Bachelorarbeit

Andreas Windorfer

20. Juni 2020

Inhaltsverzeichnis

1 Dynamische Optimalität

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

1.1 BST Zugriffsalgorithmus

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

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

2. Der Algorithmus führt eine Folge dieser Einzelschritte durch:

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

3. Zur Auswahl des nächsten Einzelschrittes können in den Knoten gespeicherte Hilfsdaten verwendet werden. Es kann nur auf die Hilfsdaten des Knotens zugegriffen werden (lesend oder schreibend), auf den p zeigt.

Es wird $n = |K|$ gesetzt. Außerdem werden hier pro Knoten nur Hilfsdaten in konstanter Größenordnung zugelassen.
Die Initialisierung sowie die Auswahl und Durchführung 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 $cost(X)$ der Zugriffsfolge mit $cost(X) = a + m$. Es muss zu jeder Schlüsselmenge und jeder Zugriffsfolge zumindest einen offline BST access algorithm geben, so dass die Kosten keines anderen niedriger sind. Diese Kosten werden als $OPT\,(X)$ bezeichnet.
In [1] wurde gezeigt, dass der Zustand eines BST mit maximal $2n-2$ Rotationen in jeden anderen gültigen BST Zustand mit der gleichen Schlüsselmenge überführt werden kann. Da bei der Berechnung der Kosten für $OPT(X)$, m ebenfalls als Summand vorkommt, können die zusätzlichen Kosten der online Varianten, für $m > n$ asymptotisch betrachtet vernachlässigt werden.
Als **dynamisch optimal** wird ein BST bezeichnet wenn er eine beliebige Zugriffssequenz X in $O\,(OPT\,(X))$ Zeit ausführen kann. Ein BST der jede Zugriffssequenz in $O\,(c \cdot OPT\,(X))$ Zeit ausführt, wird als **c-competitive** bezeichnet. Es konnte bis heute für keinen BST bewiesen werden, dass er dynamisch optimal ist. Es wurden aber mehrere untere Schranken für $OPT\,(X)$ gefunden. Eine davon wird nun vorgestellt.

1.2 Erste untere Schranke von Wilber

Robert Wilber hat in [2] zwei Methoden zur Berechnung unterer Schranken für die Laufzeit von BST access algorithm vorgestellt. Hier wird auf die Erste davon eingegangen. Im folgenden werden offline BST access algorithm betrachtet, bei denen nach einer *access(k)* Operation, der Knoten mit Schlüssel k die Wurzel des BST ist. Dies bezeichnet man als **standard offline BST access algorithm** asymptotisch betrachtet entsteht hierdurch kein Verlust der Allgemeinheit. Sei d die Tiefe von p zum Zeitpunkt t direkt vor der Terminierung von *access*. Dann sind mindestens Kosten $d+1$ entstanden. Mit d Rotationen kann 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 access algorithm mit $W(X,T)$. Im folgenden wird angenommen, dass
$K = \{i \in \mathbb{N} | i \in [j,k] \ \textit{mit}\ j,k \in \mathbb{N}\}$ 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, vergleiche Kapitel **??**. j ist der Schlüssel des Vaters von v, vor Ausführung der Rotation. Für ei-

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

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

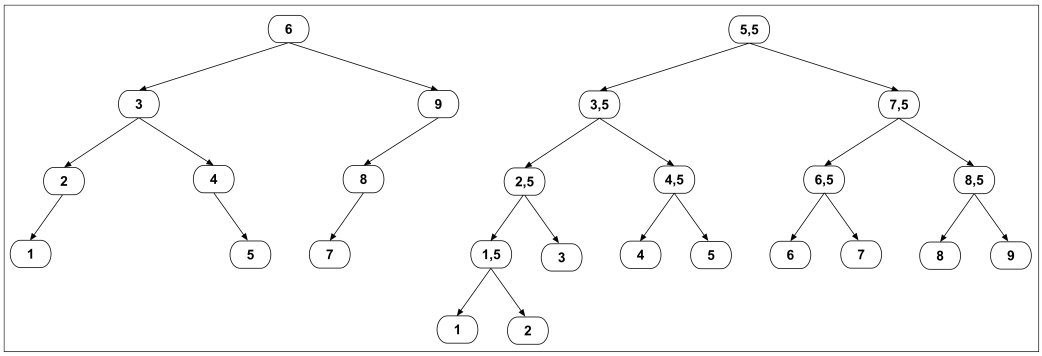

Abbildung 1: Rechts ist ein möglicher lower bound tree zum linken BST dargestellt.

