

# Bachelorarbeit

Andreas Windorfer

9. Juni 2020

# Inhaltsverzeichnis

<b>1</b>	<b>Dynamische Optimalität</b>	<b>3</b>
1.1	BST Zugriffsalgorithmus . . . . .	3
1.2	Erste untere Schranke von Wilber . . . . .	4
1.3	Bit reversal permutation . . . . .	10
1.4	Amortisierte Laufzeitanalyse . . . . .	10
1.5	Zugriffssequenzen mit besonderen Eigenschaften . . . . .	12

# 1 Dynamische Optimalität

## 1.1 BST Zugriffsalgorithmus

Sei  $T$  ein BST mit der Schlüsselmenge  $K$ . Eine Operation, welche die gleiche Rückgabe wie *search* liefert, bei der aber nur  $k \in K$  als Parameter erlaubt sind wird *access* genannt. In diesem Kapitel werden Folgen solcher *access* Operationen betrachtet. Notiert wird eine solche Zugriffsfolge durch Angabe der Parameter. Bei der Zugriffsfolge  $x_1, x_2, \dots, x_m$  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 zum Start von dem Startzustand ausgegangen werden bei dem die Kosten am höchsten sind. Einen BST der lediglich die *access* Operation anbietet nennt man **BST access algorithm**, 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.

Da *access* die Schlüsselmenge nicht verändert ist diese bei BST access algorithm statisch und es wird  $n = |K|$  gesetzt. Außerdem werden hier pro Knoten nur Hilfsdaten in konstanter Größenordnung zugelassen. Zu beachten ist, dass dies eine Abhängigkeit zu  $n$  nicht ausschließt.

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 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, nennt man **c-competitive**. 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

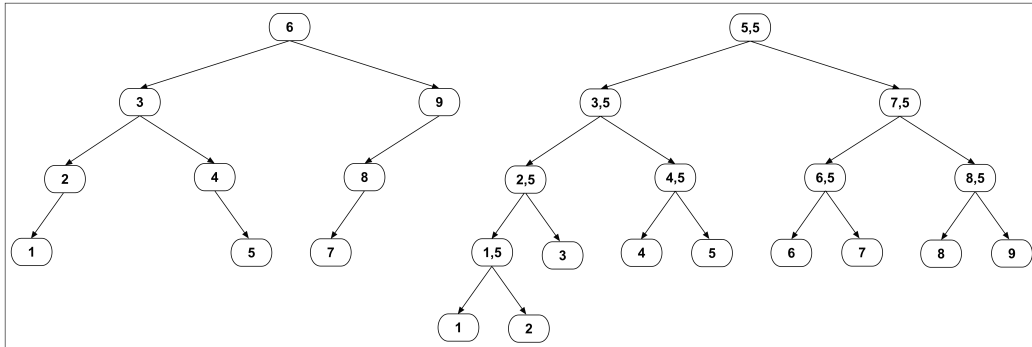
Robert Wilber hat in [2] zwei Methoden zur Berechnung unterer Schranken für die Laufzeit von BST access algorithm vorgestellt. Hier wird auf die Erste davon eingegangen. Im folgenden werden offline BST access algorithm betrachtet, bei denen nach einer  $access(k)$  Operation, der Knoten  $v$  mit Schlüssel  $k$  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. In diesem Kapitel wird eine Funktion  $key(v)$  verwendet, die zu einem Knoten  $v$  den Schlüssel von  $v$  liefert.

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

$K = \{i \in \mathbb{N} | i \in [j, k] \text{ mit } j, k \in \mathbb{N}\}$  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 auf dem die Rotation ausgeführt wird, vergleiche Kapitel ??.

$j$  ist der Schlüssel des Vaters von  $i$ , vor Ausführung der Rotation. Für eine Folge von Rotationen  $r = (i_1, j_1), (i_2, j_2), \dots, (i_n, j_n)$  erhält man die Folge  $r_x^y = (i_{1'}, j_{1'}), (i_{2'}, j_{2'}), \dots, (i_{m'}, j_{m'})$  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 man aus  $X$  alle Schlüssel  $k$  entfernt 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. Die Blätter werden 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 erkennt man, 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 Funktion  $_X(T, Y, X)$  definiert. 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 Schlüssel der internen Knoten von  $Y$  und  $m$  die Länge von  $X$ . Sei  $u \in U$  und  $l$  der kleinste Schlüssel eines Blattes im Teilbaum mit Wurzel  $u$ , sowie  $r$  der größte Schlüssel eines solchen Blattes. Sei  $v$  der tiefste gemeinsame Vorfahre der Knoten mit Schlüssel aus  $[l, r]$  in  $T$ . Sei  $o$  die Folge  $o_0, o_1, \dots, o'_m$

