Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

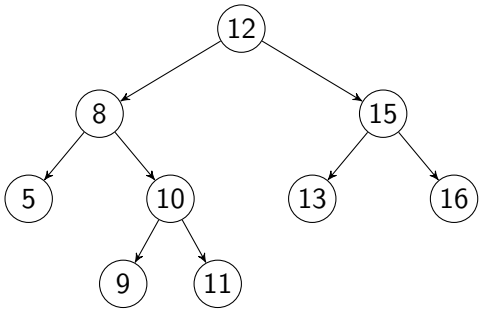
Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

6. April 2016

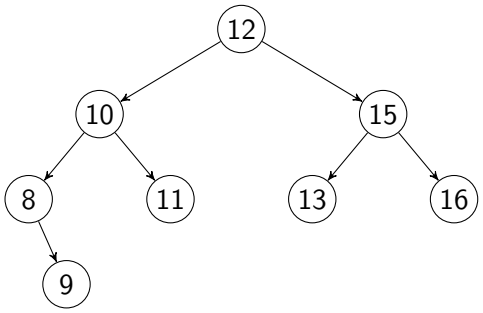
Datenstrukturen & Algorithmen Lösungen zu Blatt 5 FS 16

Lösung 5.1 *AVL-Bäume.*

a) Es entsteht der folgende AVL-Baum:



b) Wird der Schlüssel 5 aus dem oben dargestellten AVL-Baum gelöscht, so ist der Elternknoten mit Schlüssel 8 nicht mehr balanciert. Die Rebalancierung ergibt folgenden Baum:



c) Man kann T durch eine geeignete Einfüge-Reihenfolge aus einem anfangs leeren AVL-Baum erzeugen, ohne dass Rotationen durchgeführt werden müssen. Dazu werden die Schlüssel so geordnet, dass alle Schlüssel, die in Knoten mit einer gewissen Tiefe h gespeichert sind, vor Schlüsseln aus Knoten mit einer Tiefe $h' > h$ eingefügt werden. Schlüssel, die in Knoten mit gleicher Tiefe gespeichert sind, können in beliebiger Reihenfolge eingefügt werden. Wir bezeichnen den Baum, der nach der i-ten Einfügeoperation entsteht, mit T_i. Wir beweisen mit vollständiger Induktion, dass ein Schlüssel in einem Knoten mit Tiefe h in T, der als $(i+1)$-ter Schlüssel in T_i eingefügt wird, genau in der Position gespeichert wird, in der er auch in T ist und dass für diese Operation keine Rotationen notwendig sind.

Induktionsverankerung ($h = 0$): Sicherlich gilt unsere Behauptung für den ersten Schlüssel, der in den leeren Baum T_0 eingefügt wird, da er in der Wurzel von T und auch von T_1 gespeichert ist. Da T_1 nur einen Knoten enthält, treten auch keine Rotationen auf.

Induktionsannahme: Wir nehmen an, dass unsere Behauptung für alle Schlüssel gilt, die in einer Tiefe von höchstens h in T gespeichert sind.

Induktionsschritt ($h \to h+1$): Sei T_i der Baum mit genau den Schlüsseln aus Knoten mit Tiefe $h' \leq h$ in T. Beim Einfügen eines (beliebigen) Schlüssel von T mit Tiefe $h+1$ wird derselbe Suchpfad in T_i durchlaufen wie in T, da alle Vorgänger in T auch an den gleichen Positionen in T_i sind. Daher wird dieser Schlüssel auch an der korrekten Stelle gespeichert. Angenommen, dass durch das Einfügen ein Knoten v nicht mehr balanciert ist. Da T_i balanciert war, endet der Suchpfad des neuen Schlüssels in einem der beiden Teilbäume von v. Wir nehmen oBdA an, dass er im linken Teilbaum mit Höhe h_l gespeichert wird. Dann hat der rechte Teilbaum höchstens Höhe $h_r \leq h_l - 2$, da v nicht balanciert ist. Alle Knoten von T mit Tiefe kleiner gleich h sind bereits in T_i gespeichert und auch v ist an derselben Position in T und T_i. Daher wird in den folgenden Einfügeoperationen kein weiterer Schlüssel zum rechten Teilbaum von v hinzugefügt. Damit wäre aber auch der Baum T nicht balanciert, da lediglich der linke Teilbaum von v weiter wachsen kann. Ein Widerspruch.