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 internen Knoten von Y und m die Länge von X. Sei $u \in U$ und l der kleinste Schlüssel eines Blattes im Teilbaum mit Wurzel u, sowie r der größte Schlüssel eines solchen Blattes. Sei v der tiefste gemeinsame Vorfahre der Knoten mit Schlüssel aus $[l, r]$ in T. Sei o die Folge $o_0, o_1, .., o'_m$ mit $o_0 = key(v)$ und $o = key(v) \circ X_l^r$. $i \in [1, m]$ 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 $score\,(u) = |\{i \in \mathbb{N}\ i\ \textit{ist eine u-Transition}\}|$. 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 $OPT\,(X) \geq {}_X(T, Y, X)$ gezeigt werden. Dafür werden aber noch ein Lemma und einige Begriffe benötigt. Der **linke innere Pfad** $(v_0, v_1, .., v_n)$ eines Knotens u ist der längst mögliche Pfad für den gilt, v_0 ist das linke Kind von u und für $i \in \{1, .., n\}$,v_i ist rechtes Kind von v_{i-1}. Der **rechte innere Pfad** $(v_0, v_1, .., v_n)$ eines Knotens u ist der längst mögliche Pfad für den gilt, v_0 ist das rechte Kind von u und v_i ist linkes Kind von v_{i-1}. T_l^r ist ein mit $[l, r]$ von T abgeleiteter BST. Sei v_r 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 $key(v_r) \in [l, r]$ gelten. Denn hat v_r keine Kinder ist sein Schlüssel der Einzige aus $[l, r]$. Hat v_r ein Kind v_{rc} und $key(v_r) \notin [l, r]$, dann wäre v_{rc} ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten. Hat v_r zwei Kinder gibt es drei Fälle:

- Die Schlüssel beider Kinder sind aus $[l, r]$. Dann muss aufgrund der Links-Rechts-Beziehung $key(v_r)$ auch in $[l, r]$ enthalten sein.
- Der Schlüssel genau eines Kindes ist aus $[l, r]$. Sei v_{rc} nun das linke bzw. rechte Kind zweier Kinder von v_r, mit $key(v_{rc}) \notin [l, r]$. Gilt zusätzlich $key(v_r) \notin K_l^r$, dann wäre des rechte bzw. linke Kind von v_r ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten.
- Die Schlüssel beider Kinder liegen außerhalb von $[l, r]$. Dann muss $key(v_r)$ der Einzige in T_l^r enthaltene Schlüssel sein.

Ein Knoten u_r mit Schlüssel $key(v_r)$ 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_r.

1. Gilt $U = W$, beende das Verfahren.
2. Sei $u \in W$ ein Knoten mit $u \notin U$. Sei v der Knoten in T mit $key(u) = key(v)$. 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 u_l mit Schlüssel k_l und füge ihn als linkes Kind an u an. Füge u_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 u_r mit Schlüssel k_r und füge ihn als rechtes Kind an u an. Füge u_r zu W hinzu.

7. Füge u 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 2 dar.

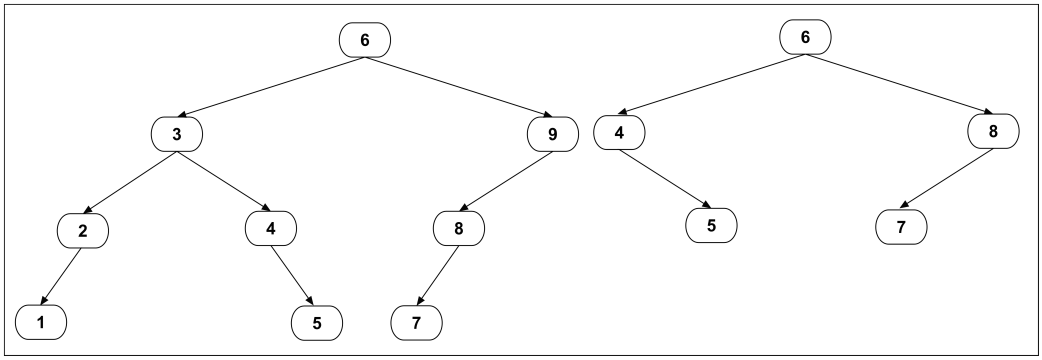

Abbildung 2: Links ein BST T. Rechts ein davon abgeleiteter BST T_4^8 .

