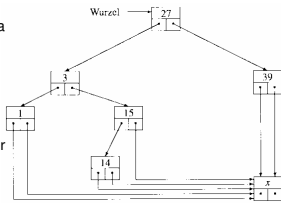


Suchbaum mit Wächter/Stopper

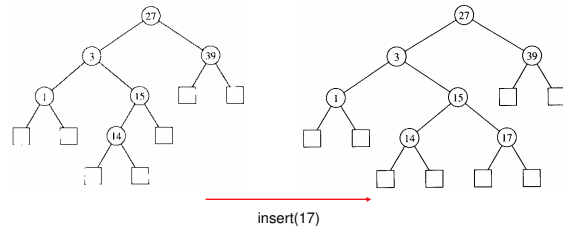
- Ansatz
 - Wie beim Suchen in Listen/Arrays mit Wächter/Stopper
 - Test auf „nicht gefunden“ entfällt, da Wert garantiert gefunden wird
 - Könnte in Java auch durch Ausnahmebehandlung ersetzt werden
- Implementierung
 - In einem Suchbaum sind alle Blätter identisch (ist somit eigentlich kein Baum mehr)
 - In dieses einzige Blatt wird der gesuchte Wert geschrieben



PI-2: Bäume 2

7

Einfügen in einen Suchbaum



PI-2: Bäume 2

8

Einfügen in einen Suchbaum

- Algorithmus
 - Suche nach dem einzufügenden Wert, wobei selbst dann weitergesucht wird, wenn er gefunden wurde
 - Es sei denn, man will jeden Wert nur einmal im Suchbaum repräsentieren
 - Dadurch wird auf jeden Fall ein externer Knoten gefunden
 - Dieser wird durch ein neues Blatt mit dem einzufügenden Wert ersetzt

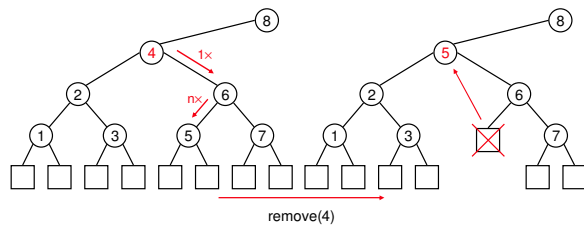
```
void insert(Comparable value)
{
    if (root == null)
        root = new Node(value);
    else
        insert2(root, value);
}

private void insert2(Node node,
                    Comparable value)
{
    if (value.compareTo(node.data) < 0)
        if (node.left == null)
            node.left = new Node(value);
        else
            insert2(node.left, value);
    else
        if (node.right == null)
            node.right = new Node(value);
        else
            insert2(node.right, value);
}
```

PI-2: Bäume 2

9

Löschen aus einem Suchbaum



PI-2: Bäume 2

10

Löschen aus einem Suchbaum

- Suche den zu löschenden Knoten
 - Merke unterwegs jeweils den Elternknoten
- Wenn er gefunden wurde
 - Falls einer seiner Nachfolger leer ist, ersetze ihn in seinem Elternknoten durch den jeweils anderen Nachfolger
 - Ansonsten suche den Knoten k mit dem nächst größeren Wert, überschreibe im zu löschenden Knoten den Wert mit dem Wert von k und lösche danach k

```
void remove(Comparable value)
{
    remove2(root, null, value);
}

private void remove2(Node node,
                    Node parent,
                    Comparable value)
{
    if (node != null)
    {
        int c = value.compareTo(node.data);
        if (c < 0)
            remove2(node.left, node, value);
        else if (c > 0)
            remove2(node.right, node, value);
        else
            removeNode(node, parent);
    }
}
```

PI-2: Bäume 2

11

Löschen aus einem Suchbaum

```
private void removeNode(Node node,
                       Node parent)
{
    if (node.right == null)
        replaceChild(parent, node,
                    node.left);
    else if (node.left == null)
        replaceChild(parent, node,
                    node.right);
    else if (node.right.left == null)
    {
        node.right.left = node.left;
        replaceChild(parent, node,
                    node.right);
    }
    else
    {
        Node pon = getParentOfNext(node);
        node.data = pon.left.data;
        pon.left = pon.left.right;
    }
}

private void replaceChild(Node node,
                        Node oldNode,
                        Node newNode)
{
    if (node == null)
        root = newNode;
    else if (node.left == oldNode)
        node.left = newNode;
    else
        node.right = newNode;
}

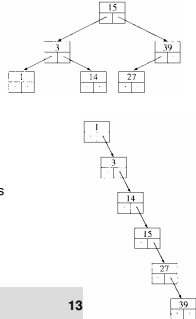
private Node getParentOfNext(Node node)
{
    node = node.right;
    while (node.left.left != null)
        node = node.left;
    return node;
}
```

