



# Suchen

## Such-Algorithmen

Gegeben sei eine Menge  $M$  von Objekten. Ein **Such-Algorithmus** sucht in  $M$  nach Mustern oder nach Objekten mit bestimmten Eigenschaften.

Beispiele:

- Suche von Adresse von Person in Telefonbuch
- Suche nach Webseite mit Google Search
- Suche nach Produkt auf Amazon
- Suche nach ähnlichen Mustern: Viren-Scanner
- Suche nach Mustern: Bilderkennung
- Suche nach Tumoren in medizinischen Bildern von Patienten

3

Notizen

---

---

---

---

---

---

---

---

---

---

## Lineare Suche

Gegeben sei Array  $A$  der Länge  $n$ , das Such-Schlüssel enthält.

- **einfachster Such-Algorithmus:** Durchlaufen des Feldes  $A$  bis gewünschter Schlüssel gefunden
- auch genannt: **Lineare Suche**
- **Algorithmus:**

**Input:** Array  $A[1..n]$  mit Schlüsseln,  $k$  gesuchter Schlüssel

**Output:** Index  $i$  mit  $A[i] = k$  (sonst 0)

**linearSearch**( $A, k$ ):

```
i = 1;
while ( (A[i] != k) && (i <= n) ) {
    i = i + 1;
}
if (i <= n) return i; // fündig geworden
else return 0; // nichts gefunden
```

- auch anwendbar für verkettete Listen

4

Notizen

---

---

---

---

---

---

---

---

---

---

# Lineare Suche: Komplexität

5	7	3	9	11	2
---	---	---	---	----	---

Laufzeit  $T(n)$  von `linearSearch`:

- **best-case**: sofort gefunden,  $T(n) = 1$ , d.h.  $T(n) = O(1)$
- **worst-case**: alles durchsuchen,  $T(n) = n$ , d.h.  $T(n) = O(n)$
- **im Mittel**: Annahme jede Anordnung der Such-Schlüssel ist gleich wahrscheinlich:

$$T(n) = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

d.h.  $T(n) = O(n)$

- einfacher Algorithmus, aber nicht besonders effizient

Notizen

---

---

---

---

---

---

---

---

---

---

5

# Programm heute

7 Fortgeschrittene Datenstrukturen

8 Such-Algorithmen

- Lineare Suche
- Binäre Suche**
- Binäre Suchbäume
- Balancierte Suchbäume

Notizen

---

---

---

---

---

---

---

---

---

---

6

# Binäre Suche

2	3	5	7	9	11
---	---	---	---	---	----

Gegeben sei Array A der Länge n, das Such-Schlüssel enthält.

- falls häufiger gesucht wird: Array A **vorsortieren!**  $O(n \log n)$
- Such-Algorithmus mittels **Divide & Conquer**  
Algorithmen-Muster:
  - **Divide**: vergleiche mittleres Element mit gesuchtem
  - **Rekursion**: falls kleiner, Rekursion auf linker Hälfte
  - **Rekursion**: falls grösser, Rekursion auf rechter Hälfte
  - **Conquer**: falls gleich, liefere Ergebnis

7

Notizen

---

---

---

---

---

---

---

---

---

---

## Binäre Suche: Algorithmus rekursiv

**Input:** Array A[1..n] sortierter Schlüssel, k gesuchter Schlüssel  
low, high: unterer/oberer Index von aktueller Array-Hälfte

**Output:** Index i mit  $A[i] = k$  (sonst 0)

**binarySearch**(A, k, low, high):

```
if (low > high) return 0; // nichts gefunden
middle = [(low + high) / 2];
if (A[middle] == k) return middle; // fündig geworden
if (A[middle] > k)
    return binarySearch(A, k, low, middle-1);
else
    return binarySearch(A, k, middle+1, high);
```

- **erster Aufruf** mit **binarySearch**(A, k, 1, n)

Notizen

---

---

---

---

---

---

---

---

---

---

8

# Binäre Suche: Algorithmus iterativ

**Input:** Array A[1..n] sortierter Schlüssel, k gesuchter Schlüssel

**Output:** Index i mit A[i] = k (sonst 0)

**binarySearchIterative(A, k):**

```
low = 1;
high = n;
while (low <= high) {
  middle = ⌊ (low + high) / 2 ⌋;
  if (A[middle] == k) return middle; // fündig geworden
  if (A[middle] > k)
    high = middle - 1;
  else
    low = middle + 1;
}
return 0; // nichts gefunden
```