Sei K_1 die Schlüsselmenge von T und K_2 die von T_l^r. Sei $K_l^r = K \cap \{i \in \mathbb{N} | i \in [l, r]\}$. 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 $P = v_0, .., v_n$ in T geben, mit $v_0 = v_r$, $v_n = v_k$ und für $i \in \{1, .., n\}$ v_i ist Kind von v_{i-1}. Sei m die Anzahl der Knoten in P, mit einem Schlüssel in $[l, r]$. Nun folgt Induktion über m.
Für $m = 1$ gilt $k = k_r$ und $k \in K_2$.
Induktionsschritt:
Sei w der Knoten mit der größten Tiefe in $v_0, .., v_{n-1}$, mit $key(w) \in K_2$. w muss existieren da $m > 1$. Nach Induktionsvoraussetzung gibt es einen Knoten u_k mit $key(u_k) = key(w)$ in T_l^r. Es sei $key(w) > key(v_k)$, der andere Fall

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

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 1.1. *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 Schlüssel aus $[l, r]$ entfernt oder hinzugefügt. Nun werden die vier Fälle betrachtet bei denen entweder $key(u)$ oder $key(c)$ 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 **??**, und da $key(u) < l$ gilt, 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 w' und w' bezüglich des Schlüssels ihres linken Kindes nicht unterscheiden.

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

3. v ist das linke Kind von u und $key(u) < l$:
   Von T' aus Fall 1 erreicht man nach Ausführung der Rotation mit dieser Konstellation wieder T aus Fall 1. Somit muss nichts weiter gezeigt werden.

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

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. Es werden die Rotationen (u, v) und (u^*, v^*) ausgeführt. Zu zeigen ist ${T'}_l^r = T_l^{r\prime}$.
In T verändern sich maximal drei innere Pfade.

1. Sei u_r das rechte Kind von u. Sei $u, u_r, v_1, .., v_n$ der linke innere Pfad von v, dann ist $u'_r, v'_1, .., v_n$ der linke innere Pfad von v'. Es gilt $key(u), key(u_r) \in [l, r]$. Damit ist u^* das linke Kind von v^* und ${u_r}'^*$ das linke Kind von v'^*.

2. Sei $v_1, .., v_n$ der rechte innere Pfad von u, dann ist $v', v'_1, .., v'_n$ der rechte innere Pfad von u', damit ist v' rechtes Kind von u'. Damit ist v'^* das rechte Kind von $u'*$.

3. Ist v das linke bzw. rechte Kind eines Knoten z mit $key(z) \in [r, l]$, dann sei $v, v_1, .., v_n$ der linke bzw. rechte innere Pfad von z. Dann ist $u', v', v'_1, .., v'_n$ der linke bzw. rechte innere Pfad von z'. Dann ist v^* das linke bzw. rechte Kind von z^* und u'^* das linke bzw. rechte Kind von z'^*.

Nun wird auf ${T'}_l^r$ die Rotation (u^*, v^*) ausgeführt. $u^{*\prime}$ ist linkes Kind von $v^{*\prime}$. $v^{*\prime}$ das rechte Kind von $u^{*\prime}$. Ist v^* das linke bzw. rechte Kind eines Knoten z^* mit $key(z^*) \in [r, l]$, dann ist $v^{*\prime}$ 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 1.1. *Es sei T ein standard offline BST access algorithm 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 internen Knoten von Y. Die Kosten zum Ausführen sind die *Anzahl der Einzelschritte* $+m$. Es reicht also aus zu zeigen, dass mehr als $\sum_{u \in U} {}_x(u)$ Rotationen benötigt werden. Es wird Induktion über $n = |K|$ angewendet. Sei $n = 1$, dann gibt es keinen internen Knoten in