PI-2: Bäume 2

12

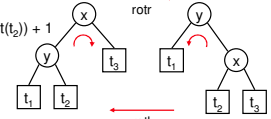
Balancierte Bäume

- Motivation**
 - Durch degenerierte Bäume kann die Suchdauer stark ansteigen
- AVL-Baum (Adelson-Velskij und Landis)**
- Definition**
 - Ein Baum ist *AVL-ausgeglichen* oder *höhenbalanciert*, wenn für jeden Knoten des Baumes gilt, dass sich die Höhe seines linken Teilbaums von der des rechten Teilbaums um maximal 1 unterscheidet
- Algorithmus-Idee**
 - Jedes Einfügen oder Löschen kann die Balanciertheit des Baumes zerstören.
 - Daher wird der Baum jeweils *rebalanciert*, sobald die Höhen des linken und rechten Teilbaums um zwei auseinander liegen



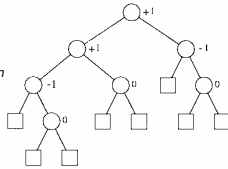
Balanciertheit, Höhe, Rotieren

- Baum**
 - Leerer Baum: \square
 - Sonst: (t_1, x, t_2)
- Balanciertheit**
 - $\text{heightbal}(\square) = \text{true}$
 - $\text{heightbal}((t_1, x, t_2)) = |\text{height}(t_1) - \text{height}(t_2)| \leq 1 \wedge \text{heightbal}(t_1) \wedge \text{heightbal}(t_2)$
- Höhe**
 - $\text{height}(\square) = 0$
 - $\text{height}((t_1, x, t_2)) = \max(\text{height}(t_1), \text{height}(t_2)) + 1$
- Baum rotieren**
 - $\text{rotr}((t_1, y, t_2), x, t_3) = (t_1, y, (t_2, x, t_3))$
 - $\text{rotl}((t_1, y, t_2), x, t_3) = ((t_1, y, t_2), x, t_3)$



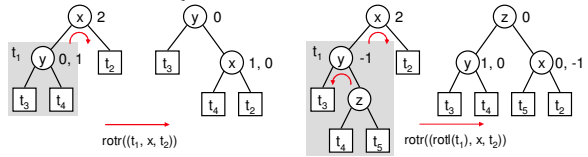
Einfügen in AVL-Baum

- Einfügen**
 - $\text{insert}(x, \square) = (\square, x, \square)$
 - $\text{insert}(x, (t_1, y, t_2)) = \begin{cases} \text{rebal}(\text{insert}(x, t_1), y, t_2) & \text{falls } x < y \\ \text{rebal}((t_1, y, \text{insert}(x, t_2))) & \text{sonst} \end{cases}$
- Neigung eines Knotens**
 - $\text{slope}(\square) = 0$
 - $\text{slope}((t_1, x, t_2)) = \text{height}(t_1) - \text{height}(t_2)$
 - Kann im Baum gespeichert werden und braucht nicht immer wieder neu ausgerechnet zu werden
- Rebalancieren**
 - $\text{rebal}(t) = \begin{cases} \text{shiftr}(t) & \text{falls } \text{slope}(t) = 2 \\ \text{shiftl}(t) & \text{falls } \text{slope}(t) = -2 \\ t & \text{sonst} \end{cases}$



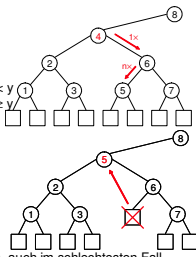
(Doppel-)Rotieren in AVL-Baum

- $\text{shiftr}((t_1, x, t_2)) = \begin{cases} \text{rotr}(\text{rotr}(t_1, x, t_2)) & \text{falls } \text{slope}(t_1) = -1 \\ \text{rotr}((t_1, x, t_2)) & \text{sonst} \end{cases}$
- $\text{shiftl}((t_1, x, t_2)) = \begin{cases} \text{rotl}((t_1, x, \text{rotr}(t_2))) & \text{falls } \text{slope}(t_2) = 1 \\ \text{rotl}((t_1, x, t_2)) & \text{sonst} \end{cases}$



