Bachelorarbeit

Andreas Windorfer

27. Juni 2020

Inhaltsverzeichnis

1 Tango Baum

Der Tango Baum ist ein aus BSTs, den **Hilfsbäumen**, bestehender BST. Auf die Anforderungen an die Hilfsbäume wird in Abschnitt 1.2 eingegangen und mit dem Rot-Schwarz-Baum wird eine mögliche Variante noch detailliert vorgestellt. Der Tango Baum wurde in [1],von Demaine, Harmon, Iacono und Patrascu beschrieben, inklusive eines Beweises über seine *lg lg n*-competitiveness. Ebenfalls in [1] 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.

1.1 Interleave Lower Bound

Sei X 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 **??**. Hier ist der lower bound tree Y zu einer Zugriffsfolge X, der komplette BST mit Schlüsselmenge K. Anders als in Abschnitt **??** gibt es hier somit zu jeder Zugriffssequenz nur genau einen lower bound tree. Abbildung 1 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 $X = x_1, x_2, ., x_m$ die Zugriffssequenz und $X_l^r = x_{1'}, x_{'2}, .., x_{jm'}$ wie in Abschnitt **??** 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(v)* die Funktion die zu dem Knoten $v \in U$ die Anzahl der Interleaves durch v zurückgibt. Die Funktion $IB(X)$ ist definiert durch:

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

Sei T_0 der BST mit Schlüsselmenge K auf 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.

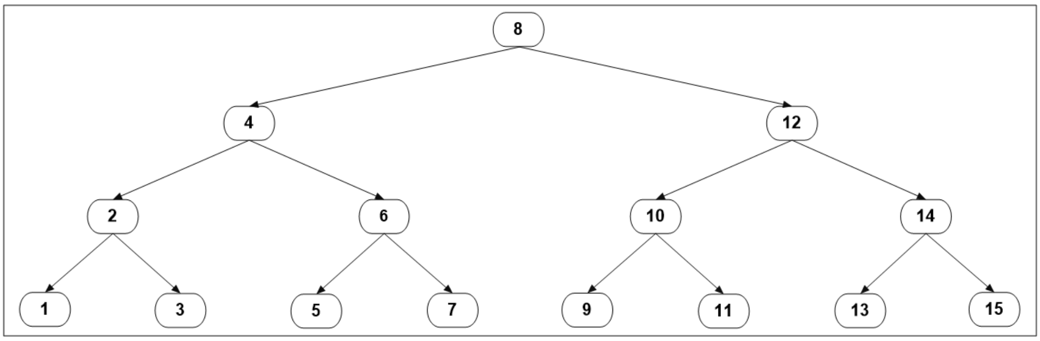

Abbildung 1: Der lower bound tree zur Zugriffsfolge $1, 2, .., 15$

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 1.1. *Sei $X = x_1, x_2, .., x_m$ eine Zugriffssequenz und Y ein zu X erstellter lower bound tree mit Schlüselmenge 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. 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. 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 **??**. 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 1.1 definiert.

Lemma 1.1. *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 in T_l.*

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 in 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 $v_{li} \neq v_l$ 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 1.1. *Sei $u_1, u_2 \in U$, mit $u_1 \neq u_2$. Sei v ein Knoten in T_j. v kann nicht sowohl der transition point von u_1, als auch der von u_2 sein.*

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, 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, ohne Verlust der Allgemeinheit, u_1 ein Vorfahre von u_2. werden drei Fälle unterschieden:

1. Ist $mathitkey\left(v_1\right)$ ist nicht im Teilbaum mit Wurzel u_2 enthalten, so kann v_1 nicht der transition point von u_2 sein.