Y und $\sum_{u \in U} {}_x(u) = 0$. Der Induktionsanfang ist somit gemacht. Im folgenden sei $n \geq 2$.
Sei $R = r_1, r_2, .., r_r$ 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, mit Schlüssel k_w, von Y. Sei Y^1 bzw. Y^2 der linke bzw. rechte Teilbaum von w. Es ist zu beachten, dass Y^1 ein lower bound tree zu $T_1^{k_w}$ ist und Y^2 einer zu $T_{k_w}^{\infty}$. $T_{i1}^{k_w}$ wird im folgenden als T_i^1 bezeichnet und $T_{ik_w}^{\infty}$ als T_i^2. Da $n \geq 2$ muss w ein interner Knoten sein. Sei $R^1 = r_1^1, r_2^1, .., r_{r^1}^1 = R_1^{k_w}$ und $R^2 = r_1^2, r_2^2, .., r_{r^2}^2 = R_{k_w}^{\infty}$. 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 r_M die Länge von M. Es muss $r = r^1 + r^2 + r_M$ 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 $k > w$ entfernt werden. X_2 entsteht durch entfernen aller Schlüssel $k < w$ aus X. Für $j \in \{1, 2\}$, sei U^j die Menge der internen Knoten von Y^j. Sei $T_0^{j*}, T_1^{j*}, .., T_{r_j}^{j*}$ die entstehende Folge, wenn aus $T_0^j, T_1^j, .., T_r^j$ die T_t^j entfernt werden für die $T_{t-1}^j = T_t^j$ gilt. Mit Lemma 1.2 kann T_{t-1}^{j*} durch Ausführung der Rotation r_t^j auf T_{t-1}^{j*} abgeleitet werden. Außerdem gilt 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 $r_j \geq \sum_{u \in U^j} {}_x(u)$.
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 $key(y) = \sigma_{w-1}$ und z der Knoten in T mit $key(z) = \sigma_w$. 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 $r_M \geq score\,(w)$.
Zusammengefasst ergibt sich:

$$r = r^1 + r^2 + r_M \geq \sum_{u \in U^1} {}_x(u) + \sum_{u \in U^2} {}_x(u) + {}_x(w)$$

□

Da T beliebig gewählt ist folgt direkt $OPT\,(X) \geq {}_X(T_0, Y, X)$. In diesem Abschnitt wurden BST access algorithm verwendet. Das Ergebnis lässt sich aber

natürlich direkt auf BST mit einer entsprechenden *access* Operation übertragen. Während einer Operationsfolge die ausschließlich aus *access* Operationen besteht, ist auch bei diesen die Schlüsselmenge konstant. Deshalb wird ab nun wieder nur von BST gesprochen.

1.3 bit reversal permutation

In diesem Abschnitt wird gezeigt, dass es Zugriffsfolgen mit Länge m für BST T gibt, so dass für die Laufzeit eines BST $\Theta(m \log n)$ gilt, mit n ist die Anzahl der Knoten von T. Hier werden speziell die Zugriffsfolgen betrachtet, die als **bit reversal permutation** bezeichnet werden. Auf $O(m \log n)$ wird hier nicht weiter eingegangen. Die balancierten BST garantieren jedoch diese Schranke und mit dem Rot-Schwarz-Baum wird später ein solcher noch vorgestellt. $\Omega(m \log n)$ wird mit Hilfe der ersten unteren Schranke von Wilber gezeigt und ein Beweis ist ebenfalls in [2] enthalten.
Nun wird zunächst der Aufbau einer solchen Zugriffsfolge eingegangen. Sei $l \in \mathbb{N}$ und $i \in \{0, 1, .., l-1\}$. Eine Folge $b_{l-1}, b_{l-2}, .., b_0$ mit $b_i \in \{0, 1\}$, kann als Zahl zur Basis 2 interpretiert werden. T enthält alle Schlüssel die als solche Folge dargestellt werden können. Die Schlüsselmenge von T ist deshalb $K_l = \{0, 1, .., 2^l - 1\}$. Die Funktion $br_l(k)\colon K \to K$ ist wie folgt definiert. Sei $b_{l-1}, b_{l-2}, .., b_0$ die Binärdarstellung von k, dann gilt

