SS 2005

Einführung in die Informatik IV

Ernst W. Mayr

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2005SS/info4/index.html.de

27. Juni 2005

2.6 Vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren sind vergleichsbasiert, d.h. sie greifen auf Schlüssel k, k' (außer in Zuweisungen) nur in Vergleichsoperationen der Form $k < k'$ zu, verwenden aber nicht, dass etwa $k \in \mathbb{N}_0$ oder dass $k \in \Sigma^*$.

Satz 174

Jedes vergleichsbasierte Sortierverfahren benötigt im worst-case mindestens

$$n \operatorname{ld} n + O(n)$$

Vergleiche und hat damit Laufzeit $\Omega(n \log n)$.

2.6 Vergleichsbasierte Sortierverfahren

Alle bisher betrachteten Sortierverfahren sind vergleichsbasiert, d.h. sie greifen auf Schlüssel k, k' (außer in Zuweisungen) nur in Vergleichsoperationen der Form $k < k'$ zu, verwenden aber nicht, dass etwa $k \in \mathbb{N}_0$ oder dass $k \in \Sigma^*$.

Satz 174

Jedes vergleichsbasierte Sortierverfahren benötigt im worst-case mindestens

$$n \operatorname{ld} n + O(n)$$

Vergleiche und hat damit Laufzeit $\Omega(n \log n)$.

Beweis:

Wir benutzen ein so genanntes Gegenspielerargument (engl. adversary argument). Soll der Algorithmus n Schlüssel sortieren, legt der Gegenspieler den Wert eines jeden Schlüssels immer erst dann fest, wenn der Algorithmus das erste Mal auf ihn in einem Vergleich zugreift. Er bestimmt den Wert des Schlüssels so, dass der Algorithmus möglichst viele Vergleiche durchführen muss.

Beweis:

Wir benutzen ein so genanntes Gegenspielerargument (engl. adversary argument). Soll der Algorithmus n Schlüssel sortieren, legt der Gegenspieler den Wert eines jeden Schlüssels immer erst dann fest, wenn der Algorithmus das erste Mal auf ihn in einem Vergleich zugreift. Er bestimmt den Wert des Schlüssels so, dass der Algorithmus möglichst viele Vergleiche durchführen muss.

Am Anfang (vor der ersten Vergleichsoperation des Algorithmus) sind alle $n!$ Sortierungen der Schlüssel möglich, da der Gegenspieler jedem Schlüssel noch einen beliebigen Wert zuweisen kann.

Beweis:

Seien nun induktiv vor einer Vergleichsoperation $A[i] < A[j]$ des Algorithmus noch r Sortierungen der Schlüssel möglich.

Beweis:

Seien nun induktiv vor einer Vergleichsoperation $A[i] < A[j]$ des Algorithmus noch r Sortierungen der Schlüssel möglich.

Falls der Gegenspieler die Werte der in $A[i]$ bzw. $A[j]$ gespeicherten Schlüssel bereits früher festgelegt hat, ändert sich die Anzahl der möglichen Sortierungen durch den Vergleich nicht, dieser ist redundant.

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Nach k Vergleichen des Algorithmus sind also immer noch $n/2^k$ Sortierungen möglich. Der Algorithmus muss jedoch Vergleiche ausführen, bis nur noch eine Sortierung möglich ist (die dann die Ausgabe des Sortieralgorithmus darstellt).

Beweis:

Andernfalls kann der Gegenspieler einen oder beide Schlüssel so festlegen, dass immer noch mindestens $r/2$ Sortierungen möglich sind (wir verwenden hier, dass die Schlüssel stets paarweise verschieden sind).

Nach k Vergleichen des Algorithmus sind also immer noch $n/2^k$ Sortierungen möglich. Der Algorithmus muss jedoch Vergleiche ausführen, bis nur noch eine Sortierung möglich ist (die dann die Ausgabe des Sortieralgorithmus darstellt).