Notizen

---

---

---

---

---

---

---

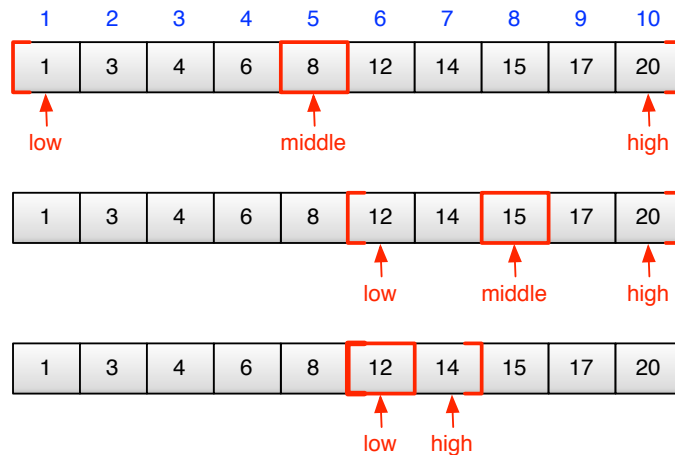
---

---

---

# Binäre Suche: Beispiel

- Gesucht: Schlüssel 12



Notizen

---

---

---

---

---

---

---

---

---

---

# Binäre Suche: Komplexität

- Komplexität:  $O(\log n)$ 
  - errechnet z.B. via Rekursionsbaum wie bei MergeSort
- Beispiel-Laufzeiten:

Algorithmus	$n = 10$	$n = 1000$	$n = 10^6$
Lineare Suche ( $n/2$ )	$\approx 5$	$\approx 500$	$\approx 500.000$
Binäre Suche ( $\log_2 n$ )	$\approx 3.3$	$\approx 9.9$	$\approx 19.9$

- sehr effizienter Such-Algorithmus!
- falls sich Daten oft ändern, muss jeweils neu sortiert werden
  - besser: Suchbäume

Notizen

---

---

---

---

---

---

---

---

---

---

# Programm heute

## 7 Fortgeschrittene Datenstrukturen

## 8 Such-Algorithmen

- Lineare Suche
- Binäre Suche
- Binäre Suchbäume**
- Balancierte Suchbäume

Notizen

---

---

---

---

---

---

---

---

---

---

# Binärer Suchbaum

## Definition binärer Suchbaum

Sei  $G = (V, E)$  ein Binärbaum mit Wurzel  $w \in V$ . Jeder Knoten  $v \in V$  sei mit einem Wert  $key(v)$  verknüpft, die Werte seien durch  $\leq, \geq$  geordnet.

$G$  heißt binärer Suchbaum, falls für alle inneren Knoten  $v \in V$  gilt

- für alle Knoten  $x$  im linken Teilbaum  $v.left$  gilt

$$key(x) \leq key(v)$$

- für alle Knoten  $y$  im rechten Teilbaum  $v.right$  gilt

$$key(y) \geq key(v)$$

Notizen

---

---

---

---

---

---

---

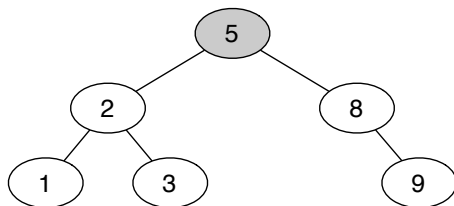
---

---

---

13

## Binärer Suchbaum: Beispiel



- binärer Baum muss nicht vollständig sein!
- Repräsentation üblicherweise mit verketteter Liste (geht aber auch als sequentielle Liste)

Notizen

---

---

---

---

---

---

---

---

---

---

14

## Binärer Suchbaum: Operationen

Operationen auf binärem Suchbaum:

- **Suchen:** finde Element mit Schlüssel  $k$
- **Minimum/Maximum:** finde Element mit minimalem/  
maximalem Schlüssel
- **Einfügen:** füge Element zum Suchbaum hinzu
- **Löschen:** entferne Element aus Suchbaum