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

$br_l(k)$ gibt also gerade den Wert der „umgekehrten“ Binärdarstellung von k zurück. Die bit reversal permutation zu l ist die Zugriffsfolge
$br_l(0), br_l(1), .., br_l(2^l - 1)$. Diese wird ab jetzt mit X bezeichnet. Tabelle 1 zeigt die bit reveral Permutation mit $l = 4$. Sei y die Hälfte von $\max(K_l)$, also $y = 2^{l-1} - 0,5$. Da b_0 in den Binärdarstellungen zu $0, 1, 2^l - 1$ alterniert, alterniert b_{l-1} in X. Mit $2^{l-1} > y$ ergeben sich die Implikationen $br_l(k) < y \Rightarrow br_l(k+1) > y$ 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 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 1.1. *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*

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 $l = 4$

permutation zu l und Y der vollständig balancierte lower bound tree zu T. Dann gilt $W\,(X, T) \geq n \log_2 n + 1$.

Beweis. Sei U die Menge der internen Knoten von Y. Mit Satz 1.2 reicht es aus

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

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

deshalb, für $i \in \{1,2\}$,

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

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

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

□

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 1.2 ist dann R_0^y eine Folge von Rotationen zum ausführen von X auf T_0^y und die Länge von R kann nicht kleiner als die von R_0^y sein. Damit ist $OPT\,(X) = \Omega\,(m \log n)$.

1.4 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 (gewöhnlich bestimmten) 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 sicher $O(mn)$ 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, dass 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 $\Phi_m \geq \Phi_0 = 0$ Damit gilt für die Ausführungszeit der Folge $O(m)$.
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.

1.5 Obere Schranken für Zugriffssequenzen

In Abschnitt 1.3 wurde gezeigt, dass für die Ausführung einer beliebigen Zugriffsfolge X, mit Länge m, für einen BST T mit n Knoten im worst case Kosten von $\Omega(m \log n)$ angenommen werden müssen. Im folgendem werden einige obere Laufzeitschranken für Zugriffssequenzen vorgestellt. Es ist bekannt, dass es obere Schranken sind, da mit dem Splaybaum ein BST bekannt ist der jede Schranke einhält. Der Splaybaum wird später noch vorgestellt. Es wird wieder ohne Verlust der Allgemeinheit eine Schlüsselmenge $K = \{1, 2, .., n\}$ angenommen. Wenn nicht anders angegeben wird $X = x_1, x_2, .., x_m$ als Zugriffssequenz verwendet. Es wird $m \geq n$ und $m \geq n \log_2 n$ 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, Zugriffssequenzen schnell auszuführen, wenn ihre Schlüssel betragsmäßig nahe beieinander liegen. Sei $k_f \in K$. Ein BST erfüllt static finger wenn für die amortisierte Laufzeit von X

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

gilt.

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

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

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

Working Set Property Für x_i sei $J_i = \{j \in \mathbb{N} | j < i \wedge x_j = x_i\}$ Sei $t_{xi} = \max(J)$, falls J nicht leer ist, ansonsten $t_{xi} = 0$. t_{xi} liefert also den Index des vorherigen Zugriffes auf x_i, falls ein solcher existiert. Sei $w_i = |\{x_j | t_{xi} < j \leq i\}|$. Ein BST erfüllt das working set propery wenn seine amortisierte Laufzeit für X

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

Unified Property Das Unified Property kombiniert die oberen Eigenschaften zu einer. Die Bezeichner werden deshalb übernommen. Ein BST erfüllt das Unified Property, wenn für die wenn für die amortisierte Laufzeit von X:

$$O\left(\sum_{i=1}^{m} \log \min\{\frac{m}{q(x_i)}, |k_f - x_i| + 1, w_i\}\right)$$

gilt. Dass das Balanced Property enthalten ist, sieht man dem Ausdruck nicht direkt an, es gilt jedoch $|k_f - x_i| + 1 \leq n$.

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

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

gilt.

Literatur

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

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

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