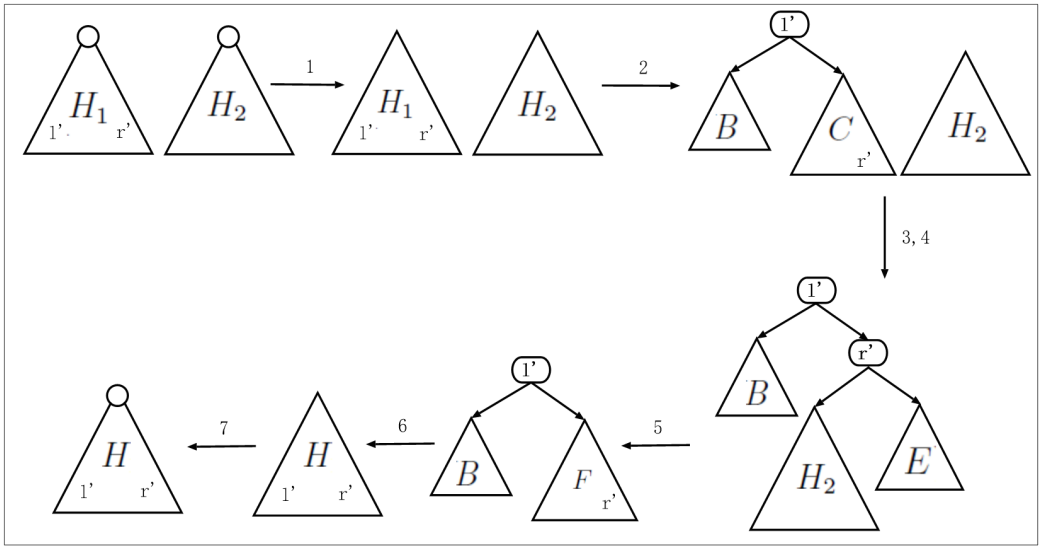


Abbildung 31: Ablauf von *join*(H_1, H_2). Die Abbildung basiert auf einer aus [2], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

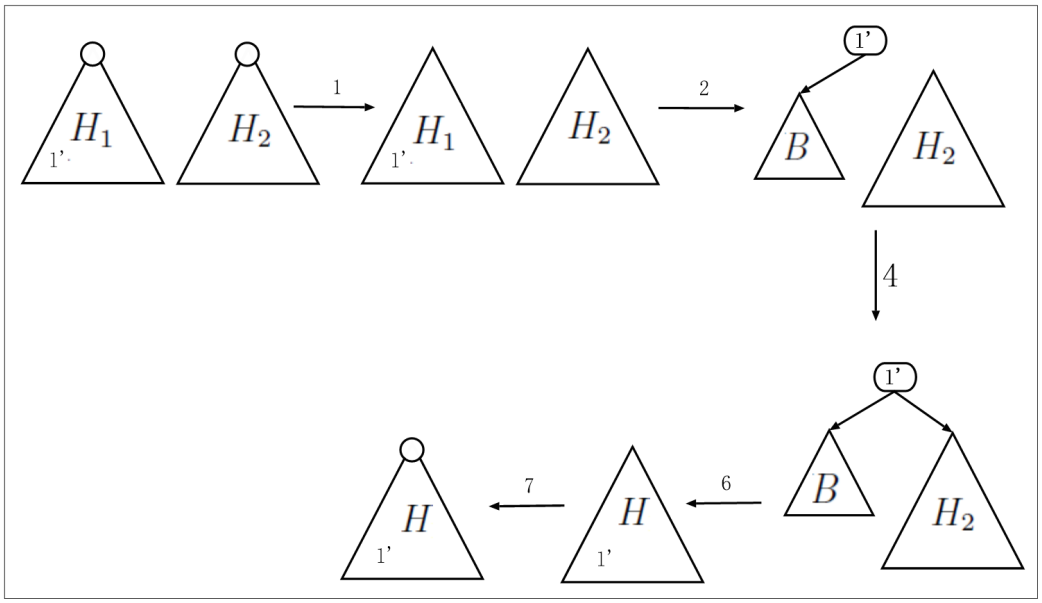


Abbildung 32: Ablauf von *join*(H_1, H_2) bei fehlendem r'. Die Abbildung basiert auf einer aus [2], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

Sei n die Anzahl der Knoten von H. Jeder der neun 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 sich p 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 *key* (k) so wird das preferred child des Knoten mit Schlüssel k in P auf *left* gesetzt. So dass nochmals eine Anpassung notwendig sein kann. Die Operation wird noch etwas detaillierter beschrieben. Zur Vereinfachung bezeichnet T immer den aktuellen Zustand des Tango Baums 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 5. Ansonsten wird die Wurzel eines Hilfsbaumes H_2 erreicht.
3. Sei w_2 die Wurzel von H_2. Führe $H_3 = cut(w_2.minDepth - 1)$ aus.

4. Führe $join(H_3,\ H_2)$ aus. Weiter bei 1.

5. Sei v der Knoten mit $key(v) = k$. Führe $H_3 = cut(v.depth)$ aus.

6. Suche 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 mit 8.

7. Sei H_4 der Hilfsbaum, auf dessen Wurzel p zeigt. Führe $join(H_3,\ H_4)$ aus.

8. Gib p zurück.

Zu klären ist noch, warum im sechsten Punkt, die Wurzel des richtigen Hilfsbaums gefunden werden muss. Seien u und u_l 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 $key\,(v) = key\,(u)$ bzw. $key\,(v_c) = key\,(u_c)$. Sei H_1, mit Wurzel w_1, der Hilfsbaum der v enthält und H_2, mit Wurzel w_2, der Hilfsbaum der v_c enthält. Es muss einen Pfad $P = (v_0, v_1, .., v_m)$ 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_v im rechten. Ein Widerspruch dazu, dass v_v der Vorgänger von v ist.
Es gibt also in jedem Fall einen Pfad von v zu w_2. w_2 kann bezogen auf T nur im linken Teilbaum von v enthalten und für alle in H_1 enthaltenen Schlüssel k_1 gilt entweder $k_1 > key\,(v) > key\,(v_v)$ oder $key\,(v) > key\,(v_v) > k_1$.

5.4 Laufzeitanalyse für access

Zunächst wird in zwei Lemmas 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 Baum T_{i-1}. Sei k die Anzahl der Knoten bei denen sich während der Ausführung von $access(x_i)$ eine Änderung des preferred child ergeben hat. Für die Laufzeit $access(x_i)$ gilt dann $O\left((k+1)\left(1+\log\left(\log\left(n\right)\right)\right)\right)$.*

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 Baum. 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 childs kommt es zu Kosten $O(\log_2(\log_2(n)))$ aufgrund einer *cut* und einer *join* Operation. Für das Suchen des Hilfsbaumes im bei der letzten Transformation zu T_i (Punkt 6 in der Beschreibung) entstehen auch wieder Kosten von $O(\log(\log(n)))$. Somit gilt auch für die Gesamtkosten $O((k+1)(1+\log(\log(n))))$.

□

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

Lemma 5.5. *Während der Ausführung von access(x_i) kommt es an genau $IB_i(X)$ Knoten zu einer Änderung des preferred child.*

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 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 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 Wurzel p weder zu einem Interleave noch zu einem Wechsel eines preferred child von links bzw. rechts zu rechts bzw. links führt. □

Satz 5.2. *Für die Laufzeit eines Tango Baum mit n Knoten, zum Ausführen einer Zugriffsfolge $X = x_1, x_2, .., x_m$ gilt $O((OPT(X) + n) + (1 + \log(\log(n))))$*

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

Mit $m \in \Omega(n)$ gilt dann auch $O(OPT(X)(1 + \log(\log(n))))$. Außerdem kann die angegebene obere Schranke nicht verbessert werden. Kommt es bei $access(x)$ zu $\Omega(\log(n))$ Wechsel bei preferred childs, muss der Hilfsbaum an

der Wurzel des Tango Baumes $\Omega(\log(n))$ mal durchlaufen werden. Somit hat der Tango Baum die balanced Eigenschaft aus Abschnitt 4.5 nicht. Somit kann er aufgrund der Implikationen auch die anderen Eigenschaften aus diesem Abschnitt nicht haben. Später werden zwei $\log(\log(n))$-competitve BST vorgestellt, welche das balanced property erfüllen.

5.5 Tango Baum konformes Vereinigen beim Rot-Schwarz-Baum

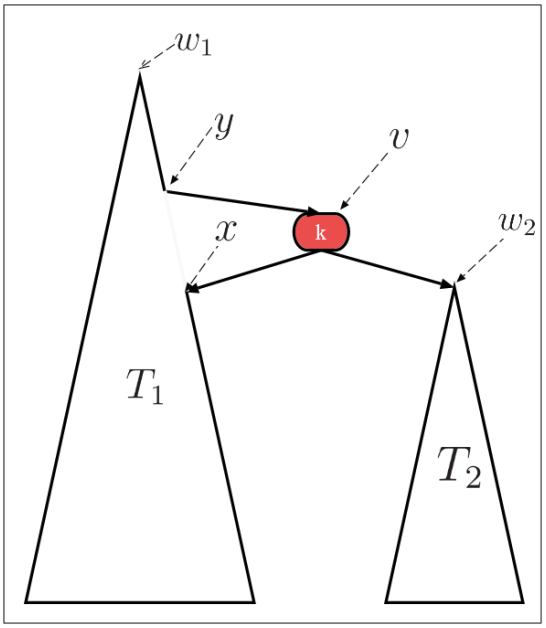


Abbildung 33: Beispielhaftes *concatenate* zweier RBT unterschiedlicher Schwarz-Höhe, nach Schritt 1.