mit  $o_0 = v$  und  $o = v \circ X_l^r$ .  $i \in [1, m]$  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  $_x(u): U \rightarrow \mathbb{N}$  ist definiert durch  $_x(u) = |\{i \in \mathbb{N} \mid i \text{ ist eine } u\text{-Transition}\}|$ .

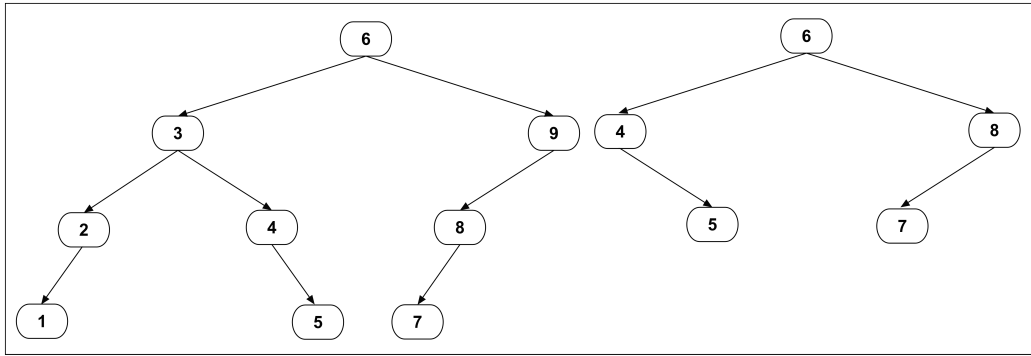
$$_x(T, Y, X) = m + \sum_{u \in U} _x(u)$$

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**  $(v_0, v_1, \dots, v_n)$  eines Knotens  $u$  ist der längste mögliche Pfad für den gilt,  $v_0$  ist das linke Kind von  $u$  und für  $i \in \{1, \dots, n\}$ ,  $v_i$  ist rechtes Kind von  $v_{i-1}$ . Der **rechte innere Pfad**  $(v_0, v_1, \dots, v_n)$  eines Knotens  $u$  ist der längste 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  $[l, r]$  aus  $T$  abgeleiteter BST. Sei  $v_r$  der tiefste gemeinsame Vorfahre der Knoten mit Schlüssel aus  $[l, r]$  in  $T$ . 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 mit Schlüsseln aus  $[l, r]$  dann muss aufgrund der Links-Rechts-Beziehung  $key(v_r)$  auch in  $[l, r]$  enthalten sein. 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 das rechte bzw. linke Kind von  $v_r$  ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten. Ein Knoten  $u_r$  mit Schlüssel  $key(v_r)$  ist 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

$K_2 \subseteq K_l^r$  erkennt man direkt an dem Verfahren zur Konstruktion von  $T_l^r$ .

$K_l^r \subseteq K_2$ :

Sei  $k \in K_l^r$  und  $v_k$  der Knoten in  $T$  mit  $\text{key}(v) = k$ . Es muss einen Pfad  $P = v_0, \dots, v_n$  in  $T$  geben, mit  $v_0 = v_r$ ,  $v_n = v_k$  und für  $i \in \{1, \dots, n\}$   $v_i$  ist Kind von  $v_{i-1}$ . Sei  $m$  die Anzahl der Knoten  $w$  in  $P$ , mit  $\text{key}(w) \in [l, r]$ . Nun folgt Induktion über  $m$ .

Für  $m = 1$  gilt  $k = k_r$  und  $k \in K_2$ .