Damit

$$\#\text{Vergleiche} \geq \lceil \mathrm{ld}(n!) \rceil = n \,\mathrm{ld}\, n + O(n)$$

mit Hilfe der Stirlingschen Approximation für $n!$. □

2.7 Bucket-Sort

Bucket-Sort ist ein nicht-vergleichsbasiertes Sortierverfahren. Hier können z.B. n Schlüssel aus

$$\{0, 1, \ldots, B-1\}^d$$

in Zeit $O(d(n+B))$ sortiert werden, indem sie zuerst gemäß dem letzten Zeichen auf B Behälter verteilt werden, die so entstandene Teilsortierung dann stabil gemäß dem vorletzten Zeichen auf B Behälter verteilt wird, usw. bis zur ersten Position. Das letzte Zeichen eines jeden Schlüssels wird also als das niedrigwertigste aufgefasst.

Bucket-Sort sortiert damit n Schlüssel aus $\{0, 1, \ldots, B-1\}^d$ in Zeit $O(nd)$, also linear in der Länge der Eingabe (gemessen als Anzahl der Zeichen).

2.7 Bucket-Sort

Bucket-Sort ist ein nicht-vergleichsbasiertes Sortierverfahren. Hier können z.B. n Schlüssel aus

$$\{0, 1, \ldots, B-1\}^d$$

in Zeit $O(d(n+B))$ sortiert werden, indem sie zuerst gemäß dem letzten Zeichen auf B Behälter verteilt werden, die so entstandene Teilsortierung dann stabil gemäß dem vorletzten Zeichen auf B Behälter verteilt wird, usw. bis zur ersten Position. Das letzte Zeichen eines jeden Schlüssels wird also als das niedrigwertigste aufgefasst.

Bucket-Sort sortiert damit n Schlüssel aus $\{0, 1, \ldots, B-1\}^d$ in Zeit $O(nd)$, also linear in der Länge der Eingabe (gemessen als Anzahl der Zeichen).

3. Suchverfahren

Es ist eine Menge von Datensätzen gegeben, wobei jeder Datensatz D_i durch einen eindeutigen Schlüssel k_i gekennzeichnet ist. Der Zugriff zu den Datensätzen erfolgt per Zeiger über die zugeordneten Schlüssel, so dass wir uns nur mit der Verwaltung der Schlüssel beschäftigen.

I. A. ist die Menge der Datensätze dynamisch, d.h. es können Datensätze neu hinzukommen oder gelöscht werden, oder Datensätze bzw. Schlüssel können geändert werden.

3. Suchverfahren

Es ist eine Menge von Datensätzen gegeben, wobei jeder Datensatz D_i durch einen eindeutigen Schlüssel k_i gekennzeichnet ist. Der Zugriff zu den Datensätzen erfolgt per Zeiger über die zugeordneten Schlüssel, so dass wir uns nur mit der Verwaltung der Schlüssel beschäftigen.

I. A. ist die Menge der Datensätze dynamisch, d.h. es können Datensätze neu hinzukommen oder gelöscht werden, oder Datensätze bzw. Schlüssel können geändert werden.

Definition 175

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Definition 175

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Definition 175

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Definition 175

Ein Wörterbuch (engl. dictionary) ist eine Datenstruktur, die folgende Operationen auf einer Menge von Schlüsseln effizient unterstützt:

1. is_member(k): teste, ob der Schlüssel k in der Schlüsselmenge enthalten ist;
2. insert(k): füge den Schlüssel k zur Schlüsselmenge hinzu, falls er noch nicht vorhanden ist;
3. delete(k): entferne den Schlüssel k aus der Schlüsselmenge, falls er dort vorhanden ist.

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Wir betrachten zuerst Suchbäume. Wir nehmen an, dass die Schlüssel aus einer total geordneten Menge, dem Universum $\mathcal{U}$ stammen.

Es gibt zwei grundsätzlich verschiedene Ansätze, um Wörterbücher zu implementieren:

- Suchbäume
- Hashing (Streuspeicherverfahren)

Wir betrachten zuerst Suchbäume. Wir nehmen an, dass die Schlüssel aus einer total geordneten Menge, dem Universum $\mathcal{U}$ stammen.

Sind die Schlüssel nur in den Blättern des Suchbaums gespeichert, sprechen wir von einem externen Suchbaum, ansonsten von einem internen Suchbaum (wo dann der Einfachheit halber Schlüssel *nur* in den internen Knoten gespeichert werden).