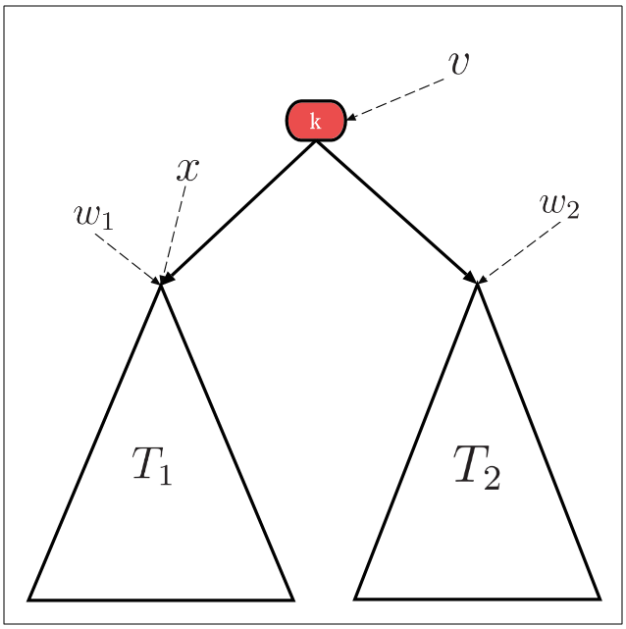


Abbildung 34: Beispielhaftes *concatenate* zweier RBT gleicher Schwarz-Höhe, nach Schritt 1

Hier wird die *concatenate(RBT* T_1*, key* k*, RBT* T_2*)* Operation, beim Rot-Schwarz-Baum so eingeführt, wie es ein Tango Baum von seiner Hilfsdatenstruktur verlangt. Bei den Ausführungen wird auf die Pflege der Attribute *depth*, *minDepth* und *maxDepth* verzichtet, damit sie nicht zu kleinteilig werden. Sei K_1 die Schlüsselmenge von T_1 und K_2 die von T_2. Die Operation gibt eine Referenz auf die Wurzel eines vereinigten RBT T mit 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 ein roter Knoten v mit Schlüssel k und Schwarz-Höhe 1 erzeugt. An diesem 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 geringeren Schwarz-Höhe wird dabei an den mit der höheren „seitlich angefügt“. Abbildung 33 zeigt dies beispielhaft. Sind die Schwarz-Höhen gleich wird wie in Abbildung 34 vorgegangen. 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 k, so angefügt, dass die Schwarz-Höhe jedes Knoten unverändert bleibt. Es sei w_1 die Wurzel von T_1. Es sei P ein Pfad $(r_0, r_1, ..., r_l)$ in T_1, so dass $r_0 = w_1$, r_l ein Blatt ist und $\forall i \in \{1, 2, ...l\}$: r_i ist rechtes 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 ein neuer roter Knoten v mit Schlüssel k und Schwarz-Höhe $bh(x)+1$ erzeugt. Als linkes Kind von v wird x gesetzt, als rechtes Kind w_2. Ist x die Wurzel in T_1, so ist v die Wurzel von T. Ansonsten ist x rechtes Kind eines Knoten y und das rechte Kind von y wird auf v gesetzt. Außerdem ist dann w_1 die Wurzel von T.
Fall 3: $bh(T_1) < bh(T_2)$
Dieser Fall ist fast symmetrisch zu Fall 2, jedoch kann der neue Knoten nicht zur Wurzel von T werden, da $bh(T_1) \neq bh(T_2)$. Es wird T_1 bei T_2, mit Hilfe von k angefügt. Es sei w_2 die Wurzel von T_2. Es sei P ein Pfad $(r_0, r_1, ..., r_l)$ in T_2, so dass $r_0 = w_2$, r_l ein Blatt ist und $\forall i \in \{1, 2, ...l\}$: r_i ist linkes Kind von r_{i-1}. P ist also der am weitesten links liegende Pfad von der Wurzel zu einem Blatt. Sei x der schwarze Knoten in P, mit $bh(x) = bh(w_1)$. x muss existieren denn $bh(w_2) < bh(w_1)$ und $bh(r_l) \leq bh(w_1)$, außerdem sind w_2 und r_l schwarz. Nun wird ein neuer roter Knoten v mit Schlüssel k und Schwarz-Höhe $bh(x) + 1$ erzeugt. Als rechtes Kind von v wird x gesetzt, als linkes Kind w_2. x ist linkes Kind eines Knoten y und das linke Kind von y wird auf v gesetzt. w_2 ist die Wurzel von T.

Resultat nach der Fallbehandlung Dass ein Baum mit Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ entstanden ist, erkennt man direkt an den Abbildungen 33 und 34. Aufgrund der Vorbedingung an die Parameter, muss T auch ein BST sein. Es müssen aber wieder die fünf Eigenschaften eines RBT 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. Anderenfalls 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 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, denn v ist rot. Bei keinem anderen Knoten hat sich bezüglich bzgl. der Schwarz-Höhe etwas geändert.

Wir sind also in der Situation dass nur entweder Eigenschaft zwei oder vier verletzt sein kann. Wenn Eigenschaft vier verletzt ist dann nur an 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 RBT zurückgegeben.

Laufzeit Sei n_1 die Anzahl der Knoten von T_1, sei n_2 die Anzahl der Knoten von T_2 und $n = n_1 + n_2$. Der Tango Baum fordert von seiner Hilfsdatenstruktur eine Laufzeit von $O(\log n)$ für die eben vorgestellte Operation. Das Ablaufen eines Pfades in T_1 oder T_2 zum Finden von x liegt in $O(\log(n))$. v erzeugen und in die Struktur einzubinden benötigt konstante Zeit und *insertFixup* benötigt $O(\log(n))$ Zeit.
$O(\log(n)) + O(1) + O(\log(n)) = O(\log(n))$
Die Vorgabe des Tango Baumes wird also eingehalten.
Für den nächsten Abschnitt wird noch eine genauere Betrachtung der Laufzeit benötigt. Es sei $d = |bh(T_1) - bh(T_2)|$. Die Suche nach x endet spätestens nach dem ein Pfad der Länge $2d + 1$ betrachtet wurde. Dabei steht die 1 für den Zugriff auf x selbst. Zu jedem schwarzen Knoten über x könnte noch ein roter kommen.
Nun wird noch auf die Anzahl der Iterationen innerhalb von *insertFixup* eingegangen. Sei v der Parameter von *insertFixup*. Sei w die Wurzel von T. v ist ein roter Knoten mit $bh(v) = bh(w) - d + 1$ und liegt in Ebene $2d + 2$ oder höher. 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 als Hilfsdatenstruktur für einen Tango Baum benötigt wird. Vorbedingung an die Parameter ist, dass k in der Schlüsselmenge K von T vorhanden ist. Zurückgegeben wird eine Referenz auf den Knoten v_k mit Schlüssel k. Linkes Kind von v_k ist die Wurzel eines RBT T_L mit Schlüsselmenge K_L, wobei gilt $K_L = \{i \mid i \in K \wedge i < k\}$. Rechtes Kind von v ist die Wurzel eines RBT T_2 mit Schlüsselmenge K_R, wobei gilt $K_R = \{i \mid i \in K \wedge i > k\}$. *split* gibt also in den meisten Fällen keinen RBT zurück. Die Operation setzt zunächst T_L auf den linken Teilbaum von v und T_R auf den rechten Teilbaum von v_k. Eventuell müssen die Wurzeln schwarz gefärbt werden. Sei $(v_0, v_1, .., v_m)$ der

Pfad von der Wurzel von T zu v_k. Es wird sich nun bei v_m startend Knoten für Knoten in dem Pfad nach oben gearbeitet. Ist der Schlüssel eines Knotens kleiner als k, so wird dieser Schlüssel und der linke Teilbaum des Knotens zu T_L hinzugefügt. Dies übernimmt unsere *vereinigen* Operation. Ist der Schlüssel größer als k, so wird dieser Schlüssel und der rechte Teilbaum des Knotens zu T_R hinzugefügt. Folgende Aufzählung beschreibt den Vorgang genauer.

1. Verwende die *search* Operation um den Knoten v_k mit Schlüssel k zu finden.

2. Setze den linken Teilbaum von v_k als T_L, den rechten als T_R. Löse beide Teilbäume aus T heraus.

3. Färbe die Wurzeln von T_L und T_R schwarz.

4. $\forall i \in \{1, 2, .., m\}$ absteigend sortiert. Ist der Schlüssel k_i von v_i kleiner als k, vereinige T_L mit dem aus T herausgelösten linken Teilbaum von v_i, mit k_i als dritten Parameter. Ansonsten vereinige T_R mit dem aus T herausgelösten rechten Teilbaum von v_i, mit k_i als dritten Parameter. Evtl. muss die Wurzel des herausgelösten Teilbaumes vor dem vereinigen schwarz gefärbt werden.

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 der beiden Teilbäume wird korrekt zugeordnet. Die Wurzel des anderen Teilbaumes von v_0 ist v_1. Alle Schlüssel die nicht im Teilbaum mit Wurzel v_1 liegen sind somit korrekt zugeordnet. Diese Betrachtung iteriert bis man auf v_k trifft. Die Schlüssel im Teilbaum mit Wurzel v_k werden korrekt zugeordnet.

Laufzeit Der Tango Baum fordert eine Laufzeit von $O(\log(n))$ von seiner Hilfsdatenstruktur für *split*, mit n ist die Anzahl der Knoten. Punkt 1 kostet $O(\log(n))$. 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 Gesamtlaufzeit ist. 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 $\{t_1, t_2, .., t_l\}$ die Menge der aus T herausgelösten Teilbäume die für die l Aufrufe verwendet wurden, wobei t_1 zum ersten Aufruf gehört, t_2 zum

zweiten usw.. Mit T_{Li} wird der Zustand von T_L bezeichnet nach dem t_i Parameter von *concatenate* war. T_{L0} steht für den Zustand vor dem ersten Aufruf.

Dass die zweite Ungleichung gilt, erkennt man direkt.

$$bh(T_{Li}) \leq bh(v_{i+1}) \leq bh(t_{i+1}) + 1 \tag{1}$$

Die Erste wird nun durch Induktion gezeigt:
T_{L0} war der linke Teilbaum von v_1, der Induktionsanfang ist somit gemacht.
$bh(T_{Li})$ mit $i > 0$ entsteht durch *concatenate*(T_{Li-1}, *key*(v_i), t_i).
Fall v_i ist rot:

$$\begin{aligned} & bh(t_i) < bh(v_i) \wedge bh(T_{Li-1}) \overset{IA}{\leq} bh(v_i) \\ \Rightarrow & bh(T_{Li}) \leq bh(v_i) \leq bh(v_{i+1}) \end{aligned}$$

Fall v_i ist schwarz:

$$\begin{aligned} & bh(t_i) \leq bh(v_i) \wedge bh(T_{Li-1}) \overset{IA}{\leq} bh(v_i) \\ \Rightarrow & bh(T_{Li}) \leq bh(v_i) + 1 = bh(v_{i+1}) \end{aligned}$$

Dadurch gilt

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

denn

$$\begin{aligned} & \sum_{i=1}^{l} |bh\left(t_i\right) - bh\left(T_{Li-1}\right)| \overset{1}{\leq} \sum_{i=1}^{l} \left(bh\left(t_i\right) - bh\left(T_{Li-1}\right) + 2\right) \\ \leq & \sum_{i=1}^{l} \left(bh\left(t_i\right) - bh\left(t_{i-1}\right) + 2\right) = bh\left(t_l\right) - bh\left(t_0\right) + 2 \cdot l = O\left(\log\left(n\right)\right) \end{aligned}$$

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

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

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

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

Die Kosten innerhalb einer Iteration sind konstant, damit ist $O(\log(n))$ eine ober 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 neben den Kosten für die Suche und *insertFixup* fallen pro Aufruf lediglich konstante Kosten an. 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.