2. $key\left(v_1\right)$ ist im Teilbaum mit Wurzel u_2 enthalten und $key\left(v_1\right)$ 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.$key\left(v_1\right)$ 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_1 angeben der zwei Knoten aus der linken Region von u_1 enthält, dass ist jedoch ein Widerspruch dazu, dass $key\left(v_1\right)$ in der linken Region von u_1 enthalten ist und v_1 zudem der transition point für u_1 ist.
   v_2 ist entweder der Knoten w_l oder w_r je nachdem wessen Tiefe größer ist, somit gilt $v_1 \neq v_2$.

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

□

Satz 1.1. *Für eine Zugriffssequenz $X = x_1, x_2, .., x_m$ und n die Anzahl der Knoten im zu X erstellten lower bound tree. Dann gilt*
$OPT\left(X\right) \geq IB\left(X\right)/2 - n$.

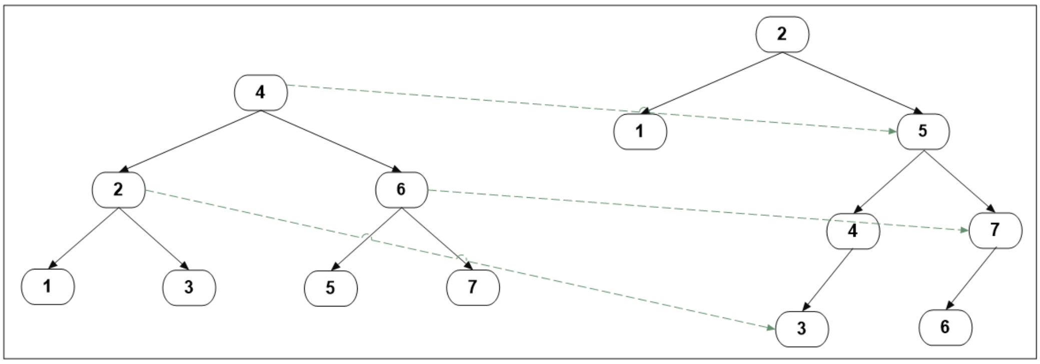

Abbildung 2: Transition point Zuordnung. Links ein lower bound tree, rechts ein möglicher T_j.

Beweis. Es wird die Mindestanzahl der Berührungen von transition points gezählt. Durch 1.1 kann die Anzahl der Berührungen für jedes $y \in P$ einzeln bestimmt werden, diese müssen dann lediglich noch aufaddiert werden. Sei l, r, v_r und v_l wie in Lemma1 zu y definiert, so dass entweder l oder r der transition point zu y sein muss, je nachdem welcher der beiden Knoten die größere Tiefe hat (**??**). Sei $X_l^{r}{}' = x_{i_1}, x_{i_2}, .., x_{i_p}$ 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}. Nun wird angenommen, dass die x_{j_i} mit i ist gerade in der rechten Region von y liegen, und die x_{j_i} mit i ist ungerade in der linken Region. Der andere Fall kann wieder direkt abgeleitet werden. Sei $q \in \mathbb{N}$ mit $1 \geq q \geq \lfloor p/2 \rfloor$. $access\left(x_{i_{2q-1}}\right)$ muss v_l berühren und $access\left(x_{i_{2q}}\right)$ muss v_r berühren. Sei ki_{2q-1} der Schlüssel des transition point von y zu Beginn von $access\left(x_{i_{2q-1}}\right)$ und ki_{2q} der Schlüssel des transition point von y zu Beginn von $access\left(x_{i_{2q}}\right)$. Gilt $ki_{2q-1} = ki_{2q}$ so muss der transition point von y in $access\left(x_{i_{2q}}\right)$ berührt worden sein. Gilt $ki_{2q-1} \neq ki_{2q}$ so muss der transition point von y, nach 1.1, in $access\left(x_{i_{2q-1}}\right)$ berührt worden sein. Aus der Konstruktion von $X_l^{r}{}'$ folgen daraus mindestens $\lfloor p/2 \rfloor \geq p/2 - 1$ Berührungen des transition point von y. Aufaddieren über alle Knoten ergibt eine Anzahl von Berührungen von transition points von zumindest $IB\left(X\right)/2 - |U| \geq IB\left(X\right)/2 - n$.