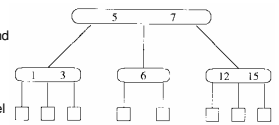
Löschen aus einem AVL-Baum

- Löschen**
 - $\text{remove}(x, \square) = \square$
 - $\text{remove}(x, (t_1, x, \square)) = t_1$
 - $\text{remove}(x, (\square, x, t_2)) = t_2$
 - $\text{remove}(x, (t_1, x, t_2)) = \text{rebal}(t_1, y, t)$ wobei $(y, t) = \text{leftest}(t_2)$
 - $\text{remove}(x, (t_1, y, t_2)) = \text{rebal}(\text{remove}(x, t_1), y, t_2)$ falls $x < y$
 - $\text{remove}(x, (t_1, y, t_2)) = \text{rebal}(t_1, y, \text{remove}(x, t_2))$ falls $x \geq y$
- Finden des nächsten Elements**
 - $\text{leftest}(\square, x, t) = (x, t)$
 - $\text{leftest}((t_1, x, t_2)) = (y, \text{rebal}(t, x, t_2))$ wobei $(y, t) = \text{leftest}(t_1)$
- Aufwand**
 - Rebalancieren eines Knotens passiert in konstanter Zeit
 - Bei Einfügen und Löschen wird auf dem Pfad von der Wurzel zu einem Knoten pro Knoten einmal rebalanciert
 - Also ist der Aufwand $O(\text{Höhe des Baumes})$
 - Da der Baum balanciert ist, ist der Aufwand also $O(\log n)$, auch im schlechtesten Fall



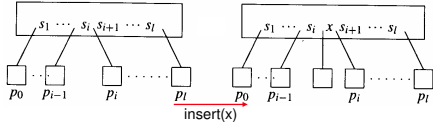
B-Bäume

- Motivation**
 - AVL-Bäume eignen sich nicht gut für die Verwaltung von Daten auf Massenspeichern (Festplatten)
 - Auf Daten auf Festplatten wird in Form von Sektoren zugegriffen
 - Der Wechsel zwischen Sektoren kostet Zeit (Kopfbewegung)
 - Daher: Daten linear, Baumstruktur als Index (ein Blattsuchbaum)
 - Index ist auch ein Baum, aber ein B-Baum, bei dem ein Knoten in einen Sektor passt
- B-Baum der Ordnung m**
 - Alle Blätter haben die gleiche Tiefe
 - Jeder Knoten mit Ausnahme der Wurzel und der Blätter hat wenigstens $\lceil m/2 \rceil$ Kinder
 - Die Wurzel hat mindestens 2 Kinder
 - Jeder Knoten hat höchstens m Kinder
 - Jeder Knoten mit i Kindern hat i-1 Schlüssel



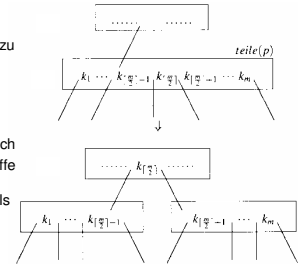
Einfügen und Suchen in B-Baum

- Suchen**
 - Binäre Suche in den Schlüsseln eines Knotens
 - Dann zum nächsten Knoten, so lange, bis Blatt erreicht
- Einfügen**
 - Suche nach Einfügeposition (ein Blatt)
 - Fall 1: Übergeordneter Knoten ist noch nicht voll
 - Füge Schlüssel in Knoten ein
 - Erzeuge neues Blatt und referenziere dieses aus dem Knoten



Einfügen in einen B-Baum

- Fall 2: Übergeordneter Knoten ist voll**
 - Füge Schlüssel in Knoten ein (ist zu groß)
 - Teile Knoten in der Mitte
 - Füge mittleren Knoten dem Elternknoten hinzu
 - Falls dieser zu voll ist, teile ihn auch
 - Falls die Wurzel geteilt wird, schaffe eine neue Wurzel und hänge die beiden Hälften der alten Wurzel als Nachfolger daran



Beispiel: Einfügen in B-Baum