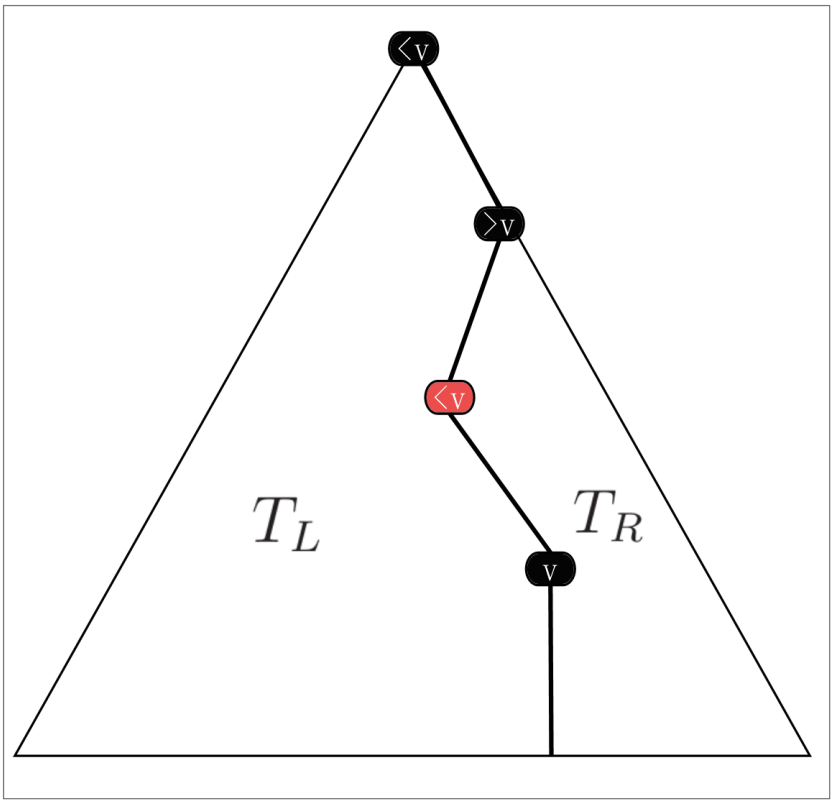


Abbildung 35: Beispielhaftes *split* eines RBT mit Parameter v

6 Splay Baum

Der in [1] vorgestellte Splay Baum ist ein online BST der ohne zusätzliche Hilfsdaten in seinen Knoten auskommt. Nach einer $access(k)$ Operation, ist der Knoten mit Schlüssel k die Wurzel, des Splaybaum. 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 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 Splaybaum und die Operation wird beendet.

2. v ist das rechte Kind der Wurzel (zag-Fall):
   Symmetrisch zu zig.

3. v ist ein linkes Kind und u ist ein linkes Kind. (zig-zig-Fall):
   Dieser Fall unterscheidet den Splaybaum vom einem anderen BST (move-to-root), mit schlechteren Laufzeiteigenschaften. Es wird zuerst eine Rechtsrotation auf u ausgeführt und erst danach eine Rechtsrotation auf v. Bei move-to-root ist es genau anders herum.

4. v ist ein rechtes Kind und u ist ein rechtes Kind. (zag-zag-Fall):
   Symmetrisch 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):
   Symmetrisch zu zig-zag.

Abbildung 36 zeigt drei der Fälle. Trotz der Einfachheit kann die Auswirkung einer einzelnen *splay* Operation sehr groß sein. Abbildung 37 aus [1] zeigt eine solche Konstellation.
Die Laufzeit von *access* auf einem BST mit n Knoten ist $O(n)$.

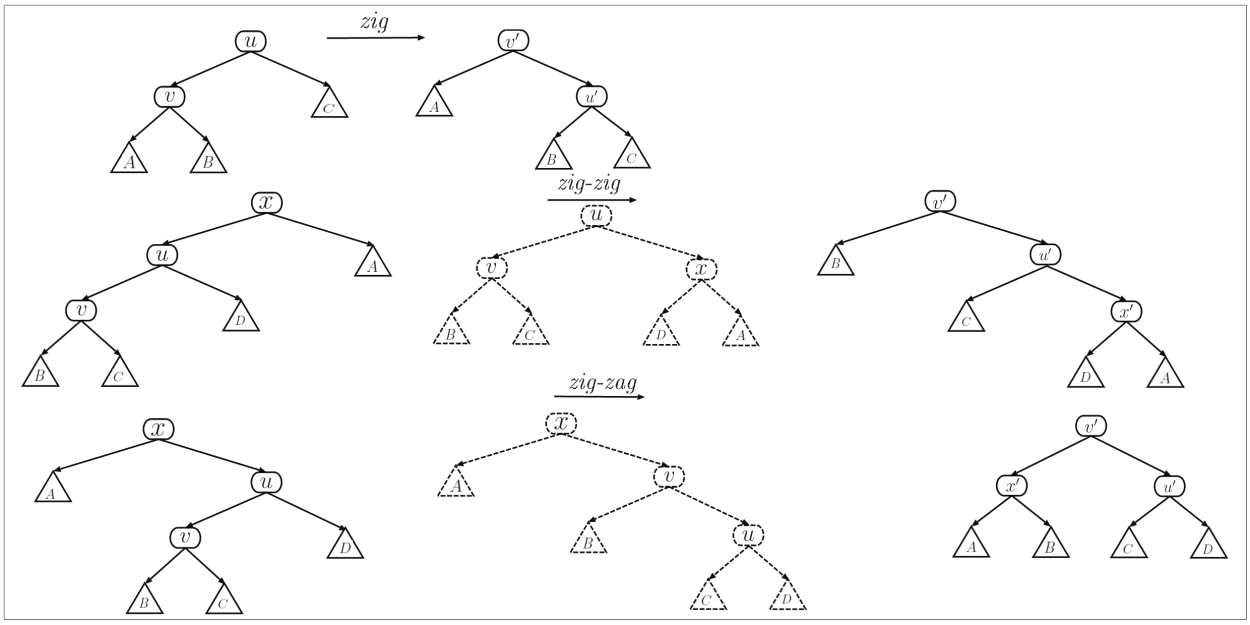


Abbildung 36: Darstellung von zig, zig-zag und zig-zig.

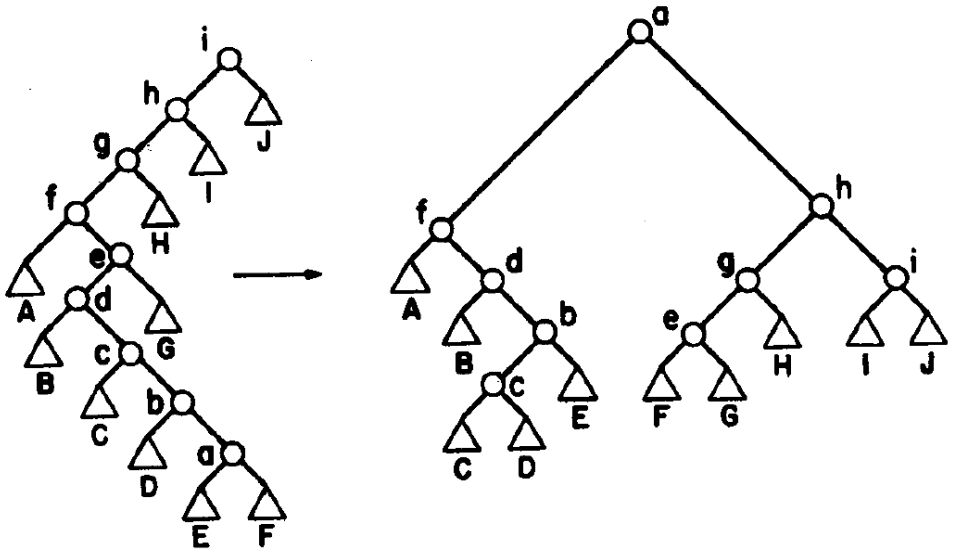


FIG. 4. Splaying at node *a*.

Abbildung 37: 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 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 Schlüssel k. Es werden den Knoten fest zugeordnete Gewichte angenommen. Die amortisierte Laufzeit von splay(k) ist maximal* $3(r(w) - r(v)) + 1 = O(\log(tw(w)/tw(v))) = O(\log(n))$

Beweis. Zunächst wird für *zig*, *zig-zag* und *zig-zig* gezeigt, dass die amortisierten Kosten nicht größer als $3\left(r(v)' - r(v)\right) + 1$ sind. Für die anderen drei Fälle folgt es dann aus der Symmetrie. Im Anschluss wird die gesamte Operation betrachtet. An Abbildung 36 ist zu erkennen, dass sich der Wert von $tw()$ 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 von Zustand des Splay Baum, da an ihr alle im Baum vorhandenen Gewichte aufsummiert werden. Deshalb muss $tw(v)' = tw(u)$ gelten. Daraus folgt $\Phi' - \Phi = r(u)' - r(v)$. Aus $r(v)' \geq r(u)'$ folgt $\Phi' - \Phi \leq r(v)' - r(v) \leq 3\left(r(v)' - r(v)\right)$. Addieren von 1 aufgrund der Rotation ergibt Kosten $\leq 3\left(r(v)' - r(v)\right) + 1$.

zig-zig Es müssen zwei Rotationen ausgeführt werden, deshalb entstehen amortisierte Kosten von

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

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

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

Dass die letzte Ungleichung gilt, kann man an einer Eigenschaft des $\log_2$ ableiten. Für $a, b \in R$ mit $a, b > 0$ und $a + b \leq 1$ gilt $\log_2(a) + \log_2(a) \leq -2$. An Abbildung 36 ist zu erkennen, dass sich $tw(v)$ vom Ausgangszustand zum Zwischenzustand hin nicht verändert. $tw(x')$ ist ebenfalls unverändert zum Zwischenschritt. Es kann also bei beiden Knoten mit den Werten aus dem Zwischenschritt gearbeitet werden. $tw(v') = tw(x') + tw(v) + w(u)$. Daraus folgt $((x') + (v))/tw(v') < 1$ und mit der Eigenschaft von $\log_2$ folgen die Ungleichungen.

zig-zag

$$\begin{aligned}
&2 + r(u)' + r(v)' + r(x)' - r(u) - r(v) - r(x) \quad \textit{mit } r(x) = r(v)' \textit{ und } r(v) \leq r(u)\\
\leq &2 + r(u)' + r(x)' - 2r(v)
\end{aligned}$$

Nun wird wie bei zig-zig vorgegangen.