□

1.2 Aufbau des Tango Baum

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 $h = O\left(\log n\right)$ 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 1.1 mit Schlüsselmenge K. 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 Operation 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 der Variable kommt. Abbildung 3 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}.

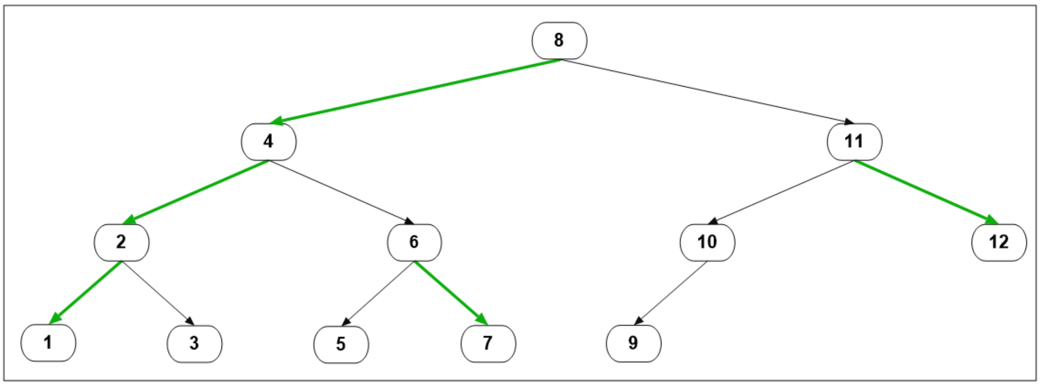

Abbildung 3: Die preferred childs werden durch die grünen Pfeile markiert.

Nun werden die preferred path des BST aus Abbildung 3 angegeben, wobei der Schlüssel jeweils als Bezeichner für den ihn enthaltenden Knoten verwendet wird.

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

Da jeder Knoten nur preferred child eines Knoten sein kann und an 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 vorschreibt zeigt Abbildung 4 nur eine mögliche Konstellation.

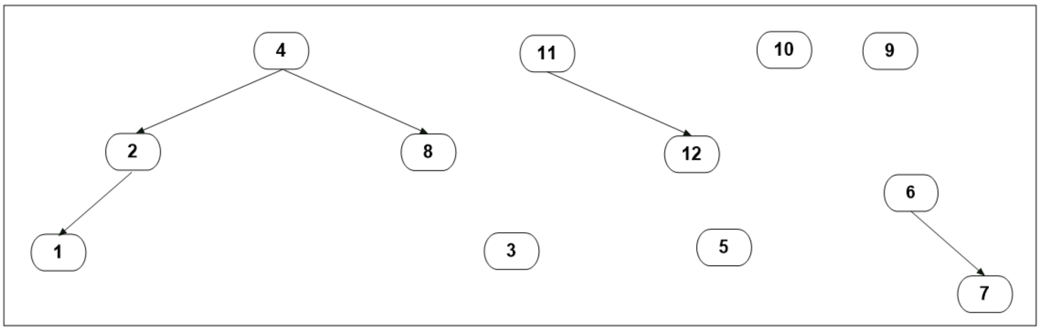

Abbildung 4: Hilfsbäume zu den preferred path aus dem Beispiel.

Sei H die Menge der erstellten Hilfbäume 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 Vaters 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 an dem die Standartvariante von *einfügen* 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 bei 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. Wird w_1 als linkes Kind eingefügt, so muss der Vorgänger von a in h_2, falls vorhanden, einen Schlüssel haben der kleiner als l ist. Wird w_1 als rechtes Kind eingefügt, so muss der Nachfolger von a in h_2, falls vorhanden, einen Schlüssel haben der größer als r ist. Im Tango Baum kann es also keine Verletzung der Links-Rechts-Beziehung geben.
Abbildung 5 zeigt unseren Tango Baum zum Beispiel. Die Wurzeln von Hilfsbäumen sind grün dargestellt.