3.1 Binäre/natürliche Suchbäume

In binären Suchbäumen gilt für alle Knoten x

- $key(x)$ ist größer als der größte Schlüssel im linken Unterbaum von x;
- $key(x)$ ist kleiner als der kleinste Schlüssel im rechten Unterbaum von x.

Die Wörterbuch-Operationen werden wie folgt realisiert:

1. is_member(k): beginnend an der Wurzel des Suchbaums, wird der gesuchte Schlüssel k mit dem am Knoten gespeicherten Schlüssel k' verglichen. Falls $k < k'$ ($k > k'$), wird im linken (rechten) Unterbaum fortgefahren (falls der Unterbaum leer ist, ist der Schlüssel nicht vorhanden), ansonsten ist der Schlüssel gefunden.
2. insert(k): es wird zuerst geprüft, ob k bereits im Suchbaum gespeichert ist; falls ja, ist die Operation beendet, falls nein, liefert die Suche die Position (den leeren Unterbaum), wo k hinzugefügt wird.
3. delete(k): es wird zuerst geprüft, ob k im Suchbaum gespeichert ist; falls nein, ist die Operation beendet, falls ja, sei x der Knoten, in dem k gespeichert ist, und es sei x' der *linkste* Knoten im rechten Unterbaum von x (x' ist nicht unbedingt ein *Blatt*!); dann wird $key(x')$ im Knoten x gespeichert und x' durch seinen rechten Unterbaum ersetzt (falls vorhanden) bzw. gelöscht. Spezialfälle, wie z.B., dass x' nicht existiert, sind kanonisch zu behandeln.

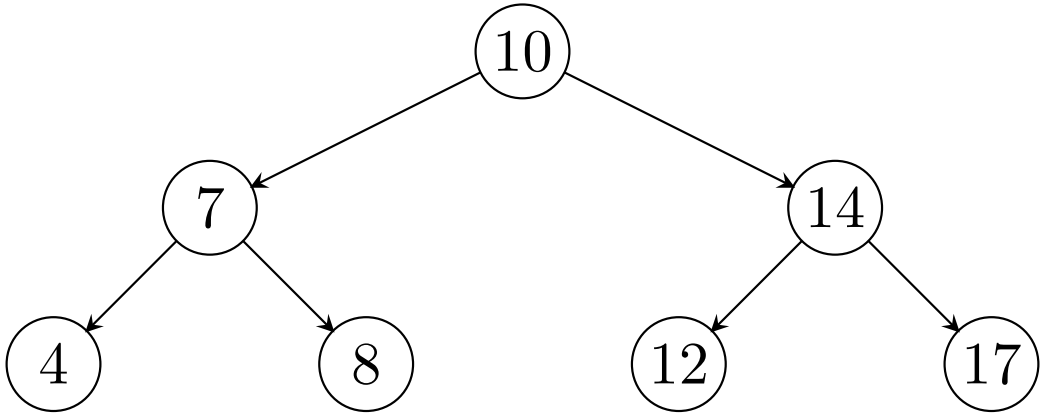
Die Wörterbuch-Operationen werden wie folgt realisiert:

1. is_member(k): beginnend an der Wurzel des Suchbaums, wird der gesuchte Schlüssel k mit dem am Knoten gespeicherten Schlüssel k' verglichen. Falls $k < k'$ ($k > k'$), wird im linken (rechten) Unterbaum fortgefahren (falls der Unterbaum leer ist, ist der Schlüssel nicht vorhanden), ansonsten ist der Schlüssel gefunden.
2. insert(k): es wird zuerst geprüft, ob k bereits im Suchbaum gespeichert ist; falls ja, ist die Operation beendet, falls nein, liefert die Suche die Position (den leeren Unterbaum), wo k hinzugefügt wird.
3. delete(k): es wird zuerst geprüft, ob k im Suchbaum gespeichert ist; falls nein, ist die Operation beendet, falls ja, sei x der Knoten, in dem k gespeichert ist, und es sei x' der *linkste* Knoten im rechten Unterbaum von x (x' ist nicht unbedingt ein *Blatt*!); dann wird $key(x')$ im Knoten x gespeichert und x' durch seinen rechten Unterbaum ersetzt (falls vorhanden) bzw. gelöscht. Spezialfälle, wie z.B., dass x' nicht existiert, sind kanonisch zu behandeln.