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

Mit der $\log_2$ Regal aus zig-zig und Abbildung 36 folgt die Behauptung. Betrachtet man die Kosten der Gesamtoperation so bildet sich eine Teleskopsumme, die möglicherweise, wenn ein zig bzw. zag Fall enthalten ist, mit eins addiert werden muss. Daraus folgt das Lemma. □

6.3 Dynamische Optimalitäts Vermutung

Der Splay Baum erfüllt working set und dynamig finger aus Abschnitt 4.5 und auch noch einige weitere in dieser Arbeit nicht aufgeführte Eigenschaften. Für dynamic finger ist ein sehr aufwändiger Beweis in [9] enthalten. working set wurde bereits in [1] gezeigt. Dieser Beweis wird hier vorgestellt. Die anderen

Eigenschaften (außer dynamisch optimal) aus Kapitel 4.5 folgen dann aus diesen beiden. Aufgrund dieser oberen Schranken wurde in [1] 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.

working set 6.1. *Es sei T ein Splay Baum mit n Knoten. Sei $X = x_1, x_2, .., x_m$ eine für T erstellte Zugriffsfolge, mit $m \geq n$. Sei $w_i = |\{x_j | t_{xi} < j \leq i\}|$ definiert, wie in Kapitel 4.5. Dann gilt für die amortisierte Laufzeit* $O(n \log(n) + \sum_{i=1}^{m} \log(w_i))$.

Beweis. Den Knoten werden die Gewichte $1, 1/2^2, 1/2^3.., 1/2^n$ zugeordnet. Diesmal ist die Zuordnung nicht fest. Nach jeder *access* Operation können sich Gewichte ändern. Sei i der kleinste Index mit $x_i = key(v)$ für einen Knoten v. Sei $a = |\{x_l | l < i\}|$, dann wird v zum Start das Gewicht $1/2^{a+1}$ 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 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 $k^* \in \{1, 4, .., (k-1)^2\}$ direkt vor dem Zugriff x_j. Dann ist $1/(k^*+1)^2$ das Gewicht von u nach dem Zugriff 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 $1/{w_i}^2$ hat, somit gilt $tw(v_j) \geq 1/{w_i}^2$. Der Wert von $tw()$ für die Wurzel vom T ist $W = \sum_{i=1}^{n} 1/n^2 < 2 = O(1)$. Diese Werte in das Access Lemma eingesetzt, ergibt Kosten von $O\left(\log\left(1/\left(1/{w_i}^2\right)\right)\right) = O(\log({w_i}^2)) = O(log(w_i))$.
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.
Über die gesamte Zugriffsfolge kann das Potential nicht um mehr als $\sum_{i=1}^{n} \log_2(w(i)/W)$ 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 $O(n \log(n))$. □

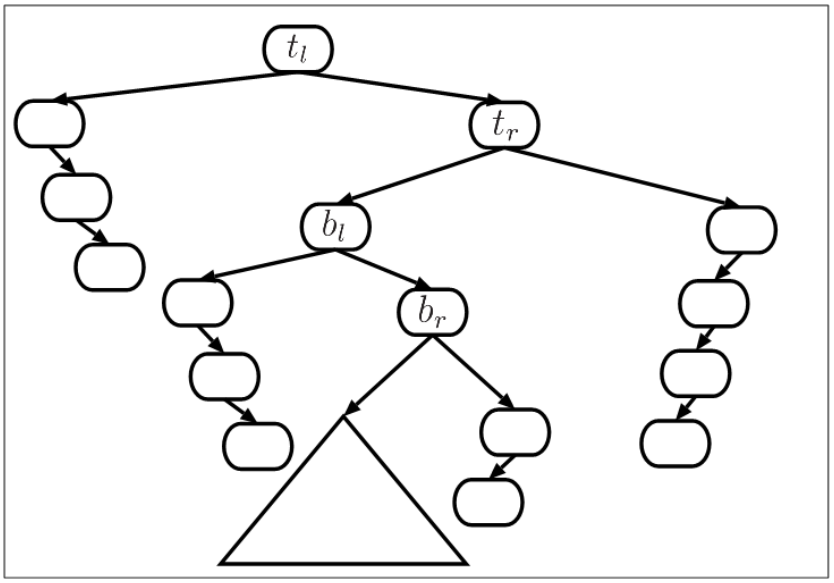
7 Weitere dynamische Suchbäume

Hier werden kurz zwei Variationen zum Tango Baum vorgestellt. Zum einen der Zipper Baum. Er wurde in [10] vorgestellt und 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 von T. Zum anderem den 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 vom Tango Baum. Ihre Repräsentation im eigentlichen BST T, macht den wesentlichen Unterschied zu einem Tango Baum aus. Abbildung 38 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 path $P_p = {p_1}^*, {p_2}^*, .., {p_m}^*$ in T stellt einen Hilfsbaum H dar, der in zwei Teile unterteilt wird, 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 müssen 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. zig Segmente entsprechen den längst möglichen Pfaden von Knoten mit rechten Kindern in P_p. zag Segmente entsprechen den längst möglichen Pfaden von Knoten mit linken Kindern in P_p. Das Blatt in P_p wird dem Segment seines Elternknoten zugeordnet. In Abbildung 39 sind zig und zag Segmente dargestellt.
Sei S_{zig} die Folge der in zig Segmenten enthalten Knoten, aufsteigend sortiert nach der Tiefe und S_{zag} die Folge der in zag Segmenten enthalten 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 $P_t = {p_1}^{**}, {p_2}^*, ..^*, {p_{n_t}}^*$ und z_b die Knoten $P_b = {p_{n_t+1}}^*, {p_{n_t+2}}^*, .., {p_{n_t+n_b}}^*$. Sei ${t_l}^*$ bzw. ${t_r}^*$ der Knoten in S_{zig} bzw. S_{zag} mit der kleinsten Tiefe. 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 Tree ist das linke Kind von t_r. Dass die Links-Rechts-Beziehung eingehalten wird ergibt aus den Aufbau der zig und zag Segmente. Es ist leicht zu erkennen, dass die Wurzel des bottom tree in konstanter Zeit erreicht werden kann.



2

Abbildung 38: Pfadrepräsentation beim Zipper Baum, basiert auf einer Abbildung aus [10].

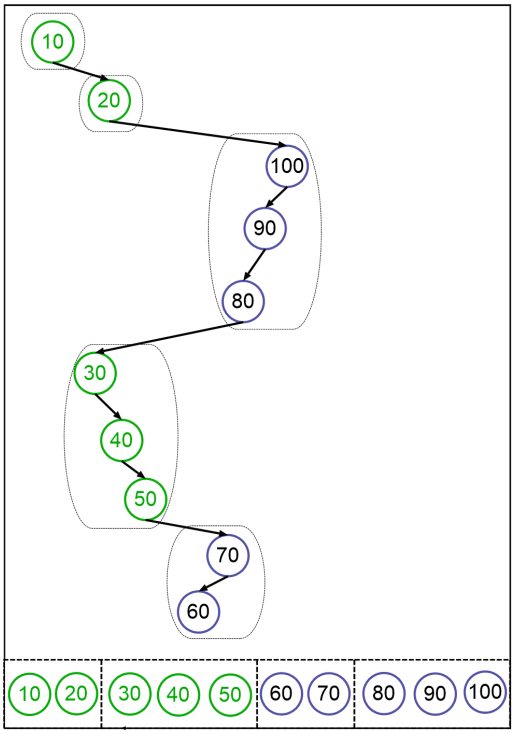


Abbildung 39: zig Segmente sind grün dargestellt. zag Segmente blau

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, mit l Knoten, soweit wie notwendig, in einen Pfad gewandelt, der $p_1, p_2, .., p_l$ 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. Ein Extraktionsprozess ist in Abbildung 40 dargestellt. l' ist der größte Schlüssel der kleiner ist, als die Schlüssel der zu extrahierenden Knoten. r' entsprechend der kleinste Schlüssel der größer ist, als diese Schlüssel.
Ein Knoten aus dem top zipper kann dem Pfad in konstanter Zeit hinzugefügt werden. Ist bei $access(k)$ der top zipper aufgebracht, sind bereits Kosten von $O(\log(\log(n)))$ entstanden. Sei P_p der preferred path den H repräsentiert. Ist der gesuchte Knoten im top zipper enthalten, entstehen in P aufgrund der Knoten von P_p asymptotisch betrachtet die gleichen Kosten, wie in H. Befindet sich der gesuchte Knoten im bottom zipper oder dem bottom tree entstehen in H ebenfalls Kosten von $O(\log(\log(n)))$. Deshalb entstehen asymptotisch betrachtet in H und innerhalb P_p die gleichen Kosten. Hieraus

wird abgeleitet, dass eine *access* Operation in $O(\log(n))$ Zeit ausgeführt werden kann.