Notizen

---

---

---

---

---

---

---

---

---

---

15

## Binärer Suchbaum: Suchen (rekursiv)

**Input:** Knoten  $v$ , dessen Teilbaum untersucht werden soll,  
 $k$  gesuchter Schlüssel

**Output:** Knoten mit gesuchtem Schlüssel,  
null falls nicht gefunden

**search**( $v, k$ ):

```
if ( $v == \text{null}$ ) return null; // hier gibt es nichts!  
if ( $\text{key}(v) == k$ ) return  $v$ ; // fündig geworden!  
if ( $k < \text{key}(v)$ )  
    search( $v.\text{left}, k$ );  
else  
    search( $v.\text{right}, k$ );
```

- **erster Aufruf** mit **search**( $w, k$ )
- falls  $v$  kein linkes/rechtes Kind hat, ist das durch null markiert

Notizen

---

---

---

---

---

---

---

---

---

---

16



## Binärer Suchbaum: Suchen (iterativ)

**Input:** Knoten  $v$ , dessen Teilbaum untersucht werden soll,  
 $k$  gesuchter Schlüssel

**Output:** Knoten mit gesuchtem Schlüssel,  
null falls nicht gefunden

**searchIterative**( $v, k$ ):

```
while ( (v != null) && (key(v) != k) ) {  
  if (k < key(v))  
    v = v.left;  
  else  
    v = v.right;  
}  
return v;
```

- Komplexität:  $O(h)$ , wobei  $h$  Höhe von Suchbaum

Notizen

---

---

---

---

---

---

---

---

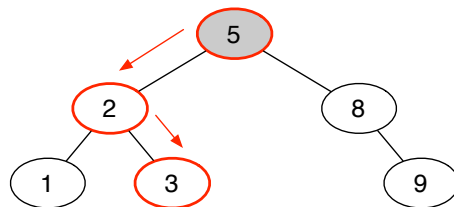
---

---

17

## Binärer Suchbaum: Suchen

- Beispiel: suche Schlüssel 3



Notizen

---

---

---

---

---

---

---

---

---

---

18

## Binärer Suchbaum: Minimum/Maximum

**Input:** Wurzel  $v$  des zu durchsuchenden Baumes

**Output:** Knoten mit **minimalem** Schlüssel

```
minimum(v):  
  while (v.left != null)  
    v = v.left;  
  return v;
```

**Input:** Wurzel  $v$  des zu durchsuchenden Baumes

**Output:** Knoten mit **maximalem** Schlüssel

```
maximum(v):  
  while (v.right != null)  
    v = v.right;  
  return v;
```

Notizen

---

---

---

---

---

---

---

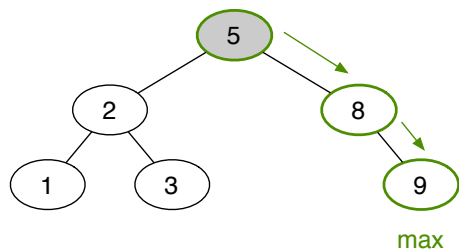
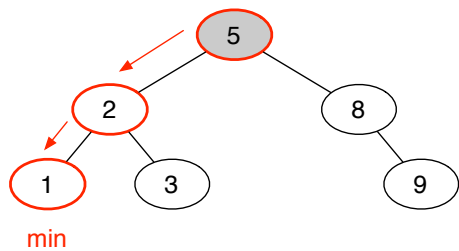
---

---

---

19

## Binärer Suchbaum: Minimum/Maximum



- Komplexität:  $O(h)$ , wobei  $h$  Höhe von Suchbaum

Notizen

---

---

---

---

---

---

---

---

---

---

20

## Binärer Suchbaum: Einfügen

**Input:** Wurzel  $v$  des Baumes,  $x$  einzufügendes Element

**insert**( $v, x$ ):

```
if ( $v == \text{null}$ ) { // Baum leer
   $v = x$ ; return;
}
while ( $v \neq \text{null}$ ) {
  hilfsKnoten =  $v$ ;
  if ( $\text{key}(x) < \text{key}(v)$ )
     $v = v.\text{left}$ ;
  else
     $v = v.\text{right}$ ;
}
 $x.\text{vater} = \text{hilfsKnoten}$ ;
if ( $\text{key}(x) < \text{key}(\text{hilfsKnoten})$ )
   $\text{hilfsKnoten}.\text{left} = x$ ;
else
   $\text{hilfsKnoten}.\text{right} = x$ ;
```

