

# Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures  
Technische Universität München



# Programm heute

7 Fortgeschrittene Datenstrukturen

**8 Such-Algorithmen**

Lineare Suche

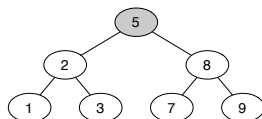
Binäre Suche

Binäre Suchbäume

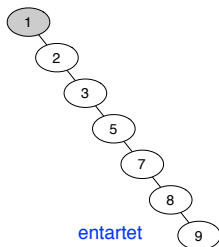
**Balancierte Suchbäume**

Suchen in Zeichenketten

# Entartete Suchbäume



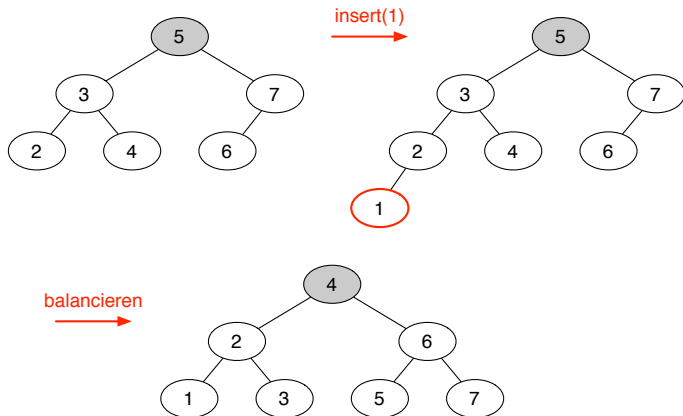
ausgeglichen



entartet

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

## Beispiel: Balancieren von Suchbaum



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

# Ansätze für balanciert 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

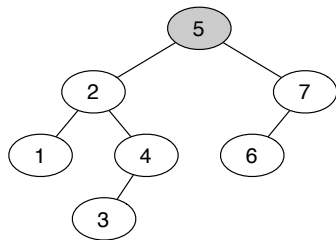
# Definition AVL-Baum

## Definition AVL-Baum

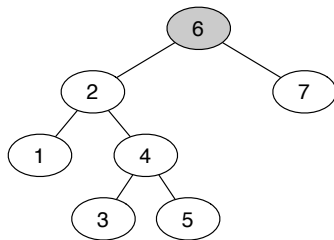
Ein binärer Suchbaum  $G = (V, E)$  mit Wurzel  $w \in V$  heißt **AVL-Baum**, falls er die **AVL-Bedingung** erfüllt:

- 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

# AVL-Baum: Beispiel



AVL-Baum



kein AVL-Baum!

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

# AVL Baum: Operationen

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



# AVL-Baum: Einfügen

Einfüge-Operation bei AVL-Baum:

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

## Einfügen / Verletzung AVL-Bedingung

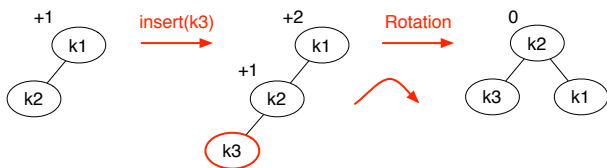
Fallunterscheidung Verletzung AVL-Bedingung bei Einfügen:

- 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

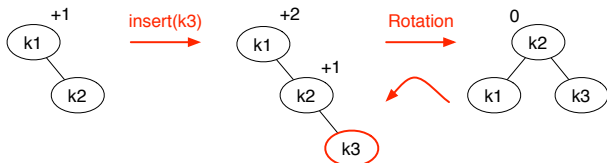
1 und 4 sind symmetrische Fälle, sowie 2 und 3

# AVL-Baum: Rotation

- ① Einfügen in linken Teilbaum des linken Kindes:

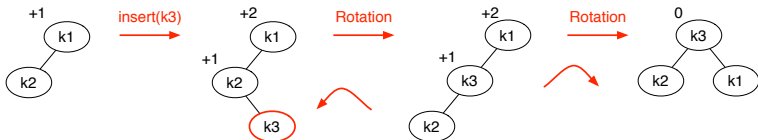


- ④ Einfügen in rechten Teilbaum des rechten Kindes:

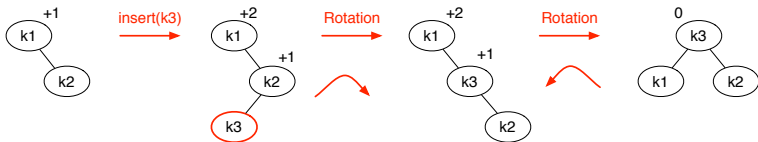


# AVL-Baum: Doppelrotation

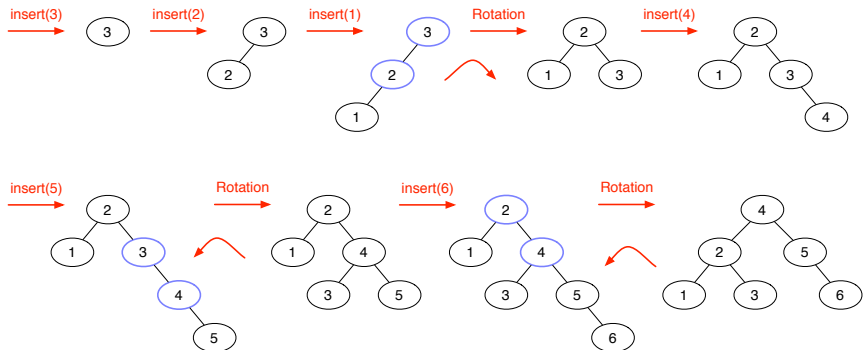
## 2 Einfügen in rechten Teilbaum des linken Kindes:



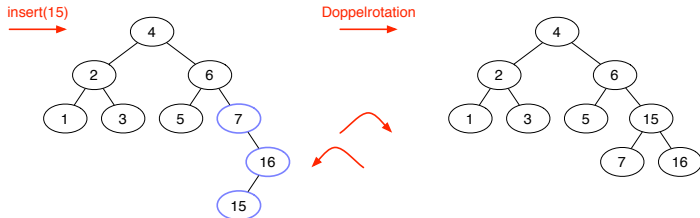
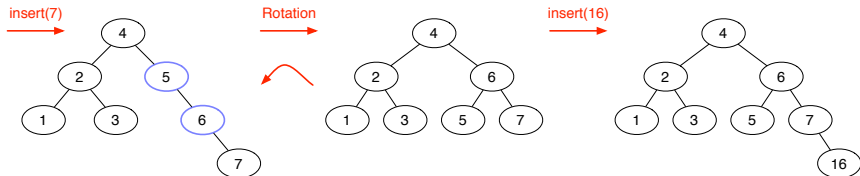
## 3 Einfügen in linken Teilbaum des rechten Kindes:



# AVL-Baum: Beispiel-Sequenz I



# AVL-Baum: Beispiel-Sequenz II



# AVL-Baum: Löschen

Löschen-Operation bei AVL-Baum:

- `erase` wie in binärem Suchbaum
- Verletzung der AVL-Bedingung in Teilbäumen durch Rotationen reparieren
- bei jedem Eltern-Knoten wieder AVL-Bedingungen reparieren, bis hin zur Wurzel

# Programm heute

7 Fortgeschrittene Datenstrukturen

**8 Such-Algorithmen**

Lineare Suche

Binäre Suche

Binäre Suchbäume

Balancierte Suchbäume

Suchen in Zeichenketten

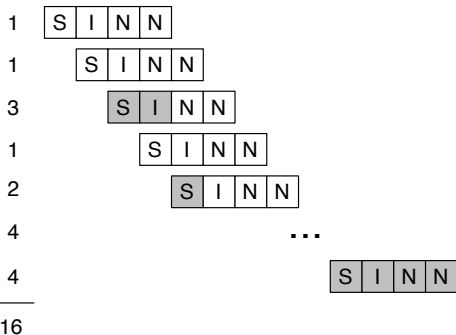


# Suchen in Zeichenketten

- **Problem:** find Teilwort in (langem) anderen Wort
- auch genannt: **String-Matching**
- Beispiele:
  - Suche Text in Textverarbeitung / Web-Browser
  - Suche Text in Dateien auf Festplatte (z.B. Spotlight, Windows Search)
  - Suche Text im Internet (z.B. Google)
- **Maß der Effizienz:** Anzahl der Vergleiche zwischen Buchstaben der Worte

# Brute-Force Suche

D A S I S T E I N S I N N L O S E R T E X T



# Notationen

- Zu durchsuchender Text:
  - $text[0..n - 1]$
  - Länge  $n$
- gesuchtes Muster = Pattern:
  - $pat[0..m - 1]$
  - Länge  $m$
- **Problem:** finde Position  $i$ , so daß  $pat == text[i..i + m - 1]$

# Brute-Force Algorithmus

**Input:** zu durchsuchender Text *text* Länge *n*,  
gesuchtes Muster *pat* Länge *m*

**Output:** Index *i* von Match (oder -1 falls nicht gefunden)

**bruteForceSearch**(*text*, *pat*):

```
for i = 0 to n - m {  
    j = 0;  
    while ( (j < m) && (pat[j] == text[i + j] ) )  
        j = j + 1;  
    if (j ≥ m) return i; // fündig geworden  
}  
return -1; // nichts gefunden
```