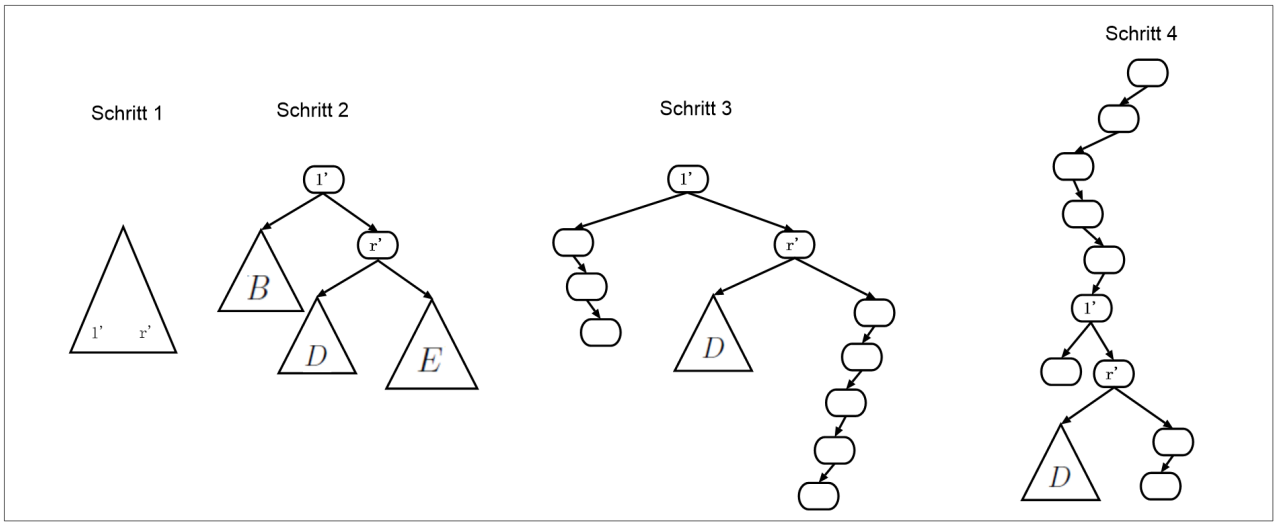


Abbildung 40: Extraktionsprozess beim Zipper Baum.

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 Baum ist, gibt es zusätzliche mögliche Zustände im Vergleich zu einem Tango Baum mit der gleichen Knotenzahl. Zu den genannten Eigenschaften bezüglich der Laufzeit sind Beweise in [11] enthalten. Der Multisplay Baum erfüllt die working set Eigenschaft [12].

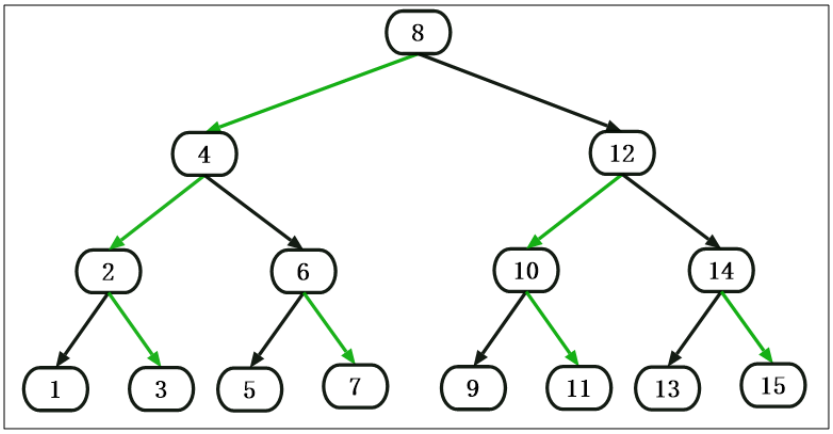


Abbildung 41: Referenzbaum mit grün gezeichneten preferred paths

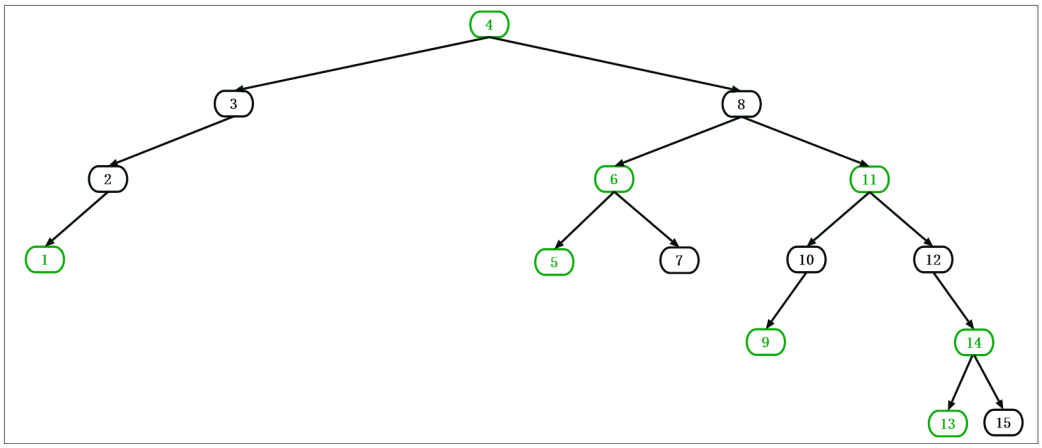


Abbildung 42: Beispielhafter Multisplay Baum zu Abbildung 41.

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 von 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.

8 Implementierung und Laufzeittests

In diesem Kapitel wird kurz die Implementierung zum Tango Baum beschrieben und dann werden noch die Laufzeittests dargestellt.

8.1 Implementierung

Implementiert wurde ein Tango Baum, ein Rot Schwarz Baum in der Rolle als Hilfsdatenstruktur für den Tango Baum. Außerdem wurde die *access* Operation des Splay Baum implementiert, um Laufzeittest 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 47 stellt ein Klassendiagramm dar.

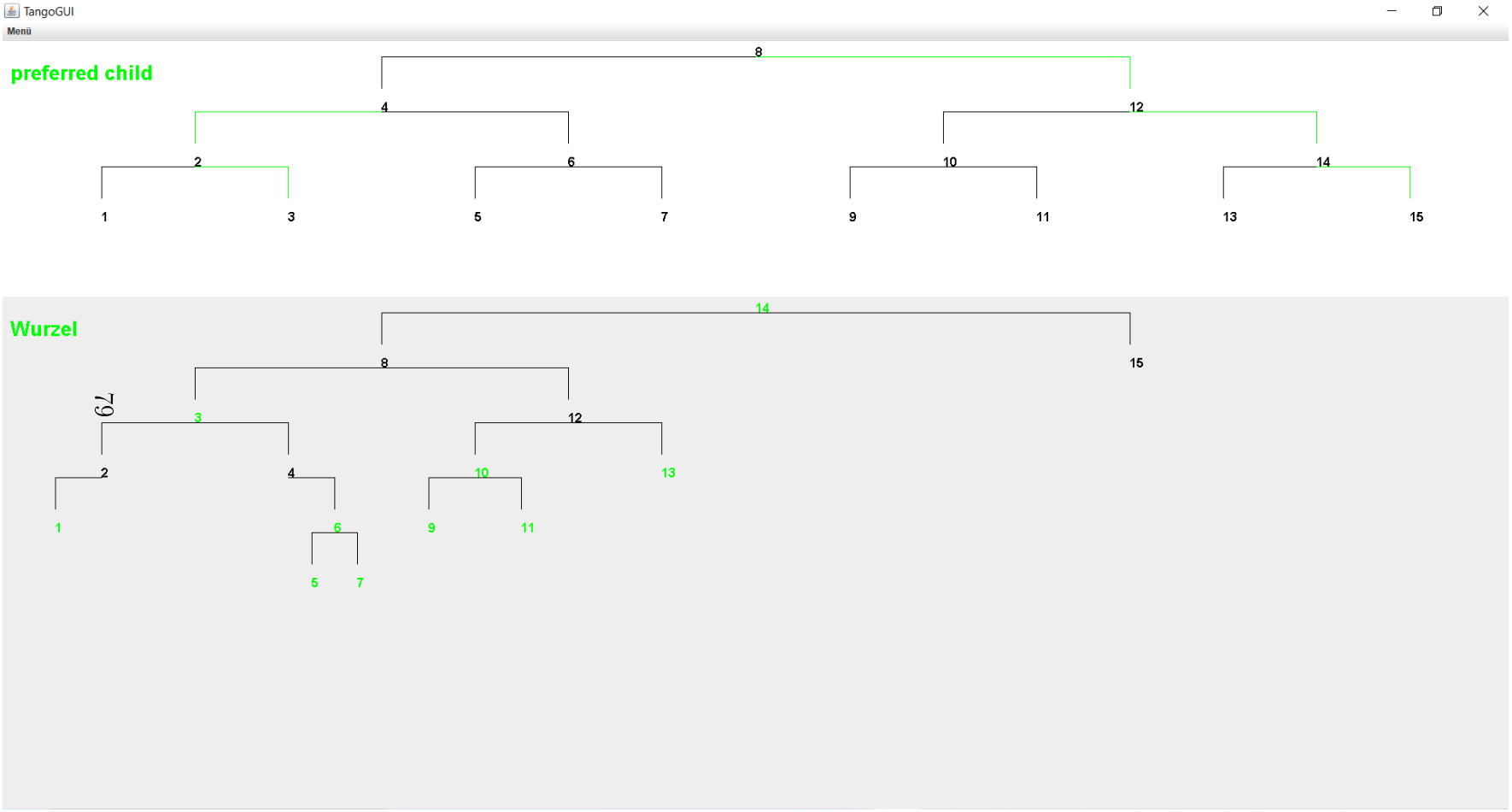


Abbildung 43: Oberfläche zum Tango Baum

Abbildung 43 zeigt das Hauptfenster. Oben ist ein Referenzbaum zu einem Tango Baum mit 15 Knoten dargestellt, unten der Tango Baum. Preferred Childs und die Wurzeln von Hilfsbäumen sind grün dargestellt.

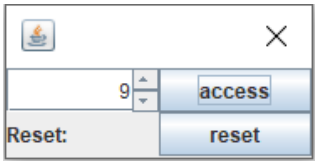


Abbildung 44: *access* Operationen anstoßen.

Mit dem Menüpunkt „access“ wird das Fenster aus Abbildung 44 geöffnet. Mit diesem werden *access* Operationen angestoßen. Außerdem können die Bäume damit zurückgesetzt werden.

