Bachelorarbeit

Andreas Windorfer

4. Juni 2020

Inhaltsverzeichnis

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 $x_1, x_2, ...x_m$ 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. 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
$K = \{i \in \mathbb{N} | i \in [j, k] \;\; mit\; j, k \in \mathbb{N}\}$ 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 $r = (i_1, j_1), (i_2, j_2), .., (i_n, j_n)$ 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 $i < x \vee j > y$ 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.

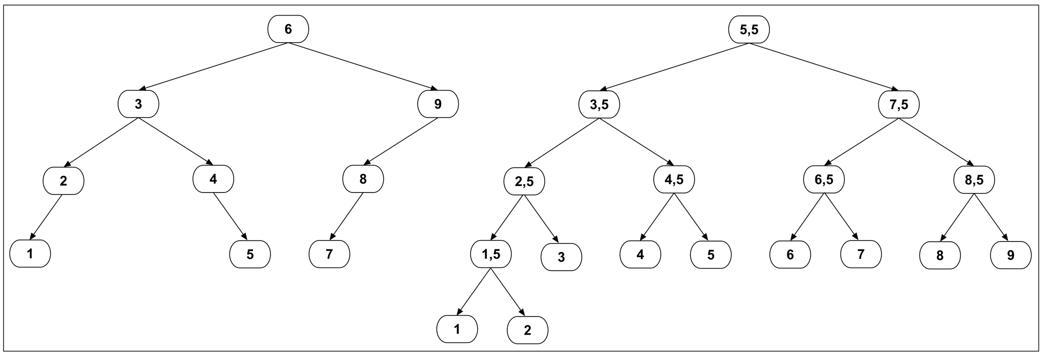

Abbildung 1: Rechts ist ein möglicher lower bound tree zum linken BST dargestellt.

Nun wird die Funkion $_X(T, Y, X)$ 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 $o_0, o_1, .., o'_m$ 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 ${}_x(u)\colon U \to \mathbb{N}$ ist definiert durch ${}_x(u) = |\{i \in \mathbb{N}\ \textit{i ist eine u-Transition}\}|$.

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

Im eigentlichen Satz wird f $OPT\,(X) \geq {}_X(T, Y, X)$ 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 dem gilt, v_0 ist das linke Kind von u und 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 von T abgeleiteter BST mit Schlüsselmenge $K_l^r = \{i \in K | l \leq i \leq r\}$. Sein nun v ein Knoten in T mit dem linken inneren Pfad P_l und rechtem inneren Pfad P_r und u ein Konten in T_l^r mit dem gleichen Schlüssel wie v. Sei k_l der Schlüssel des Knotens v_l mit dem kleinsten Index in P_l, so dass gilt $k_l \in K_l^r$, wenn ein solcher Knoten v_l existiert. Existiert v_l nicht, so hat u kein linkes Kind, anderenfalls ist der Knoten mit Schlüssel v_l sein linkes Kind. Das rechte Kind von u wird auf die gleiche Art und Weise aus P_r bestimmt. So erstellt muss T_l^r ein BST sein, so dass für zwei Schlüssel $k_1, k_2 \in K_l^r$ gilt, ist der Knoten mit Schlüssel k_2 ein Nachfahre des Knoten mit Schlüssel k_1 in T_l^r, dann gilt dies auch für T. Direkt ersichtlich ist, dass kein Knoten in T_l^r mehr als zwei Kinder hat und dass die Links-Rechts-Beziehung der Schlüssel eingehalten wird. Auf drei andere Punkte wird nun noch etwas eingegangen.

1. T_l^r hat eine Wurzel:
   Der Schlüssel k_r des tiefstem gemeinsamen Vorfahren v_r der Knoten mit Schlüsseln aus K_l^r in T, muss selbst in K_l^r liegen. Denn hat v_r keine Kinder ist sein Schlüssel der Einzige aus K_l^r. Hat v_r ein Kind v_{rc} und $k_r \notin K_l^r$, dann wäre v_{rc} ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten. Hat v_r zwei Kinder mit Schlüsseln aus K_l^r dann muss der von k_r auch in K_l^r enthalten sein. Sei v_{rc} nun das linke bzw. rechte Kind zweier Kinder von v_r, dessen Schlüssel nicht in K_l^r liegt. Gilt zusätzlich $k_r \notin K_l^r$, dann wäre des rechte bzw. linke Kind von v_r ein tieferer gemeinsamer Vorgänger der entsprechenden Knoten. v_r kann auf keinem inneren Pfad eines anderen Knoten mit Schlüssel aus K_l^r liegen, damit kann der Knoten mit Schlüssel k_r keinen Vater in T_l^r haben.

2. Für die Schlüsselmenge K von T_l^r gilt $K \subsetneqq K_l^r$:
   Sei $k \in K_l^r$ und $k < k_r$. Sei v_k der Knoten mit Schlüssel k in T. Da v_r

ein Vorfahre von v_k ist muss v_r einen Vater v_p mit Schlüssel k_p haben. Gilt $k_p \in K_l^r$ dann ist der Knoten mit Schlüssel k_p Vater des Knotens mit Schlüssel k in T_l^r. Ansonsten muss $k > k_p$ gelten und v_k ist ein rechtes Kind. Sei $v_0, v_1, .., v_n$ ein Pfad in T mit $v_0 = v_k$ und v_i ist der Vater von v_{i-1}.

1.3 Bit reversal permutation

1.4 Amortisierte Laufzeitanalyse

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 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 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, 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.

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

Folgendes gilt für die Summe der t_i:

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

$$\Rightarrow \left(\Phi\left(D\right)_m \geq \Phi\left(D\right)_0 \Rightarrow \sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} t_i\right)$$

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 $\Phi\left(D\right)_m \geq \Phi\left(D\right)_0$. 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 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 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.