$m + 1$ :

Sei  $w$  der tiefste Knoten in  $P$  mit  $v_0, \dots, v_{n-1}$ , mit  $\text{key}(w) \in K_2$ .  $w$  muss existieren da  $m > 1$ . Nach Induktionsvoraussetzung gibt es einen Knoten  $u_k$  mit  $\text{key}(u_k) = \text{key}(w)$  in  $T_l^r$ . Es sei  $\text{key}(w) > \text{key}(v_k)$ , der andere Fall ist symmetrisch. Ist  $v_k$  linkes Kind von  $w$ , dann enthält das linke Kind von  $u_k$  den Schlüssel  $\text{key}(v_k)$ . Anderenfalls gilt für alle  $v_j$  mit  $m < j < k$ ,  $\text{key}(v_j) < l < \text{key}(v_k)$ . Somit muss  $vm + 1$  ein linkes Kind sein und die Knoten in  $P$  mit größerer Tiefe als  $vm + 1$  müssen rechte Kinder sein. Damit ist auch in diesem Fall ein Knoten  $u_k$  mit  $\text{key}(u_k) = \text{key}(w)$  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  $(u, v)$  entsteht. Gilt  $u, v \in [l, r]$ , dann ist  $T_l'^r$  der BST der aus  $T_l^r$  durch Ausführen von  $(u, v)$  entsteht. Anderenfalls gilt  $T_l'^r = T_l^r$ .*

*Beweis.* Für  $u, v \notin [l, r]$  erkennt man  $T_l'^r = T_l^r$  direkt. Nun werden die vier Fälle betrachtet bei denen entweder  $u$  oder  $v$  in  $[l, r]$  liegen.

1.  $u$  ist das linke Kind von  $v$  und  $u < l$ :

Die Schlüsselmenge wird durch Rotationen nicht verändert. 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  $key(u) < l < key(w)$  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  $P_l = u \circ P_l'$  ?? 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  $u > r$ :

Links-Rechts-Symmetrisch zu Fall 1.

3.  $v$  ist das linke Kind von  $u$  und  $u < l$ :

Von  $T'$  nach Fall 1 erreicht man nach Ausführung der Rotation mit dieser Konstellation wieder  $T$ . Somit muss nichts weiter gezeigt werden.

4.  $v$  ist das rechte Kind von  $u$  und  $u > r$ :

Von  $T'$  nach Fall 2 erreicht man nach Ausführung der Rotation mit dieser Konstellation wieder  $T$ . Somit muss nichts weiter gezeigt werden.

Übrig bleibt noch die Konstellation  $u, v \in [l, r]$ . Betrachtet wird eine Rechtsrotation  $(u, v)$ , die Linksrotation ist wieder symmetrisch. In  $T_l^r$  muss es zwei Knoten  $w, x$  geben, mit  $key(u) = key(w)$ ,  $key(v) = key(x)$  und  $w$  muss linkes Kind von  $x$  sein. Es werden die Rotationen  $(u, v)$  und  $(w, x)$  ausgeführt. Zu zeigen ist  $T_l'^r = T_l'^r$ .

In  $T$  verändern sich maximal drei innere Pfade.



1. Sei  $u_r$  das rechte Kind von  $u$ . Sei  $u, u_r, v_1, \dots, v_n$  der linke innere Pfad von  $v$ , dann ist  $u'_r, v'_1, \dots, v'_n$  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  $v_1, \dots, v_n$  der rechte innere Pfad von  $u$ , dann ist  $v', v'_1, \dots, v'_n$  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  $v, \dots, v_n$  der linke bzw. rechte innere Pfad von  $z$ . Dann ist  $u', v', \dots, v'_n$  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.  $u'^*$  ist linkes Kind von  $v'^*$ .  $v'^*$  das rechte Kind von  $u'^*$ . Ist  $v^*$  das linke bzw. rechte Kind eines Knoten  $z^*$  mit  $key(z^*) \in [r, l]$ , dann ist  $v'^*$  das linke bzw. rechte Kind von  $z'^*$  und  $u'^*$  das linke bzw. rechte Kind von  $z'^*$ . Damit gilt  $T_l^{r'} = T_l^{r'}$ .