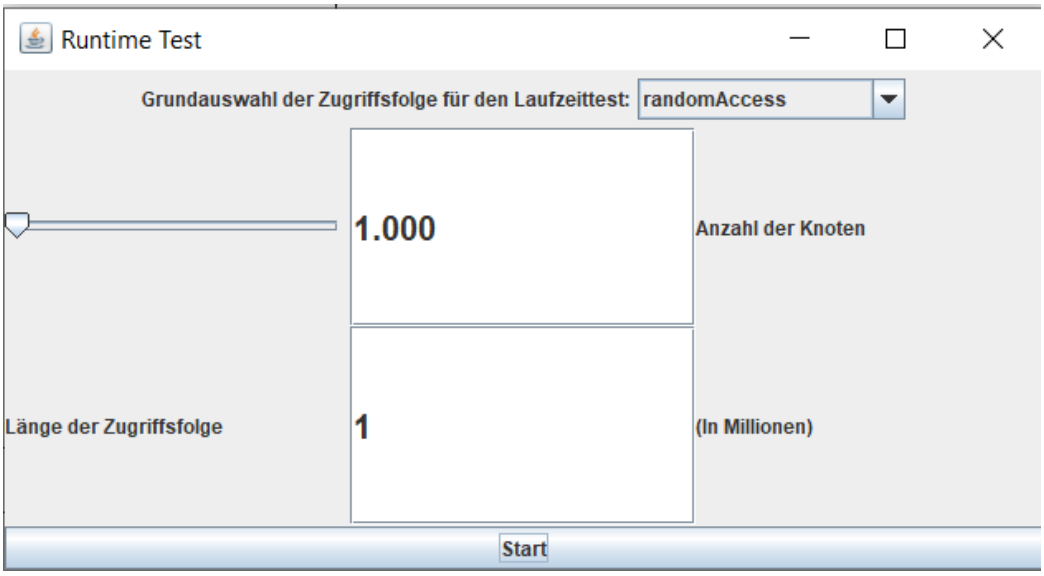


Abbildung 45: Laufzeittest anstoßen.

Mit dem Menüpunkt „RuntimeTest“ wird das Fenster aus Abbildung 45 geöffnet. Mit diesem werden Laufzeittests zwischen dem Tango Baum und dem Splay Baum angestoßen. Auf die Parameter und den Aufbau der Zugriffsfolgen, wird im Abschnitt zu den Laufzeittests eingegangen.

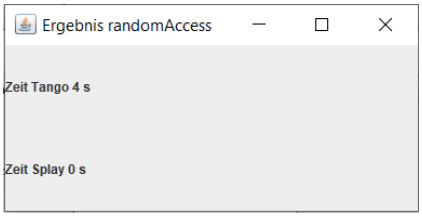


Abbildung 46: Ergebnisanzeige eines Laufzeittests.

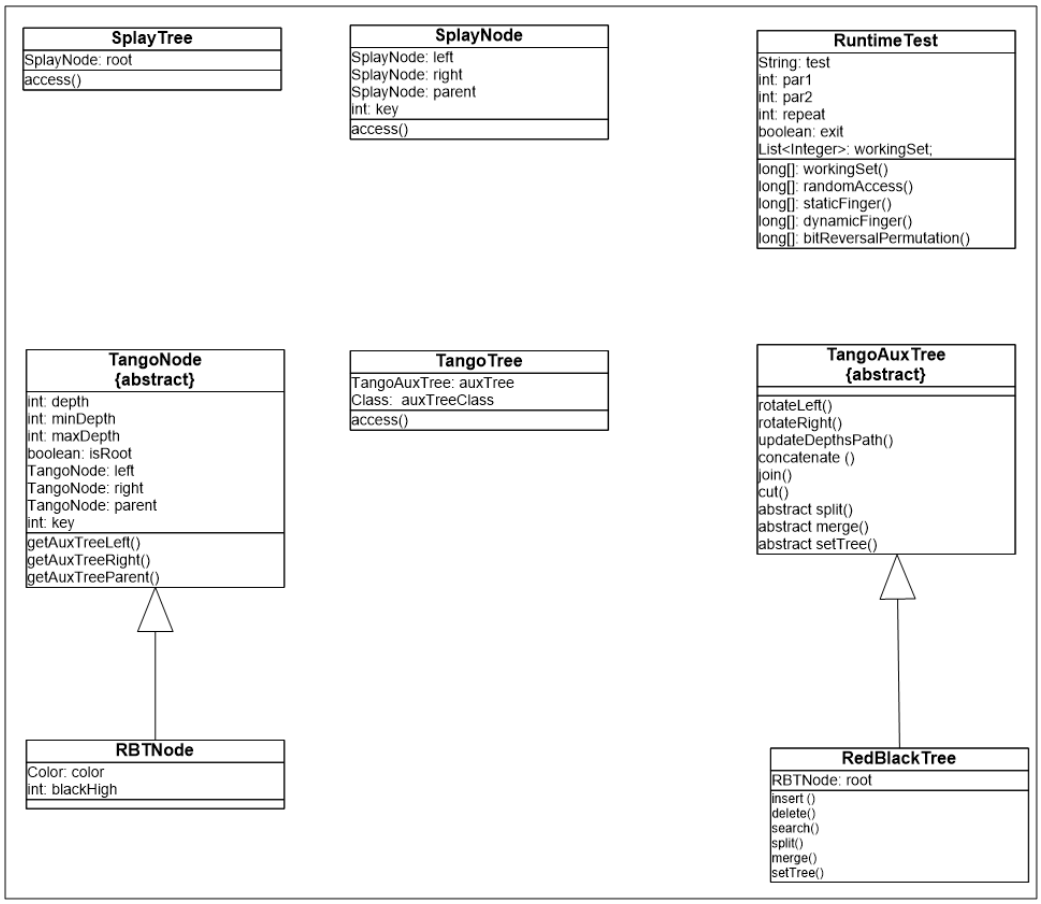


Abbildung 47: Wesentliche Klassen der Implementierung. Methoden zum direkten lesen bzw. schreiben von Attributen sind nicht dargestellt.

8.1.1 Beschreibung der Klassen

SplayTree und SplayNode Der Splay Baum startet genau wie der Tango Baum perfekt balanciert, auch wenn sich dies bei längeren Zugriffsfolgen

praktisch nicht auswirken sollte. Ansonsten gibt es keine Besonderheiten. *access* verhält sich genau wie im Kapitel zum Splay Baum beschrieben.

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

TangoAuxTree Klassen die als Hilfsdatenstruktur im Tangobaum eingesetzt werden sollen, müssen diese Klasse erweitern. *setTree* wird benötigt, da die Klasse TangoTree die BST Struktur nur über das Attribut „auxTree“ erreicht. Gibt es eine Veränderung an der Wurzel des Tango Baum, wird die BST Struktur von „auxTree“ neu gesetzt. *updateDepthsPath* pflegt die Attribute „minDepth“ und „minDepth“ der TangoNode.

TangoTree „auxTree“ macht die Wurzel des Tango Baum erreichbar. Außerdem können über diesen Attribut die *split* und *join* Operationen aufgerufen werden. „auxTreeClass“ entspricht der Klasse der eingesetzten Hilfsbäumen. Diese wird dem Constructor übergeben. Somit kann der RBT einfach durch eine andere geeignete Struktur ersetzt werden.

RedBlackTree und RBTNode Erweitern die abstrakten Klassen. RedBlackTree verhält sich wie im Kapitel zu RBT beschrieben.

RuntimeTest Hier ist die Durchführung der Laufzeittests umgesetzt. Mit „exit“ kann ein Test abgebrochen werden. Ein von dieser Klasse erzeugtes Objekt, führt genau einen Laufzeittest durch, die restlichen Attribute dienen dessen Parametrierung. Die Methoden für die Test geben Arrays der Länge 2 zurück. Der erste Wert entspricht der Laufzeit des Tango Baum, der zweite der des Splay Baum. Um Programmabbrüchen aufgrund zu wenig Speicher vorzubeugen, wurde bei den Projekteigenschaften, die Option „ -Xmx4096m“ gesetzt, Abbildung 48. Ein Datenblatt zu dem verwendeten System ist dem Kapitel angefügt.

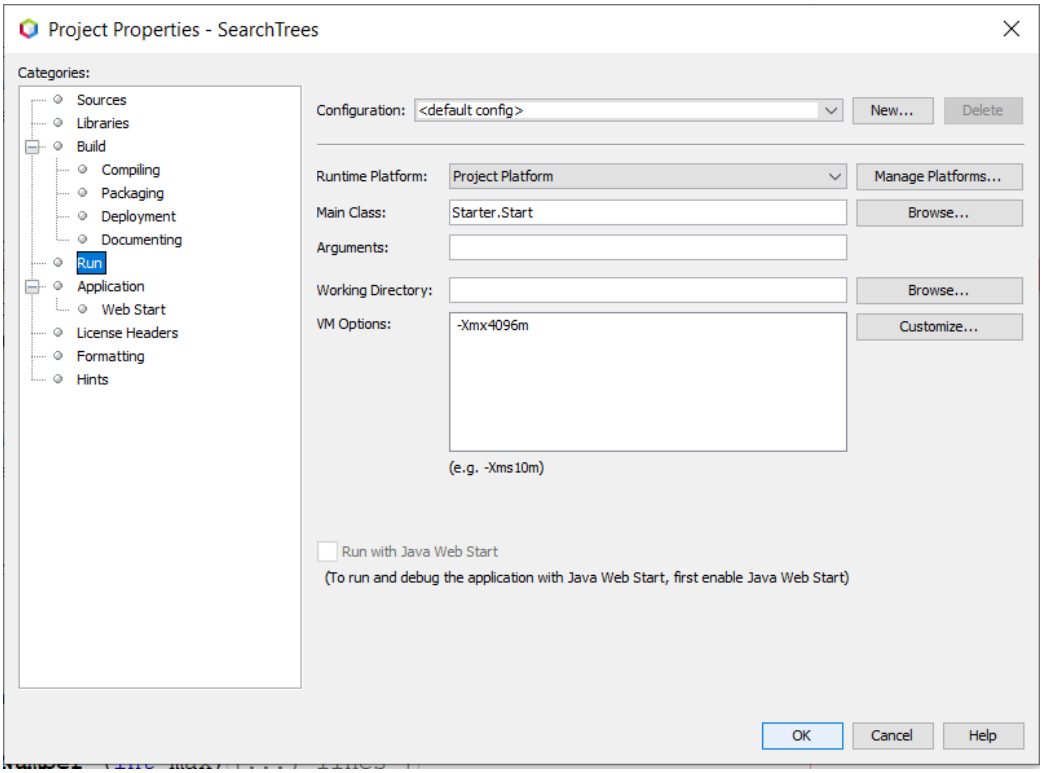


Abbildung 48: Zur Ausführung verwendbaren Speicher erweitert.