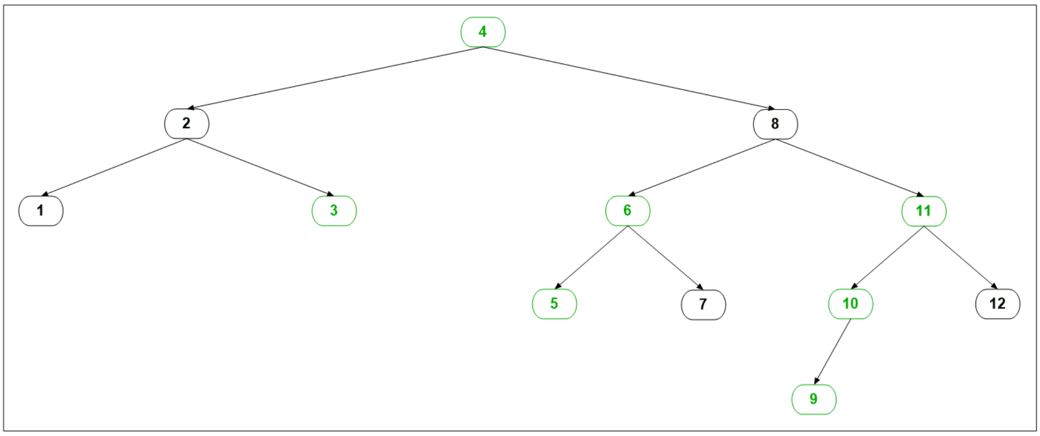

Abbildung 5: Tango Baum zu dem Beispiel.

Nehmen wir an auf T wird *access*(9) ausgeführt wird. Abbildung 3 zeigt den Zustand von P'.

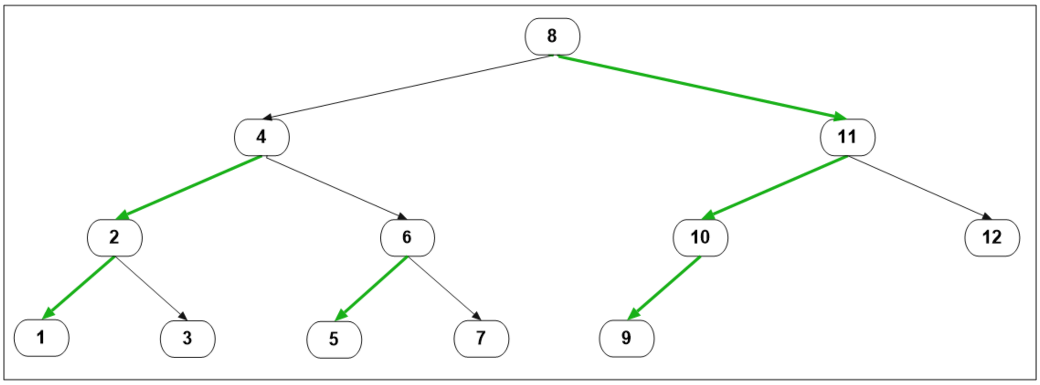

Abbildung 6: Preferred childs nach *access*(9).