□

**Satz 1.1.** *Es sei  $T$  ein standard offline BST access algorithm mit Schlüsselmenge  $K = \{i \in \mathbb{N} | i \in [j, k] \text{ mit } j, k \in \mathbb{N}\}$ . Sei  $Y$  ein für  $T$  erstellter lower bound tree und  $X$  eine zu  $T$  erstellte Zugriffsfolge mit Länge  $m$ . Dann gilt  $OPT(X) \geq_X(T_0, Y, X)$ .*

*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  $R = r_1, r_2, \dots, r_r$  die Folge der insgesamt durchgeführten Rotationen. Für  $i \in \{1, \dots, r\}$  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  $Y$  und  $Y^1$  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  $T_{k_w}^\infty$ . Da  $n \geq 2$  muss  $w$  ein interner Knoten sein. Sei  $R^1 = r_1^1, r_2^1, \dots, r_{r^1}^1 = R_1^{k_w}$  und  $R^2 = r_1^2, r_2^2, \dots, r_{r^2}^2 = R_{k_w}^\infty$ . 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  $T_0^{j'}, T_1^{j'}, \dots, T_{r_j}^{j'}$  die entstehende Folge, wenn aus  $T_0^j, T_1^j, \dots, T_r^j$  die  $T_t^j$  entfernt werden für die  $T_{t-1}^j = T_t^j$  gilt. Mit Lemma ?? kann  $T_{t-1}^{j'}$  durch Ausführung der Rotation  $r_t^j$  auf  $T_{t-1}^{j'}$  abgeleitet werden, außerdem muss  $r^j = r_j$  gelten. 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  $R^j$  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  $\sigma = \text{key}(w) \circ X$ . Sei  $a$  eine  $w$ -Transition. Nun wird angenommen dass  $\sigma_{w-1} < \text{key}(w) \wedge \sigma_w > \text{key}(w)$ . Der andere Fall kann davon problemlos abgeleitet werden. Sei  $y$  der Knoten in  $T$  mit  $\text{key}(y) = \sigma_{w-1}$  und  $z$  der Knoten in  $T$  mit  $\text{key}(z) = \sigma_w$ . Nach  $\text{access}(\sigma_{w-1})$  ist  $y$  die Wurzel von  $T$ .  $z$  muss sich im rechten Teilbaum von  $y$  befinden. Nach  $\text{access}(\sigma_w)$  ist  $z$  die Wurzel von  $T$ .  $y$  muss sich im linken Teilbaum von  $z$  befinden. Somit muss während  $\text{access}(\sigma_w)$  die Rotation  $(\text{key}(z), \text{key}(y))$  ausgeführt worden sein.  $(\text{key}(z), \text{key}(y))$  muss in  $M$  enthalten sein. Für jede  $w$ -Transition ist also mindestens eine Rotation in  $M$  enthalten, also  $r_M \geq x(w)$ .

Zusammengefasst ergibt sich:

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

□

### 1.3 Bit reversal permutation

### 1.4 Amortisierte Laufzeitanalyse

Sei  $i \in \{0, \dots, m\}$ . 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 exakten 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 Schranken 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, bereits eingesparte Zeit durch schnell ausgeführte Operationen, den noch folgenden Operationen zum Verbrauchen zur

Verfügung zu stellen. Dabei wird insbesondere der aktuelle Zustand der zugrunde liegenden Datenstruktur vor und nach einer Operation betrachtet. Hier soll die amortisierte Laufzeitanalyse verwendet werden um im folgenden Abschnitt eine niedrigere obere Schranke als  $O(\log(n))$  für `einfügenFixup` zu finden. Es gibt drei Methoden zur amortisierten Analyse, hier wird 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.  $t_i$  steht für die **tatsächlichen Kosten** zum durchführen der  $i$ -ten Operation. Dabei handelt es sich um die exakten Kosten die beim 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.

$$A = \sum_{i=1}^m a_i = \sum_{i=1}^m (t_i + \Phi(D)_i - \Phi(D)_{i-1}) = \Phi(D)_m - \Phi(D)_0 + \sum_{i=1}^m t_i$$

Folgendes gilt für die Summe der  $t_i$ :

$$\begin{aligned} \sum_{i=1}^m t_i &= \sum_{i=1}^m (a_i - \Phi(D)_i + \Phi(D)_{i-1}) = \Phi(D)_0 - \Phi(D)_m + \sum_{i=1}^m a_i \\ &\Rightarrow \left( \Phi(D)_m \geq \Phi(D)_0 \Rightarrow \sum_{i=1}^m a_i \geq \sum_{i=1}^m t_i \right) \end{aligned}$$

Ist das Potenzial nach 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  $\Phi(D)_m \geq \Phi(D)_0$ . 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  $\Phi$  verwenden wir eine Funktion, welche die aktuelle Anzahl der im Stack enthaltenen Elemente zurück gibt.  $\Phi_0$  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 leeren 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 Zugriffssequenzen mit besonderen Eigenschaften

## 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.