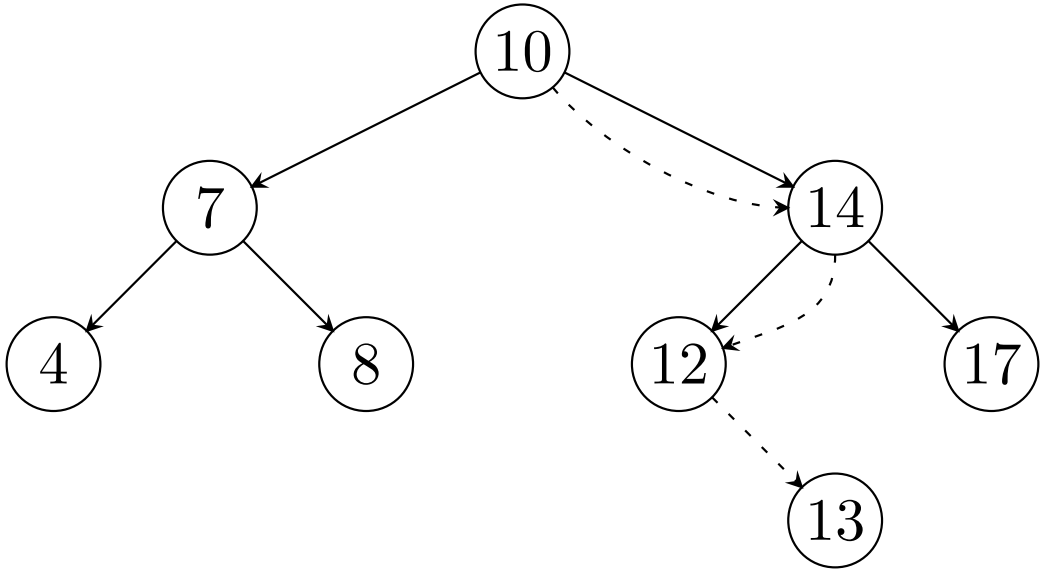
Die Wörterbuch-Operationen werden wie folgt realisiert:

1. is_member(k): beginnend an der Wurzel des Suchbaums, wird der gesuchte Schlüssel k mit dem am Knoten gespeicherten Schlüssel k' verglichen. Falls $k < k'$ ($k > k'$), wird im linken (rechten) Unterbaum fortgefahren (falls der Unterbaum leer ist, ist der Schlüssel nicht vorhanden), ansonsten ist der Schlüssel gefunden.
2. insert(k): es wird zuerst geprüft, ob k bereits im Suchbaum gespeichert ist; falls ja, ist die Operation beendet, falls nein, liefert die Suche die Position (den leeren Unterbaum), wo k hinzugefügt wird.
3. delete(k): es wird zuerst geprüft, ob k im Suchbaum gespeichert ist; falls nein, ist die Operation beendet, falls ja, sei x der Knoten, in dem k gespeichert ist, und es sei x' der *linkste* Knoten im rechten Unterbaum von x (x' ist nicht unbedingt ein *Blatt*!); dann wird $key(x')$ im Knoten x gespeichert und x' durch seinen rechten Unterbaum ersetzt (falls vorhanden) bzw. gelöscht. Spezialfälle, wie z.B., dass x' nicht existiert, sind kanonisch zu behandeln.