Notizen

---

---

---

---

---

---

---

---

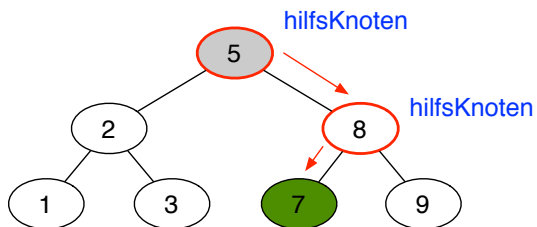
---

---

21

## Binärer Suchbaum: Einfügen

- Einfügen von Knoten mit Schlüssel 7:



- Komplexität:  $O(h)$ , wobei  $h$  Höhe von Suchbaum

Notizen

---

---

---

---

---

---

---

---

---

---

22

# Binärer Suchbaum: Löschen

- Löschen von Knoten  $x$  in Suchbaum ist etwas komplizierter
- Drei Fälle:
  - ①  $x$  ist Blatt: einfach entfernen
  - ②  $x$  hat nur ein Kind: setze Kind an Stelle von  $x$
  - ③  $x$  hat zwei Kinder: setze **minimales** Element von **rechtem** Teilbaum an Stelle von  $x$   
(alternativ: maximales Element von linkem Teilbaum)
- Komplexität:  $O(h)$ , wobei  $h$  Höhe von Suchbaum

Notizen

---

---

---

---

---

---

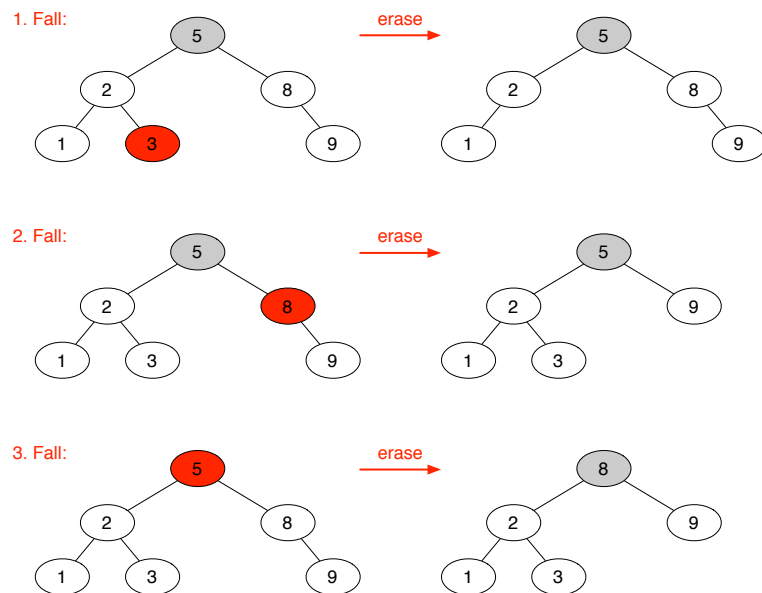
---

---

---

---

# Binärer Suchbaum: Löschen



Notizen

---

---

---

---

---

---

---

---

---

---

## Binärer Suchbaum: Löschen

**Input:** Wurzel  $v$  des Baumes,  $x$  zu löschendes Element

**erase**( $v, x$ ):

```
if (x ist Blatt) { // 1. Fall
  if (x ist linkes Kind) x.vater.left = null;
  else x.vater.right = null;
} else { // 2. Fall
  if (x.left == null) {
    if (x ist linkes Kind) x.vater.left = x.right;
    else x.vater.right = x.right;
  } else {
    if (x.right == null) {
      if (x ist linkes Kind) x.vater.left = x.left;
      else x.vater.right = x.left;
    } else { // 3. Fall
      kind = minimum(x.right);
      ersetze x durch kind;
    }
  }
}
```

Notizen

---

---

---

---

---

---

---

---

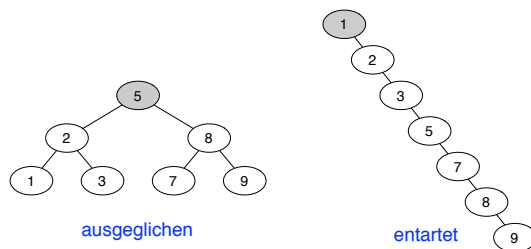
---

---

25

## Binärer Suchbaum: Effizienz

- Suchbäume mit  $n$  Knoten sind sehr effizient
  - aber nur wenn sie ausgeglichen ("balanciert") sind!
- best-case Komplexität:  $O(\log n)$
- worst-case Komplexität:  $O(n)$