8.2 Laufzeittests zwischen Tango Baum und Splay Baum

Es werden Tests zu fünf Arten von Zugriffsfolgen durchgeführt. Zunächst wird immer der Aufbau der Zugriffsfolge beschrieben und dann die Ergebnisse präsentiert. n entspricht der Anzahl der Knoten, m der Länge der Zugriffsfolge. Die Schlüsselmengen haben immer die Form $\{1, 2, .., n\}$. Bei jeder Testart wurde zu jeder Knotenzahl, beim Tango Baum und Splay Baum immer exakt die gleiche Zugriffsfolge verwendet. m ist die Länge der Zugrissfolge. Gibt es keine Abhängigkeit von m zu n, so wurde für m immer 40 Millionen verwendet. Der Splay Baum wird durchweg die kürzeren Zeiten liefern. Eine mögliche Erklärung für die vergleichsweise oft hohen Zeiten des Tango Baum ist, dass dieser erst bei noch größeren Instanzen einen Vorteil, aus der Unterteilung in Hilfsbäume ziehen kann.

8.2.1 Zufällige Zugriffsfolge

Die Zugriffsfolgen werden von einem Pseudozufallsgenerator erzeugt.

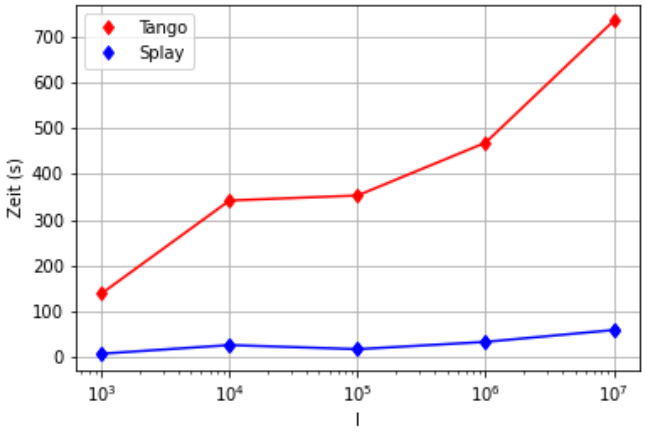


Abbildung 49: Laufzeittest mit zufälliger Zugriffsfolge.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	138	6
10^4	342	16
10^5	353	16
10^6	469	32
10^7	736	58

Tabelle 2: Laufzeittest mit zufälliger Zugriffsfolge

8.2.2 Bit reversal permutation

Auf der X-Achse wird die Länge der einzelnen Binärdarstellungen l dargestellt. In der Spalte 2^l kann die Länge der Zugriffsfolge und die Anzahl der Knoten abgelesen werden. Die Ergebnisse bestätigen, dass es sich um eine aufwändige Zugriffsfolge für BST handelt.

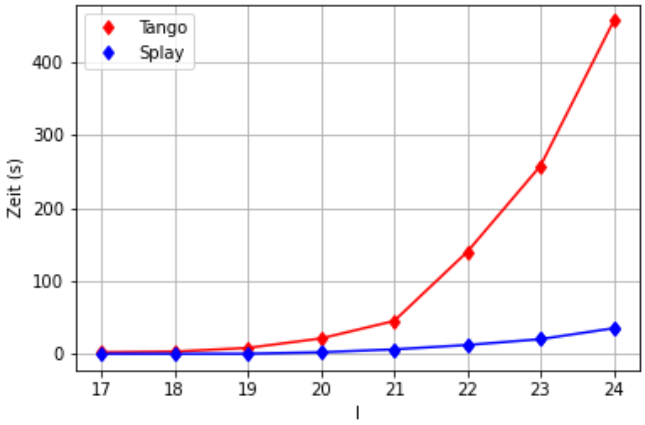


Abbildung 50: Laufzeittest bit reversal permutation.

l	2^l	Zeit Tango in (s)	Zeit Splay in (s)
17	131.072	2	1
18	262.144	3	1
19	524.288	8	1
20	1.048.576	21	2
21	2.097.152	45	6
22	4.194.304	140	12
23	8.388.608	258	20
24	16.777.216	457	35

Tabelle 3: Laufzeittest bit reversal permutation

8.2.3 Static Finger

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 in der Zugriffsfolge geschieht wieder über einen Pseudozufallsgenerator.

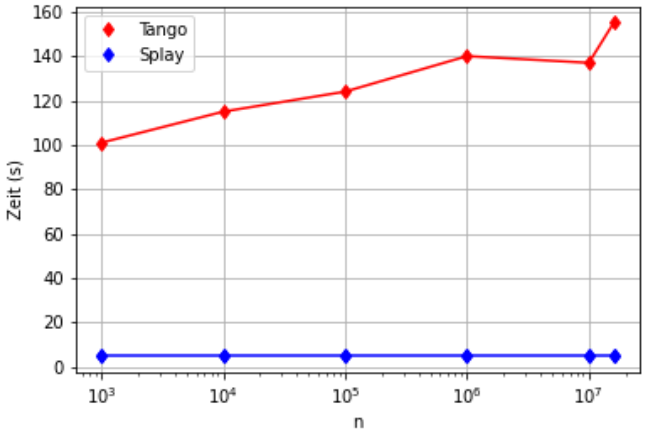


Abbildung 51: Laufzeittest static finger.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	101	5
10^4	115	5
10^5	124	5
10^6	140	5
10^7	137	5
$1,6 * 10^7$	155	5

Tabelle 4: Laufzeittest static finger

8.2.4 Dynamic Finger

Beim ersten Test wird Zugriffsfolge $1, 3, 5, .., n-1, 1, 3, 5, .., n-1, ...$ verwendet. Der Abstand a zwischen zwei aufeinanderfolgenden Schlüssel ist also 2.

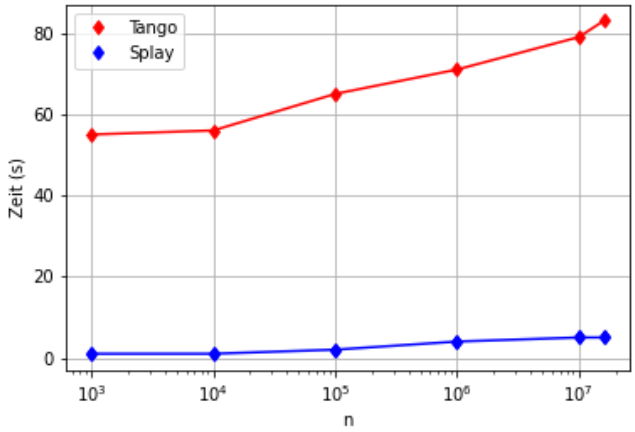


Abbildung 52: Laufzeittest dynamic finger.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	55	1
10^4	56	1
10^5	65	2
10^6	71	4
10^7	79	5
$1,6 * 10^7$	83	5

Tabelle 5: Laufzeittest dynamic finger

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

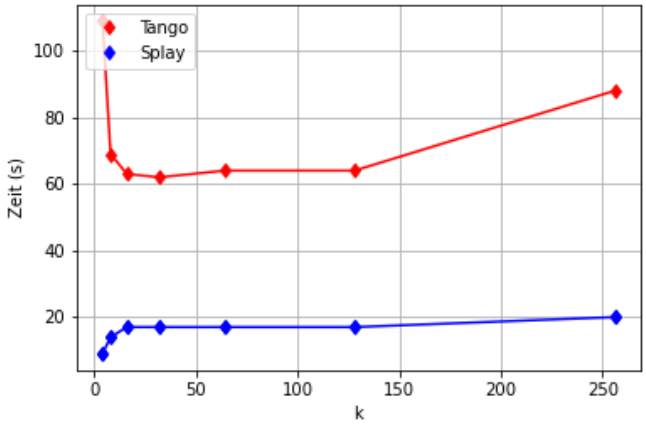


Abbildung 53: Laufzeittest dynamic finger, mit unterschiedlichen a.

a	Zeit Tango in (s)	Zeit Splay in (s)
4	109	9
8	69	14
16	63	17
32	62	17
64	64	17
128	64	17
256	88	20

Tabelle 6: Laufzeittest dynamic finger, mit unterschiedlichen a

8.2.5 Working Set

Das working set enthält 5 Prozent der Schlüssel des BST. Diese wurden gleichmäßig über die enthaltenen Schlüssel verteilt. Wird z.B. jeder dritte Schlüssel benötigt wird das working set also mit $1, 4, 7$ usw. aufgebaut. Die Zugriffsfolge besteht nur aus Schlüsseln des working set. Die Auswahl daraus geschieht dann für jede *access* Operation wieder zufällig.

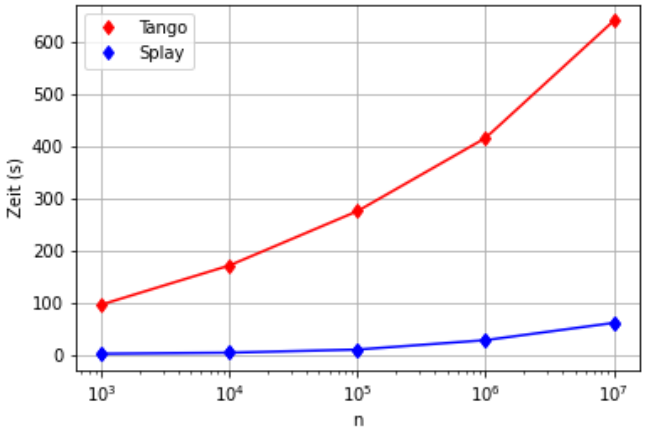


Abbildung 54: Laufzeittest working set.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	97	3
10^4	172	5
10^5	276	11
10^6	416	29
10^7	640	62

Tabelle 7: Laufzeittest working set

Hier ist $n = 1.000.000$ fest und der Prozentwert p für die Anzahl der Schlüssel im working set veränderlich.