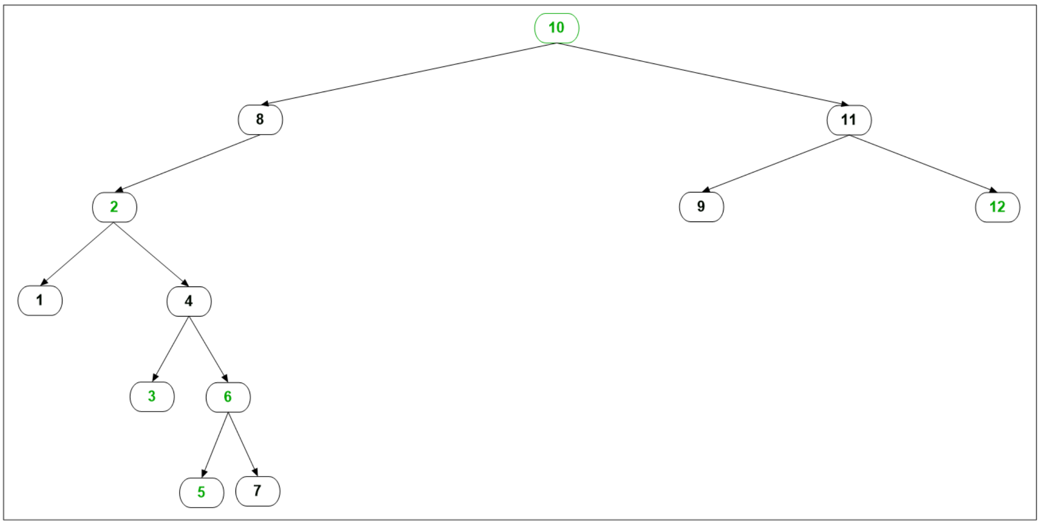

Abbildung 7: Tango Baum nach *access*(9).

Abbildung **??** zeigt einen möglichen Zustand von T'.

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

1.3 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. Dann gibt es nach 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.
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 sie gewöhnliche *search*(Schlüssel k) Operation an.

5. H bietet eine Operation *vereinigen(HB H_1, Schlüssel k, HB H_2)* an. HB ist eine Abkürzung für Hilfsbaum. 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. Die Operation gibt die Wurzel eines Hilfsbaum mit Schlüsselmenge $K_1 \cup K_2 \cup \{k\}$ zurück. Für die Laufzeit der Operation muss $O(\log(|K_1| + |K_2|))$ gelten.

6. H bietet eine Operation *aufteilen*(Schlüssel 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 ein Hilfsbaum mit Schlüsselmenge $K_l = \{i \in K \mid i < k\}$ sein. Das rechte Kind von v muss ein Hilfsbaum 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(Tiefe 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 mit *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. Zunächst werden Knoten l, l', r und r' in H gesucht. l ist der kleinsten Schlüssel eines Knoten v_l mit *depth* $> d$ in A . r ist der größte Schlüssel eines Knoten v_r mit *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 mit $depth > d$ 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 simultan.
A besteht aus Schlüsseln aus einem preferred path. Somit muss für jeden Schlüssel k eines Knotens mit $depth \leq d$ 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 $G = concatenate(B,\ l',\ E)$ aus.
8. Setze die *isRoot* Variable der Wurzel von G auf *true*
9. Ist w_a ein linkes bzw. rechtes Kind eines Knoten v ?. Wenn ja setze die Wurzel von G als linkes bzw. rechtes Kind von v

Abbildung 8 demonstriert den Ablauf nochmals und Abbildung 9 zeigt einen verkürzten Ablauf bei fehlendem r' Sei n die Anzahl der Knoten von A. Jeder der neun 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 einen 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, vergleiche 1.3. r' ist der größte Schlüssel in H_1 mit $l' < l$. l' 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 $F = concatenate(H_2,\ r',\ C)$ aus.

6. Führe $H = concatenate(B,\ l',\ F)$ aus.

7. Setze die *isRoot* Variable der Wurzel von H auf *true*

8. Ist w_1 ein linkes bzw. rechtes Kind eines Knoten v ?. Wenn ja setze die Wurzel von H als linkes bzw. rechtes Kind von v

Abbildung 10 demonstriert den Ablauf nochmals und Abbildung 11 zeigt einen verkürzten Ablauf bei fehlendem r'

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

1.4 Laufzeitanalyse

Literatur

[1] Erik D. Demaine, Dion. Harmon, John. Iacono, and Mihai. Patrascu. Dynamic optimality—almost. *SIAM Journal on Computing*, 37(1):240–251, 2007.