Lösung 5.2 *Erweiterte Suchbäume.*

Jeder Knoten v des Baums speichert die Anzahl g_v gerader Elemente, die sich im linken Teilbaum von v befinden.

Bei der Anfrage nach der Anzahl gerader Elemente, die kleiner als k sind, suchen wir wie gewohnt nach k. Besuchen wir bei der Suche einen Knoten v und ist k grösser als der in v gespeicherte Schlüssel, so inkrementieren wir den anfangs mit 0 initialisierten Zähler um g_v bzw. um $g_v + 1$, falls der Schlüssel von v gerade ist. Wird der Schlüssel k schliesslich in einem Knoten w gefunden, so addieren wir zusätzlich noch g_w zum Zähler.

Um die Frage zu beantworten, wie viele gerade Zahlen zwischen k_1 und k_2 der Baum speichert, prüfen wir zuerst, ob $k_2 = k_1 + 1$. Ist das der Fall, so kann die Anfrage sofort beantwortet werden, da keine geraden Schlüssel echt zwischen k_1 und k_2 liegen können. Andernfalls benutzen wir das eben beschriebene Verfahren, um die Zahlen L_1 und L_2 von geraden Elementen kleiner k_1+1 bzw. k_2 zu bestimmen. Subtrahiert man nun L_1 von L_2, so erhält man die Anzahl gerader Schlüssel, die echt zwischen k_1 und k_2 liegen.

Beim Einfügen überprüfen wir zuerst, ob i bereits im Baum gespeichert ist. Ist das nicht der Fall, erfolgt das Einfügen eines ungeraden Elements i wie gewohnt, und wir setzen im entsprechend neu eingefügten Knoten v den Zähler g_v auf 0. Falls i gerade ist, müssen wir zusätzlich jedes Mal, wenn wir einen Knoten u treffen, der einen grösseren Schlüssel als i enthält, g_u noch um eins erhöhen (da i in den linken Teilbaum von u eingefügt wird).

Zum Löschen eines Elements i suchen wir zunächst ebenfalls nach i, um zu prüfen, ob i im Baum gespeichert ist. Ist dies nicht der Fall, dann sind wir fertig. Andernfalls führen wir eine erneute Suche nach i durch. Ist i gerade, dann dekrementieren wir g_u bei jedem Knoten u, wenn wir auf der Suche nach i im linken Teilbaum von u fortfahren. Am Ende der Suche wird i in einem Knoten v gefunden. Nun unterscheiden wir drei Fälle.

1. **Fall:** *v ist ein Blatt.* Dann kann v direkt gelöscht werden und wir sind fertig.

2. **Fall:** *v hat genau einen Nachfolger.* Auch hier kann v ohne Weiteres gelöscht und durch seinen entsprechenden Nachfolger ersetzt werden.

3. **Fall:** *v hat zwei Nachfolger.* Dann enthält der linke Teilbaum von v den symmetrischen Vorgänger j von i. Wir ersetzen den in v gespeicherten Schlüssel durch j. Danach wird der Knoten w, der j enthält, wie gewohnt gelöscht. Ist j gerade, dann enthält der linke Teilbaum von v einen geraden Schlüssel weniger, also muss in dem Fall g_v noch dekrementiert

werden.

Alle drei neuen Operationen haben einen konstanten Aufwand pro Knoten und daher keinen negativen Einfluss auf die Laufzeit der drei Operationen Einfügen, Löschen und Suchen. Die Laufzeit ist also weiterhin $\mathcal{O}(h)$, wobei h die Höhe des Baums bezeichnet.

Lösung 5.3 *Amortisierte Analyse.*

Eine gute Wahl ist $k = 2n$. Dies bedeutet, dass ein Array doppelter Länge erzeugt wird, sobald das aktuelle Array voll ist. Um zu zeigen, dass mit dieser Wahl jede Einfügeoperation konstante amortisierte Kosten hat, führen wir eine amortisierte Analyse durch. Dazu definieren wir eine Potentialfunktion, welche jedem Array-Zustand einen Wert zuordnet (diesen Wert kann man intuitiv als "Kontostand" interpretieren).