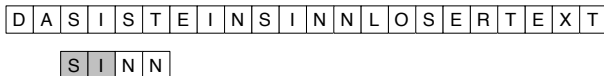
- Komplexität:  $O((n - m)m) = O(nm)$

# Knuth-Morris-Pratt Algorithmus

## Knuth-Morris-Pratt Algorithmus (kurz: KMP)

- **Idee:** verbessere Brute-Force Algorithmus durch Ausnutzung der bereits gelesenen Information bei einem Mismatch
- Mismatch an Stelle  $j$  von  $pat$  impliziert

$$pat[0..j - 1] == text[i..i + j - 1]$$



- **Vorverarbeitungsschritt:** analysiere vor Suche das Muster  $pat$ , speichere mögliche Überspringungen in **Feld  $next$**

# KMP Algorithmus: Beispiel

A	B	C	A	B	A	B	A	B	C	A	A	B	A	B
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	B	C	A	A	B
---	---	---	---	---	---

j = 0 1 2 3 4

A	B	C	A	A	B
---	---	---	---	---	---

j = 0 1 2

A	B	C	A	A	B
---	---	---	---	---	---

j = 0 1 2

A	B	C	A	A	B
---	---	---	---	---	---

j = 0 1 2 3 4 5

<i>j</i>	0	1	2	3	4	5
<i>next</i> [ <i>j</i> ]	-1	0	0	0	1	1

# KMP Algorithmus

**Input:** zu durchsuchender Text *text* Länge *n*,  
gesuchtes Muster *pat* Länge *m*

**Output:** Index *i* von Match (oder -1 falls nicht gefunden)

**KMPSearch**(*text*, *pat*):

```
j = 0;
```

```
for i = 0 to n - 1 {
```

```
    while ( (j ≥ 0) && (pat[j] != text[i]) )
```

```
        j = next[j];
```

```
    j = j + 1;
```

```
    if (j == m)
```

```
        return i - m + 1; // fündig geworden
```

```
}
```

```
return -1; // war wohl nix
```

# KMP Algorithmus: next Tabelle I

$j$		$next[j]$												
1	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table> <table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table>	A	B	C	A	A	B	A	B	C	A	A	B	0
A	B	C	A	A	B									
A	B	C	A	A	B									
2	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table> <table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table>	A	B	C	A	A	B	A	B	C	A	A	B	0
A	B	C	A	A	B									
A	B	C	A	A	B									
3	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table> <table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table>	A	B	C	A	A	B	A	B	C	A	A	B	0
A	B	C	A	A	B									
A	B	C	A	A	B									
4	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table> <table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table>	A	B	C	A	A	B	A	B	C	A	A	B	1
A	B	C	A	A	B									
A	B	C	A	A	B									
5	<table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table> <table border="1"><tr><td>A</td><td>B</td><td>C</td><td>A</td><td>A</td><td>B</td></tr></table>	A	B	C	A	A	B	A	B	C	A	A	B	1
A	B	C	A	A	B									
A	B	C	A	A	B									



## KMP Algorithmus: next Tabelle II

**Input:** Muster  $pat$  Länge  $m$

**initNext**( $pat$ ):

$next[0] = -1; next[1] = 0;$

$pos = 2; cnd = 0;$

**while** ( $pos < m$ ) {

**if** ( $pat[pos - 1] == pat[cnd]$ ) {

$cnd = cnd + 1;$

$next[pos] = cnd;$

$pos = pos + 1;$

**else** {

**if** ( $cnd > 0$ )  $cnd = next[cnd];$

**else** {

$next[pos] = 0;$

$pos = pos + 1;$

    }

  }

}

# KMP Algorithmus: Komplexität

Komplexität von KMP Algorithmus:

- KMPSearch: innere Schleife maximal  $2n$  Durchläufe:  $O(n)$
- initNext: innere Schleife maximal  $m$  Durchläufe:  $O(m)$
- insgesamt:  $O(n + m)$
  
- Platzbedarf:  $O(m)$

# Ausblick: Suchen in Zeichenketten

- Brute-Force Algorithmus
  - Komplexität:  $O(mn)$
- Knuth-Morris-Pratt Algorithmus
  - Komplexität:  $O(m + n)$
- Rabin-Karp Algorithmus: Suchen mit Hash-Funktion
  - Komplexität im Mittel:  $O(m + n)$
  - Komplexität worst-case:  $O(mn)$
- Boyer-Moore Algorithmus: Suchen rückwärts
  - Komplexität:  $O(n)$
  - Komplexität best-case:  $O(n/m)$
- Reguläre Ausdrücke mit endlichen Automaten
- Suche nach ähnlichen Zeichenketten

# Zusammenfassung

## 7 Fortgeschrittene Datenstrukturen

## 8 Such-Algorithmen

Lineare Suche

Binäre Suche

Binäre Suchbäume

Balancierte Suchbäume

Suchen in Zeichenketten