Beispiel 176

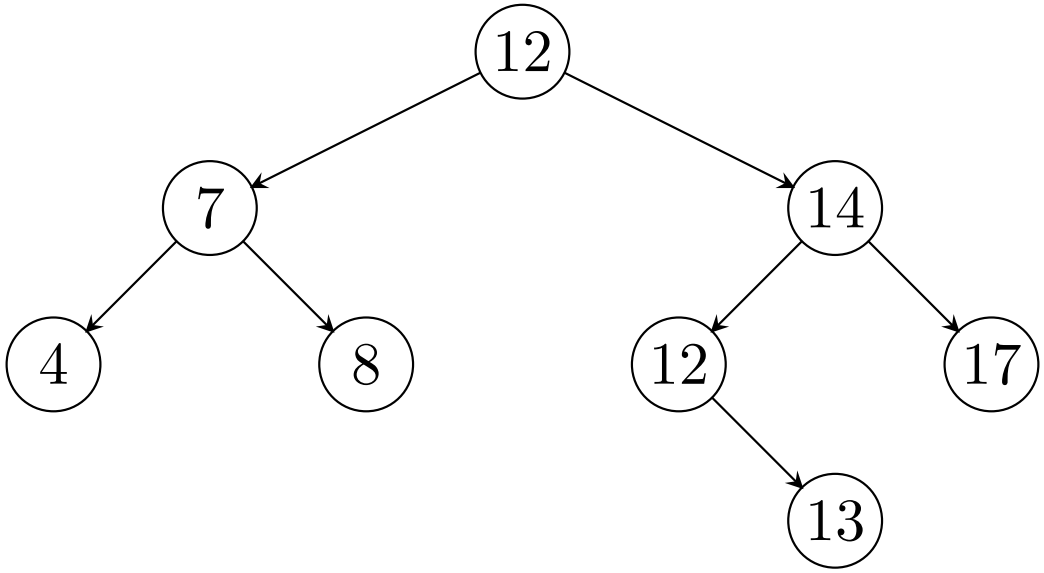


Beispiel 176



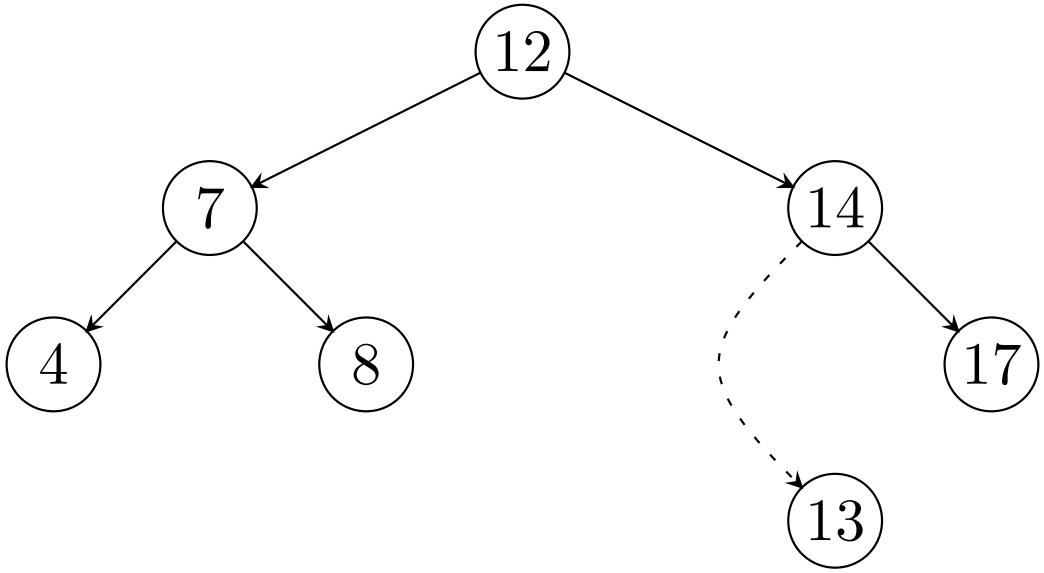
Beispiel 177

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Beispiel 177

Der Schlüssel 10 (also die Wurzel) wird gelöscht:



Satz 178

In einem natürlichen Suchbaum der Höhe h benötigen die Wörterbuch-Operationen jeweils Zeit $O(h)$.

Beweis:
Folgt aus der obigen Konstruktion! □

Bemerkung:
Das Problem bei natürlichen Suchbäumen ist, dass sie sehr entartet sein können, z.B. bei n Schlüsseln eine Tiefe von $n-1$ haben. Dies ergibt sich z.B., wenn die Schlüssel in aufsteigender Reihenfolge eingefügt werden.

Satz 178

In einem natürlichen Suchbaum der Höhe h benötigen die Wörterbuch-Operationen jeweils Zeit $O(h)$.

Beweis:

Folgt aus der obigen Konstruktion! □

Bemerkung:

Das Problem bei natürlichen Suchbäumen ist, dass sie sehr entartet sein können, z.B. bei n Schlüsseln eine Tiefe von $n-1$ haben. Dies ergibt sich z.B., wenn die Schlüssel in aufsteigender Reihenfolge eingefügt werden.