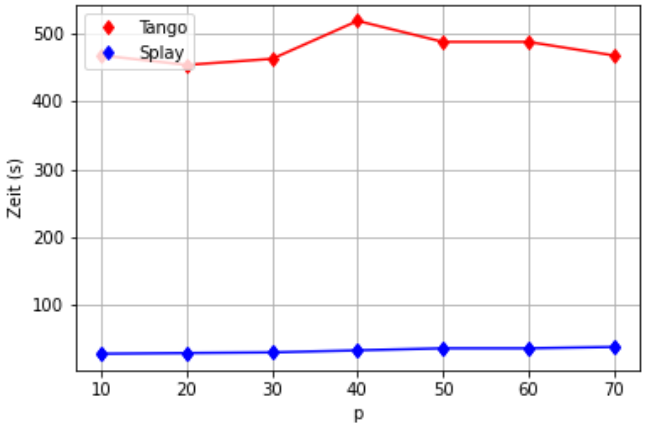


Abbildung 55: Laufzeittest working set, mit unterschiedlichem p .

p	Zeit Tango in (s)	Zeit Splay in (s)
10	468	28
20	454	29
30	463	30
40	519	33
50	488	36
60	488	36
70	468	38

Tabelle 8: Laufzeittest working set

8.2.6 Weitere Laufzeittests

Alternierend die Schlüssel 1 und n:

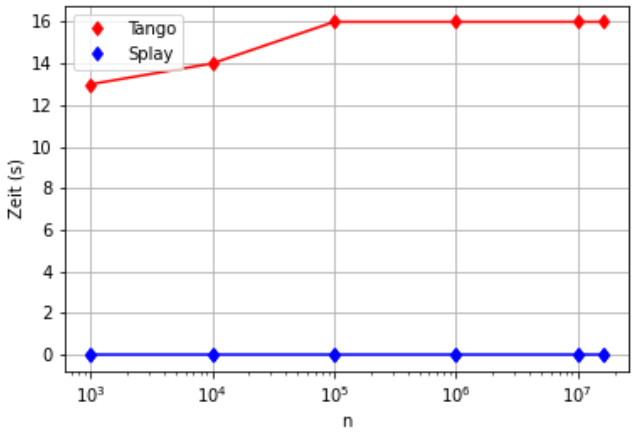


Abbildung 56: Laufzeittest alternierend.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	13	1
10^4	14	1
10^5	16	1
10^6	16	1
10^7	16	1
$1,6 * 10^7$	16	1

Tabelle 9: Laufzeittest alternierend

Aufsteigend sortiert, mit insgesamt n Zugriffen:

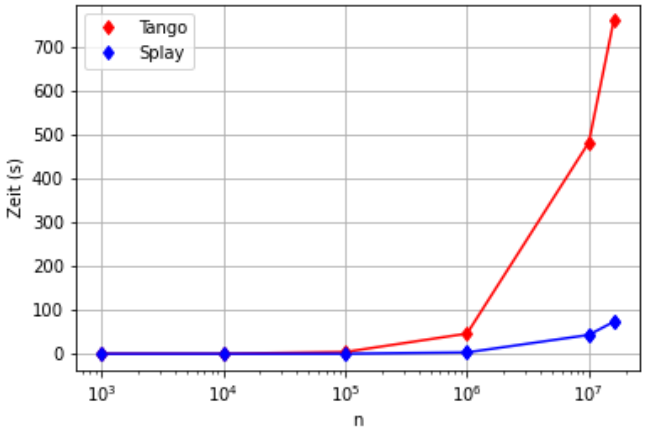


Abbildung 57: Laufzeittest aufsteigend sortiert.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	1	1
10^4	1	1
10^5	4	1
10^6	46	3
10^7	483	43
$1,6 * 10^7$	761	73

Tabelle 10: Laufzeittest aufsteigend sortiert

Absteigend sortiert, mit insgesamt n Zugriffen:

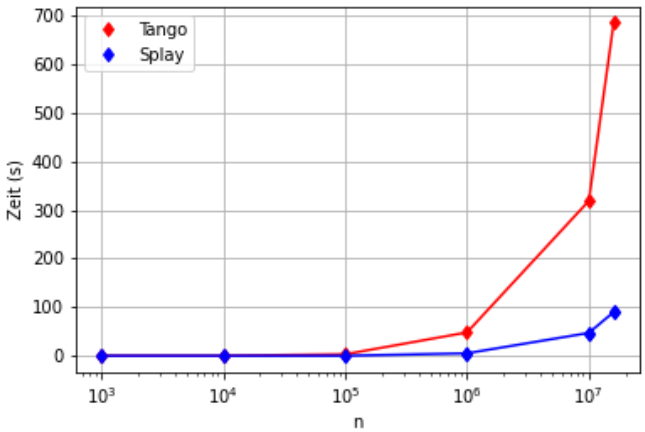


Abbildung 58: Laufzeittest absteigend sortiert.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	1	1
10^4	1	1
10^5	3	1
10^6	48	5
10^7	319	47
$1,6 * 10^7$	686	90

Tabelle 11: Laufzeittest absteigend sortiert

Aufsteigend und absteigend sortiert, geschachtelt. Die Zugriffsfolge ist $1, n, 2, n-1, ..., n-2, 2, n-1, 1$

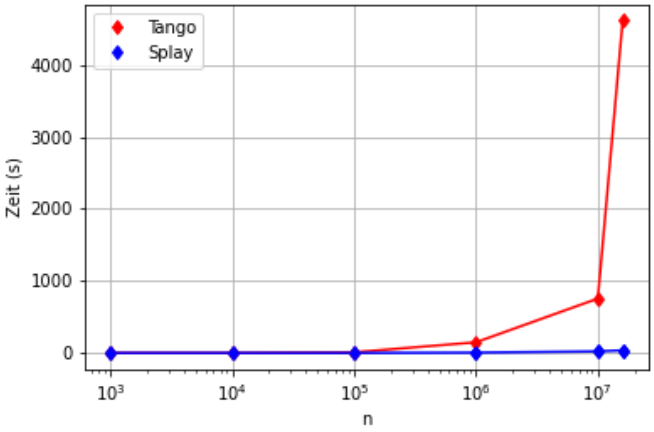


Abbildung 59: Laufzeittest auf- bzw. absteigend geschachtelt.

n	Zeit Tango in (s)	Zeit Splay in (s)
10^3	1	1
10^4	1	1
10^5	9	1
10^6	147	4
10^7	754	21
$1,6 * 10^7$	4611	36

Tabelle 12: Laufzeittest auf- bzw. absteigend geschachtelt.

Anhang: Daten des Systems zu den Laufzeittests

Hardware

Gerätenummer	8L444EA
Produktname	HP Pavilion x360 - 14-dh1133ng
Mikroprozessor	Intel® Core™ i5-10210U (1,6 GHz Basisfrequenz, bis zu 4,2 GHz mit Intel® Turbo Boost-Technologie, 6 MB Cache, 4 Kerne)
Chipsatz	Intel® Integrated SoC
Arbeitsspeicher, standard	DDR4-2666 SDRAM mit 16 GB (1 x 16 GB)
Festplatte	512 GB Intel® SSD
Optisches Laufwerk	Optisches Laufwerk nicht im Lieferumfang enthalten
Bildschirm	Nahezu randloses FHD-IPS-Display, Micro-Edge-Design, multitouchfähig, 35,6 cm (14 Zoll) Diagonale, WLED-Hintergrundbeleuchtung (1920 x 1080)
Drahtlos-Technologie	Realtek RTL8821CE 802.11b/g/n/ac (1x1) und Bluetooth® 4.2, kombiniert
Erweiterungsplätze	1 SD-Kartenleser für Medien verschiedener Formate
Externe Anschlüsse	1 USB 3.1 Gen 1 Type-C™ (nur Datenübertragung, 5 Gbit/s Signalrate); 2 USB 3.1 Gen 1 Type-A (nur Datenübertragung); 1 AC Smart-Pin; 1 HDMI 1.4; 1 Kopfhörer/Mikrofon kombiniert
Mindestabmessungen (B x T x H)	32,4 x 22,29 x 2,05 cm
Gewicht	1,58 kg
Netzteil-Typ	Smart-Netzteil mit 65 W
Batterie-Typ	Li-Ion-Akku, 3 Zellen, 41 Wh
Akkunutzungsdauer bei gemischter Nutzung	Bis zu 13 Stunden und 15 Minuten
Akkunutzungsdauer bei Videowiedergabe	Bis zu 11 Stunden und 15 Minuten
Kamera	HP Wide Vision HD-Kamera mit integriertem Dual-Array-Digitalmikrofon
Audio-Merkmale	B&O; Dual Lautsprecher; HP Audio Boost 1.0

Software

Betriebssystem	Windows 10 Home 64
HP apps	Alexa; HP 3D DriveGuard; HP Audio Switch; HP CoolSense; HP JumpStart; HP Support Assistant
Mitgelieferte Software	McAfee LiveSafe™
Vorinstallierte Software	Netflix (30-tägiges kostenloses Probeangebot)
Software – Produktivität und Finanzierung	Testversion für Kunden des neuen Microsoft Office 365 für 1 Monat

9 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. Bei ihm wurde die Idee mit den preferred child jedoch als Erstes umgesetzt, und hier könnten 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 BST. Außerdem gibt es eine interessante geometrische Sicht auf BST, von der Aussagen über BST und andere Verfahren abgeleitet werden konnten. Es gibt auch deutlich mehr obere Laufzeitschranken, für Zugriffsfolgen mit speziellen Eigenschaften, als sie hier vorgestellt wurden.
Ob es irgendwann gelingen wird die dynamische Optimalität, des Splay Baum oder irgendeiner anderen Variante eines BST zu beweisen, ist offen. Für den Einsatz bei gewöhnlich großen Schlüsselmengen, müssen dann aber auch die Summanden und Faktoren berücksichtigt werden, die in der O Notation vernachlässigt werden.

Erklärung

Name : Andreas Windorfer
Martikelnummer: q8633657
Thema: Tango Baum

Ich erkläre, dass ich die Bachelorarbeit 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 Bachelorarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Bachelorarbeit 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] Prof Erik Demaine. 6.851: Advanced data structures spring 2010.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

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

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

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

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

[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 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] Chengwen Chris Wang. *Multi-Splay Trees*. PhD thesis, USA, 2006.