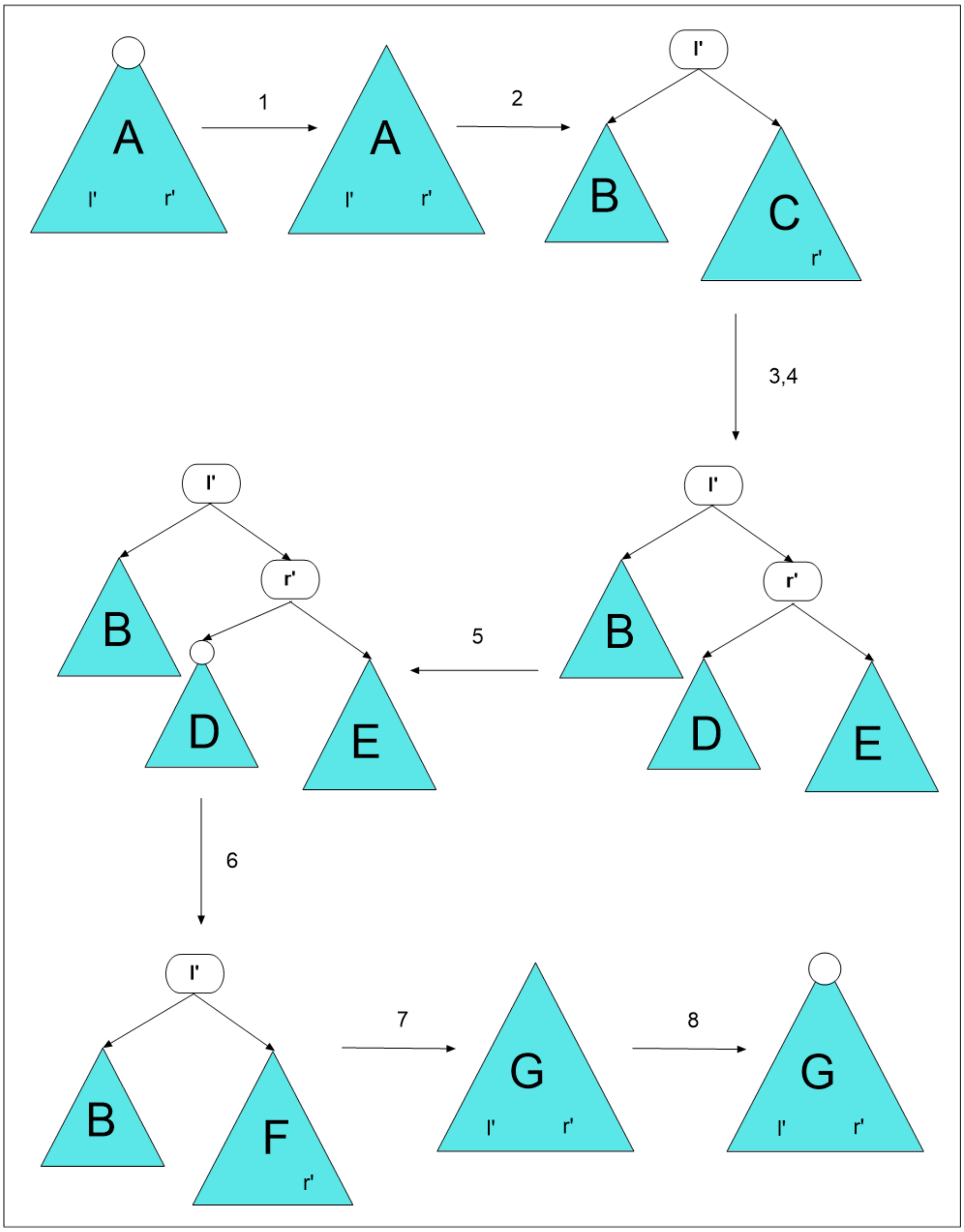

Abbildung 8: Ablauf von *cut(d)*. Die Abbildung basiert auf einer aus [1], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