Satz 178

In einem natürlichen Suchbaum der Höhe h benötigen die Wörterbuch-Operationen jeweils Zeit $O(h)$.

Beweis:

Folgt aus der obigen Konstruktion! □

Bemerkung:

Das Problem bei natürlichen Suchbäumen ist, dass sie sehr entartet sein können, z.B. bei n Schlüsseln eine Tiefe von $n-1$ haben. Dies ergibt sich z.B., wenn die Schlüssel in aufsteigender Reihenfolge eingefügt werden.

3.2 AVL-Bäume

AVL-Bäume sind interne binäre Suchbäume, bei denen für jeden Knoten gilt, dass sich die Höhe seiner beiden Unterbäume um höchstens 1 unterscheidet. AVL-Bäume sind nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962) benannt.

Satz 179

Ein AVL-Baum der Höhe h enthält mindestens $F_{h+3} - 1$ und höchstens $2^{h+1} - 1$ Knoten, wobei F_n die n-te Fibonacci-Zahl ($F_0 = 0, F_1 = 1$) und die Höhe die maximale Anzahl von Kanten auf einem Pfad von der Wurzel zu einem Blatt ist.

3.2 AVL-Bäume

AVL-Bäume sind interne binäre Suchbäume, bei denen für jeden Knoten gilt, dass sich die Höhe seiner beiden Unterbäume um höchstens 1 unterscheidet. AVL-Bäume sind nach ihren Erfindern G. Adelson-Velskii und Y. Landis (1962) benannt.

Satz 179

Ein AVL-Baum der Höhe h enthält mindestens $F_{h+3} - 1$ und höchstens $2^{h+1} - 1$ Knoten, wobei F_n die n-te Fibonacci-Zahl ($F_0 = 0, F_1 = 1$) und die Höhe die maximale Anzahl von Kanten auf einem Pfad von der Wurzel zu einem Blatt ist.

Beweis:

Die obere Schranke ist klar, da ein Binärbaum der Höhe h höchstens

$$\sum_{j=0}^{h} 2^j = 2^{h+1} - 1$$

Knoten enthalten kann.

Beweis:

Induktionsanfang:

1. ein AVL-Baum der Höhe $h = 0$ enthält mindestens einen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$
2. ein AVL-Baum der Höhe $h = 1$ enthält mindestens zwei Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

Beweis:

Induktionsanfang:

1. ein AVL-Baum der Höhe $h = 0$ enthält mindestens einen Knoten, $1 \geq F_3 - 1 = 2 - 1 = 1$
2. ein AVL-Baum der Höhe $h = 1$ enthält mindestens zwei Knoten, $2 \geq F_4 - 1 = 3 - 1 = 2$

Beweis:

Induktionsschluss: Ein AVL-Baum der Höhe $h \geq 2$ mit minimaler Knotenzahl hat als Unterbäume der Wurzel einen AVL-Baum der Höhe $h-1$ und einen der Höhe $h-2$, jeweils mit minimaler Knotenzahl. Sei

$$f_h := 1 + \text{minimale Knotenanzahl eines AVL-Baums der Höhe } h\,.$$

Dann gilt demgemäß

$$\begin{aligned} f_0 &= 2 && = F_3 \\ f_1 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{also} \\ f_h &= f_{h-1} + f_{h-2} && = F_{h+3} \end{aligned}$$

□

Korollar 180

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Satz 181

In einem AVL-Baum mit n Schlüsseln kann in Zeit $O(\log n)$ festgestellt werden, ob sich ein gegebener Schlüssel in der Schlüsselmenge befindet oder nicht.

Beweis:
Klar! □

Korollar 180

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Satz 181

In einem AVL-Baum mit n Schlüsseln kann in Zeit $O(\log n)$ festgestellt werden, ob sich ein gegebener Schlüssel in der Schlüsselmenge befindet oder nicht.

Beweis:
Klar! □

Korollar 180

Die Höhe eines AVL-Baums mit n Knoten ist $\Theta(\log n)$.

Satz 181

In einem AVL-Baum mit n Schlüsseln kann in Zeit $O(\log n)$ festgestellt werden, ob sich ein gegebener Schlüssel in der Schlüsselmenge befindet oder nicht.

Beweis:

Klar! □