- Ausweg: automatisch balancierte Suchbäume (z.B. AVL Bäume, Rot-Schwarz Bäume, B-Bäume)

Notizen

---

---

---

---

---

---

---

---

---

---

26

# Programm heute

## 7 Fortgeschrittene Datenstrukturen

## 8 Such-Algorithmen

- Lineare Suche
- Binäre Suche
- Binäre Suchbäume
- Balancierte Suchbäume

Notizen

---

---

---

---

---

---

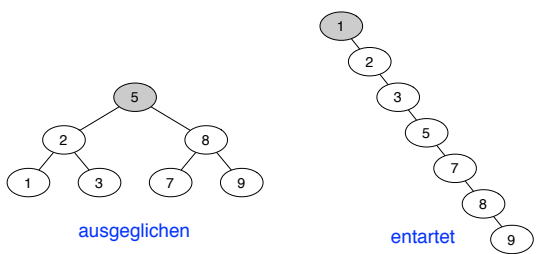
---

---

---

---

# Entartete Suchbäume



- Wie können Suchbäume entarten?
  - Beispiel: einfügen aus sortierter Liste
- **Erwünscht:** Suchbäume, die immer **ausgeglichen** (balanciert) bleiben
  - **AVL-Bäume**, Rot-Schwarz-Bäume, B-Bäume etc.

Notizen

---

---

---

---

---

---

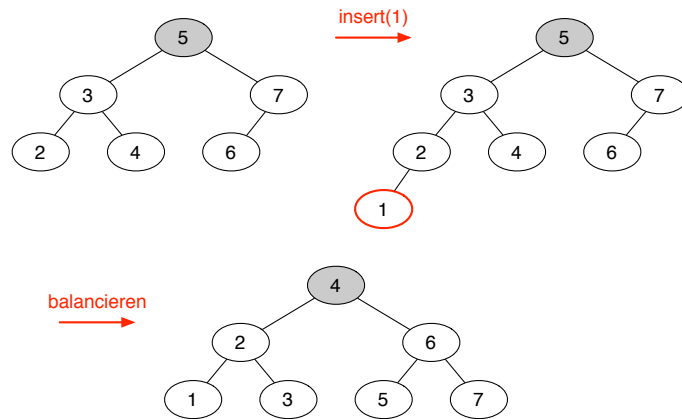
---

---

---

---

## Beispiel: Balancieren von Suchbaum



- hier müssen zum Balancieren **alle** Knoten bewegt werden → Effizienz-Problem

Notizen

---

---

---

---

---

---

---

---

---

---

29

## Ansätze für balancierte Suchbäume

- Binärbaum und gleichzeitig balanciert ist ineffizient
- **Idee:** Aufweichen eines der beiden Kriterien!
- **Abschwächung** des Kriteriums **balanciert**
  - Beispiel: AVL-Bäume
- **Abschwächung** des Kriteriums **Binärbaum**
  - Mehrweg-Bäume, Beispiel: B-Bäume
  - mehrere Verzweigungen kodiert als Binärbaum, Beispiel: Rot-Schwarz-Bäume

Notizen

---

---

---

---

---

---

---

---

---

---

30

# Definition AVL-Baum

## Definition AVL-Baum

Ein binärer Suchbaum  $G = (V, E)$  mit Wurzel  $w \in V$  heißt **AVL-Baum**, falls er die **AVL-Bedingung** erfüllt:

- für jeden inneren Knoten  $v \in V$  gilt: Höhe von linkem und rechtem Teilbaum von  $v$  unterscheidet sich maximal um 1.
- benannt nach G.M. Adelson-Velskii und E.M. Landis (russische Mathematiker)
- AVL-Bedingung nur für Wurzel  $w$  ist nicht ausreichend
  - beide Teilbäume der Wurzel können entartet sein