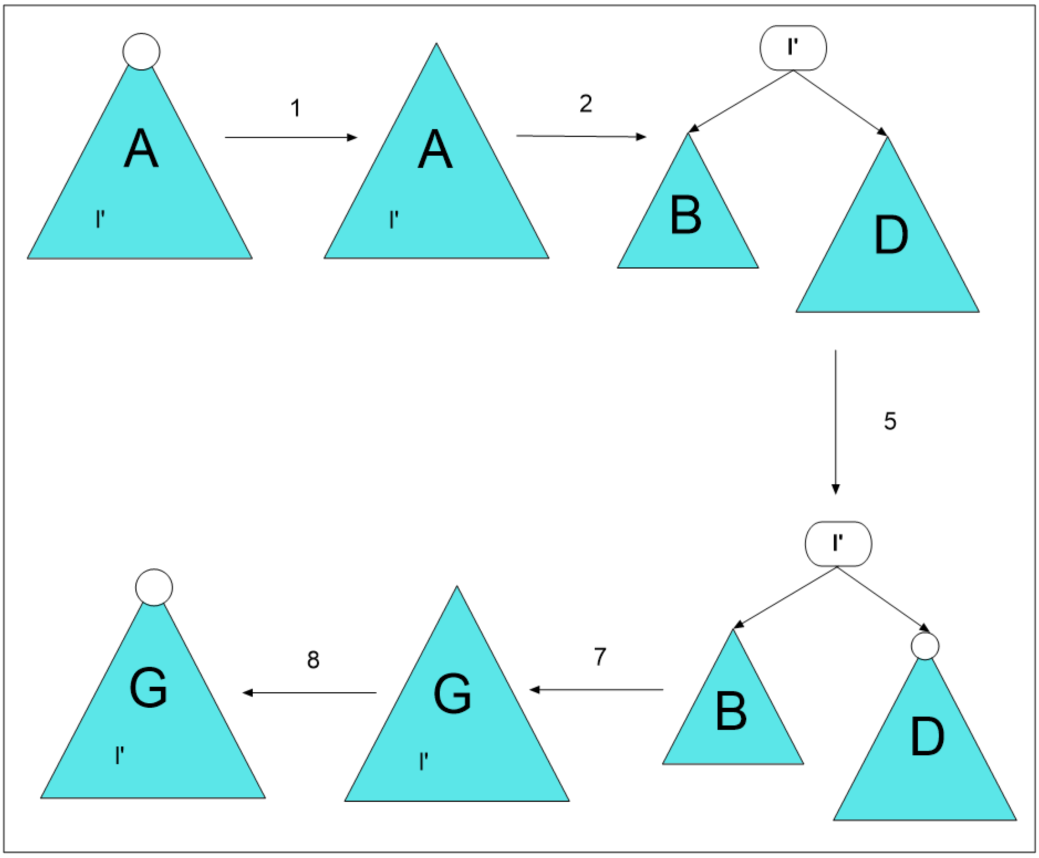

Abbildung 9: Ablauf von *cut(d)* bei fehlenden r'. Die Abbildung basiert auf einer aus [1], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