Zur Erinnerung: In der amortisierten Analyse mittels Potentialfunktion definiert man Φ_i als das Potential nach der i-ten Operation. Die i-te Operation habe tatsächliche Kosten t_i. Dann sind die *amortisierten Kosten* der i-ten Operation definiert als $a_i := t_i + \Phi_i - \Phi_{i-1}$. Mit dieser Definition folgt für eine Folge von m Operationen

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (t_i + \Phi_i - \Phi_{i-1}) = \left(\sum_{i=1}^{m} t_i \right) + \Phi_m - \Phi_0, \tag{1}$$

und somit erhalten wir

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m. \tag{2}$$

Wenn es also gelingt, die amortisierten Kosten jeder Operation sowie den Term $\Phi_0 - \Phi_m$ abzuschätzen, dann erhält man so auch eine Abschätzung für die tatsächlichen Gesamtkosten. Wird die Potenzialfunktion beispielsweise so gewählt, dass $\Phi_m \geq \Phi_0$ für jedes m, dann folgt $\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i$, d.h. die tatsächlichen Gesamtkosten können durch die Summe der amortisierten Kosten nach oben abgeschätzt werden.

a) Wir definieren das Potential (bzw. den Kontostand) eines Arrays der Grösse n als

$$6 \cdot \text{Anz. d. Elem. in der oberen Hälfte des Arrays (also in Positionen } \frac{n}{2} + 1, \ldots, n). \tag{3}$$

Zu beachten ist, dass sich auch n ändert, wenn das Array vergrössert wird. Aus der Definition folgt $\Phi_0 = 0$ (anfangs ist das Array leer), und weil Φ_i mit dieser Definition nie negativ sein kann, ist auch klar, dass $\Phi_i \geq 0$ für $i > 0$ gilt, also insbesondere $\Phi_m \geq \Phi_0$. Wir müssen also nur noch untersuchen, wie gross die amortisierten Kosten einer Einfügeoperation sind. Dazu unterscheiden wir zwei Fälle: Wenn bei der i-ten Einfügeoperation das Array nicht verdoppelt wird (d.h. es ist noch nicht voll), dann sind $t_i = 1$ und $\Phi_i - \Phi_{i-1} \leq 6$ ($= 0$ falls das Array noch nicht halb voll ist, und $= 6$ sonst), und somit $a_i \leq 1 + 6 = 7$. Wenn bei der i-ten Einfügeoperation das Array von Grösse n auf Grösse $2n$ verdoppelt wird, sind die tatsächlichen Kosten

$$t_i = \underbrace{2n}_{\text{Array anlegen}} + \underbrace{n}_{\text{Elemente kopieren}} + \underbrace{1}_{\text{neues Element einfügen}} = 3n + 1 \tag{4}$$

und die Potenzialdifferenz beträgt

$$\Phi_i - \Phi_{i-1} = 6 \cdot (1 - \frac{n}{2}) = 6 - 3n. \tag{5}$$

Die amortisierten Kosten sind in diesem Fall $a_i = 3n + 7 - 3n = 7$, also ebenfalls konstant.

b) Wir zeigen nun, dass auch für das Entfernen eine amortisiert konstante Laufzeit möglich ist. Dabei wird das Array erst dann von der Grösse n auf die Grösse $n/2$ verkleinert, wenn es nur noch $n/4$ Elemente im Array hat, und *nicht* bereits, wenn es noch $n/2$ Elemente hat. Dies verhindert, dass die Arraygrösse stets verdoppelt und wieder halbiert wird, wenn man in ein Array zuerst $n/2$ Elemente einfügt und dann immer abwechselnd eines einfügt und dieses gleich wieder löscht.

Für die amortisierte Analyse definieren wir das Potential (bzw. den Kontostand) eines Arrays der Grösse n als

$$3 \cdot \text{Anz. leerer Positionen in der unteren Hälfte des Arrays (also in Pos. } 1, \ldots, \frac{n}{2}). \quad (6)$$

Erneut haben wir $\Phi_i \geq 0$ für $i \geq 0$. Offenbar muss jedes zu löschende Elemente vorher eingefügt worden sein. Man erkennt leicht, dass nach jeder Einfügeoperation keine leeren Positionen in der unteren Hälfte des Arrays existieren, wenn wir mit einem Array der Grösse 1 oder 2 beginnen. Folglich ist $\Phi_0 = 0$.