Notizen

---

---

---

---

---

---

---

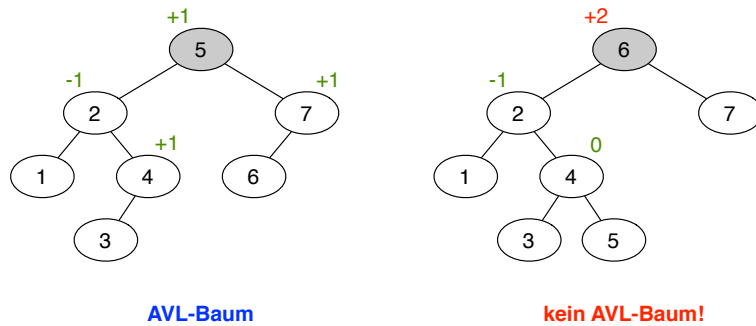
---

---

---

31

## AVL-Baum: Beispiel



- linkes Beispiel: AVL-Bedingung überall erfüllt
- rechtes Beispiel: AVL-Bedingung in Wurzel verletzt

Notizen

---

---

---

---

---

---

---

---

---

---

32



## AVL Baum: Operationen

- Operationen `search`, `minimum`, `maximum` unverändert von binärem Suchbaum
- Operationen `insert`, `erase` müssen verändert werden, damit die AVL-Bedingung erhalten wird

Notizen

---

---

---

---

---

---

---

---

---

---

33

## AVL-Baum: Einfügen

Einfüge-Operation bei AVL-Baum:

- `insert` wie in binärem Suchbaum
- nun kann AVL-Bedingung verletzt sein:
  - $balance = height(left) - height(right)$
  - AVL-Bedingung:  $balance \in \{-1, 0, +1\}$
  - nach `insert`:  $balance \in \{-2, -1, 0, 1, +2\}$
- reparieren der AVL-Bedingung mittels `Rotation` und `Doppelrotation`

Notizen

---

---

---

---

---

---

---

---

---

---

34

# Einfügen / Verletzung AVL-Bedingung

Notizen

Fallunterscheidung Verletzung AVL-Bedingung bei Einfügen:

- 1 Einfügen in linken Teilbaum des linken Kindes
- 2 Einfügen in rechten Teilbaum des linken Kindes
- 3 Einfügen in linken Teilbaum des rechten Kindes
- 4 Einfügen in rechten Teilbaum des rechten Kindes

1 und 4 sind symmetrische Fälle, sowie 2 und 3

---

---

---

---

---

---

---

---

---

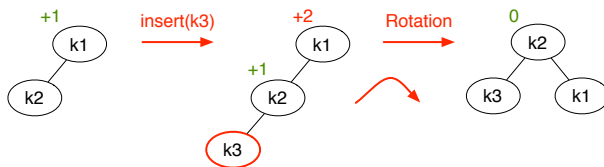
---

35

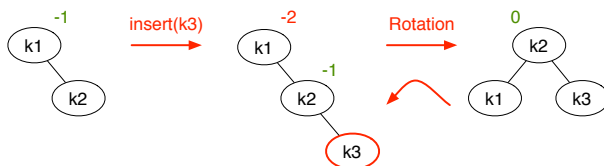
# AVL-Baum: Rotation

Notizen

1 Einfügen in linken Teilbaum des linken Kindes:



4 Einfügen in rechten Teilbaum des rechten Kindes:



---

---

---

---

---

---

---

---

---

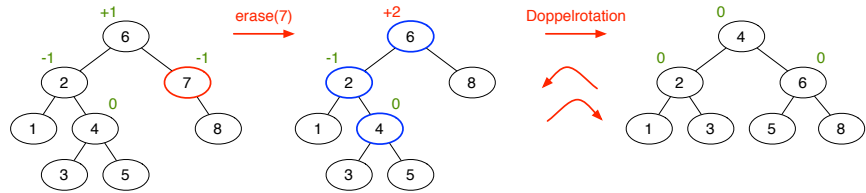
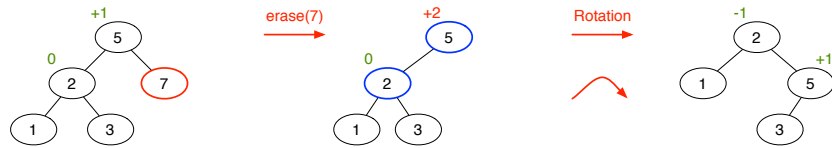
---

36





# AVL-Baum: Beispiel-Sequenz III



Notizen

---

---

---

---

---

---

---

---

---

---

---

---

# Zusammenfassung

## 7 Fortgeschrittene Datenstrukturen

## 8 Such-Algorithmen

- Lineare Suche
- Binäre Suche
- Binäre Suchbäume
- Balancierte Suchbäume

Notizen

---

---

---

---

---

---

---

---

---

---

---

---