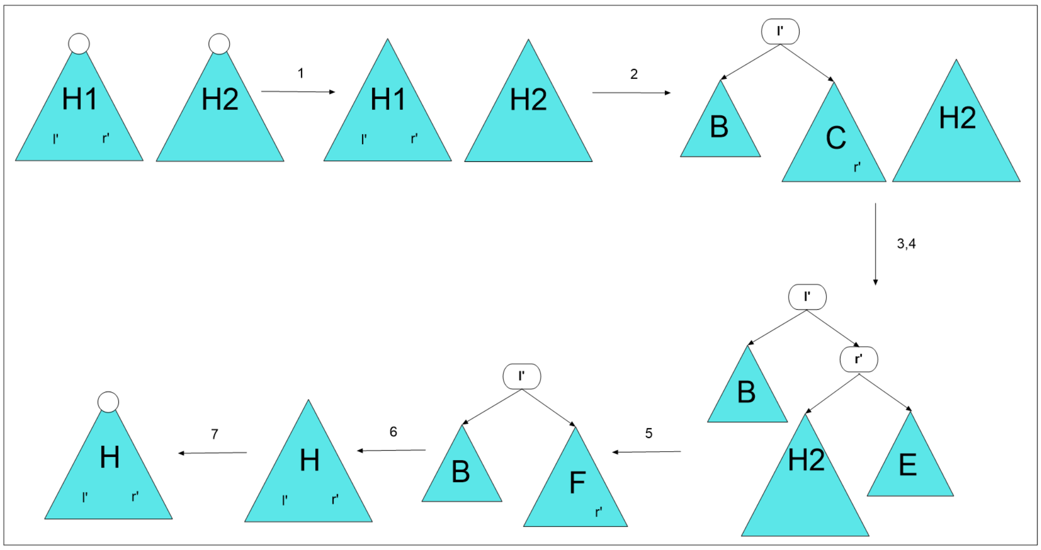

Abbildung 10: Ablauf von *join*(H_1, H_2). Die Abbildung basiert auf einer aus [1], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert

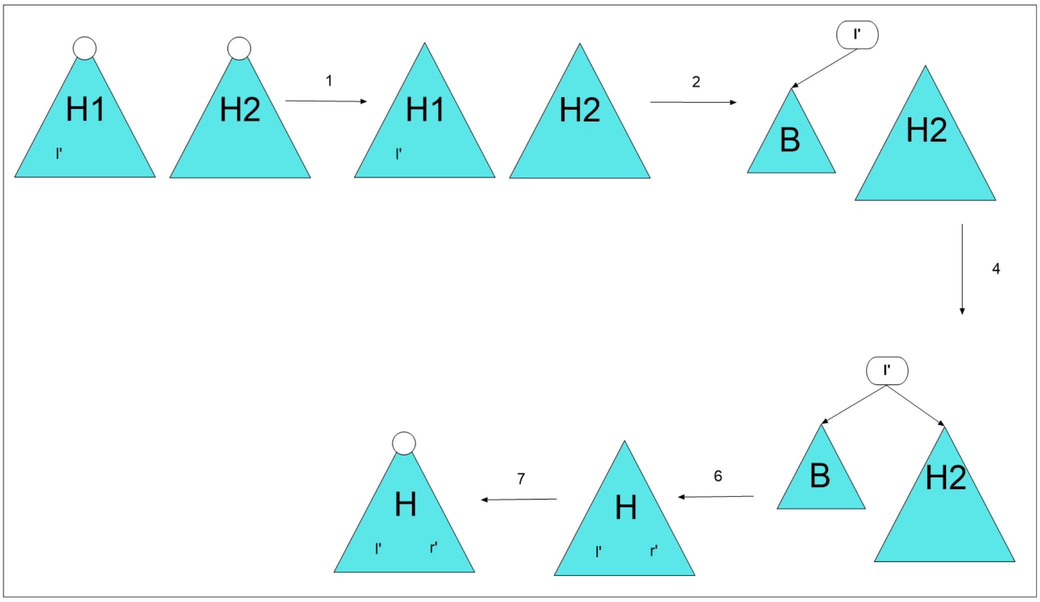

Abbildung 11: Ablauf von *join*(H_1, H_2) bei fehlendem r'. Die Abbildung basiert auf einer aus [1], Wurzeln von Hilfsbäumen werden mit einem Kreis markiert