Wenn bei einer Löschoperation i das Array nicht halbiert wird, gilt $a_i = 1 + 0$, falls das gelöschte Element in der oberen Hälfte des Arrays liegt, oder $a_i = 1 + 3$, falls das gelöschte Element in der unteren Hälfte liegt. Wird dagegen bei der Löschoperation i das Array halbiert, dann gilt

$$t_i = \underbrace{n/2}_{\text{Array anlegen}} + \underbrace{n/4}_{\text{Elemente kopieren}} = \frac{3}{4}n, \quad (7)$$

und die Potenzialdifferenz beträgt

$$\Phi_i - \Phi_{i-1} = 3 \cdot (1 - n/4). \quad (8)$$

Somit sind die amortisierten Kosten in diesem Fall $a_i = \frac{3}{4}n + 3 \cdot (1 - n/4) = 3$. Für jede Löschoperation sind also die amortisierten Kosten konstant (genauer: $a_i \leq 4$).

Es ist nun leicht zu sehen, dass man mit der Potentialfunktion

$$6 \cdot (\text{Anzahl Elemente in der oberen Hälfte des Arrays} + \text{Anzahl leerer Positionen in der unteren Hälfte des Arrays}) \quad (9)$$

auch für *beliebige* Folgen von Einfüge- und Löschoperationen zeigen kann, dass die amortisierten Kosten jeder Operation konstant sind.

Bemerkung: Man könnte natürlich auch weitere Kosten in die Analyse mit einbeziehen, z.B. wenn man davon ausgeht, dass das Löschen eines Arrays der Länge n die Kosten $\Theta(n)$ (und nicht 0) hat.

Lösung 5.4 *AVL-Bäume (**Programmieraufgabe**).*

Die Funktion `rotateLeft` kann analog zur entsprechenden Funktion für die Rotation nach rechts gebaut werden:

```
private Node rotateLeft(Node a) {

    Node b = a.right;
    b.parent = a.parent;

    a.right = b.left;

    if (a.right != null)
        a.right.parent = a;

    b.left = a;
    a.parent = b;

    if (b.parent != null) {
        if (b.parent.right == a) {
            b.parent.right = b;
        } else {
            b.parent.left = b;
        }
    }

    setBalance(a, b);

    return b;
}
```

Beim Einfügen wird der Suchpfad für den einzufügenden Schlüssel traversiert. Wird dabei der Schlüssel gefunden, so wird abgebrochen. Andernfalls wird der Schlüssel in ein neues Blatt eingefügt und der Baum anschliessend rebalanciert.

```
public boolean insert(int key) {
    if (root == null)
        root = new Node(key, null);
    else {
        Node n = root;
        Node parent;
        while (true) {
            if (n.key == key)
                return false;

            parent = n;

            boolean goLeft = n.key > key;
            n = goLeft ? n.left : n.right;

            if (n == null) {
                if (goLeft) {
                    parent.left = new Node(key, parent);
                } else {
                    parent.right = new Node(key, parent);
                }
                rebalance(parent);
                break;
            }
        }
    }
    return true;
}
```

Beim Löschen muss beachtet werden, dass gegebenenfalls immer der symmetrische Nachfolger für die Ersetzung verwendet wird:

```
public void delete(int delKey) {
    if (root == null)
        return;
    Node n = root;
    Node parent = root;
    Node delNode = null;
    Node child = root;

    while (child != null) {
        parent = n;
        n = child;
        child = delKey >= n.key ? n.right : n.left;
        if (delKey == n.key)
            delNode = n;
    }

    if (delNode != null) {
        delNode.key = n.key;

        child = n.left != null ? n.left : n.right;

        if (root.key == delKey) {
            root = child;
        } else {
            if (parent.left == n) {
                parent.left = child;
            } else {
                parent.right = child;
            }
            if (child != null) child.parent = parent;
            rebalance(parent);
        }
    }
}
```