Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Programm

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen in Zeichenketten

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

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

Lineare Suche: Komplexität

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

Programm

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen in Zeichenketten

Binäre Suche

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

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)

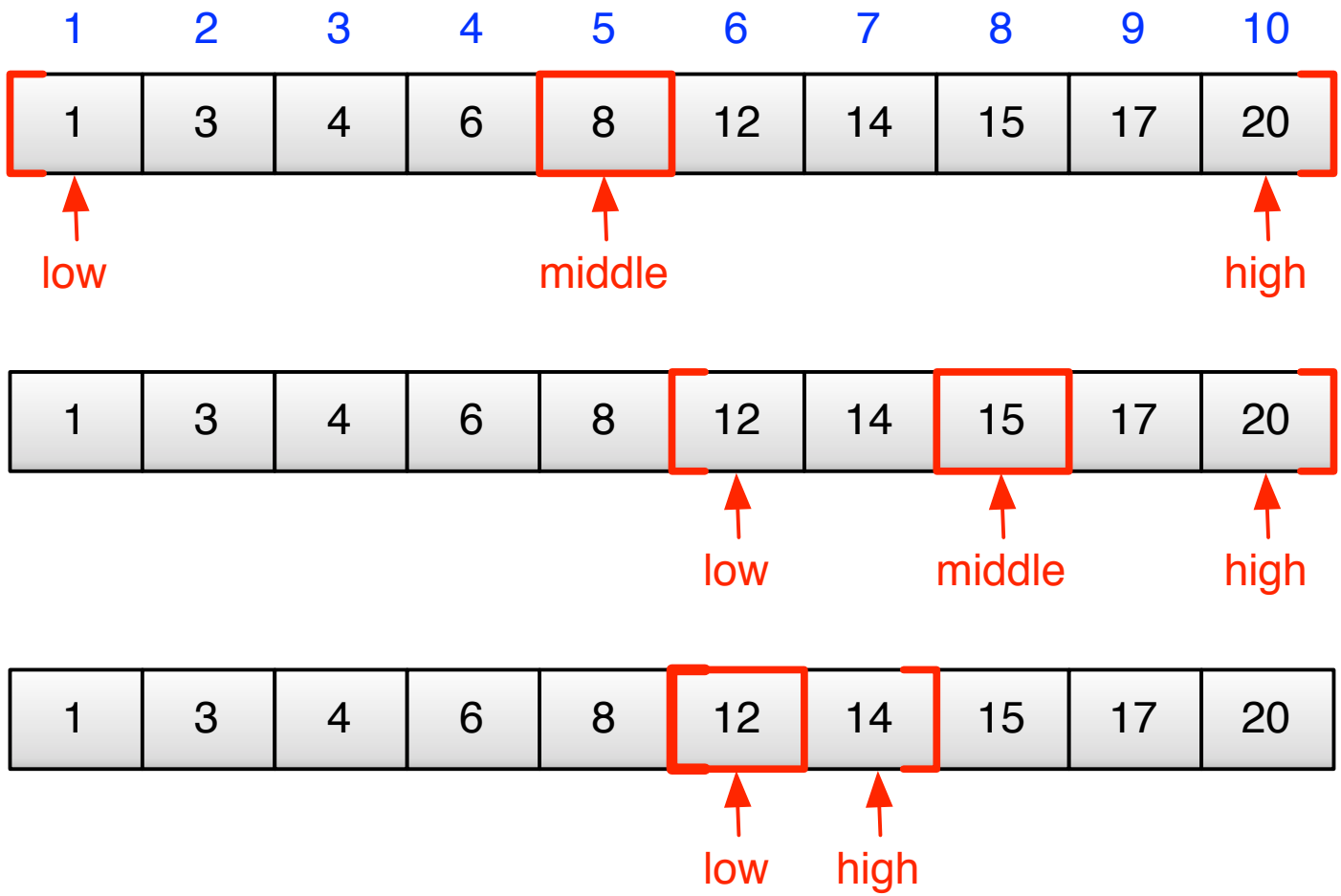
Binäre Suche: Algorithmus iterativ

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
```

Binäre Suche: Beispiel

- Gesucht: Schlüssel 12



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 $(n/2)$	≈ 5	≈ 500	≈ 500.000
Binäre Suche $(\log_2 n)$	≈ 3.3	≈ 9.9	≈ 19.9

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

Programm

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen in Zeichenketten

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

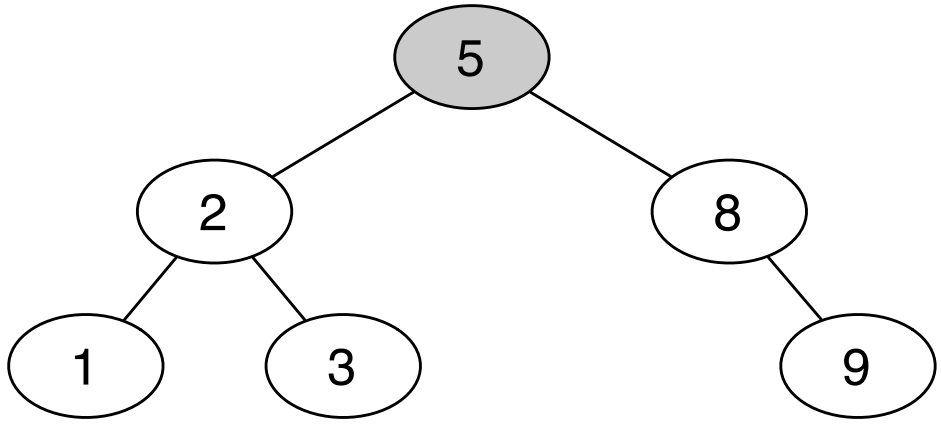
- für alle Knoten x im linken Teilbaum $v.left$ gilt

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

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

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

Binärer Suchbaum: Beispiel



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

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

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 == NULL) return NULL; // hier gibt es nichts!
    if (key(v) == k) return v; // fündig geworden!
    if (k < key(v))
       search(v.left, k);
    else
       search(v.right, k);
```

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

Binärer Suchbaum: Suchen (iterativ)

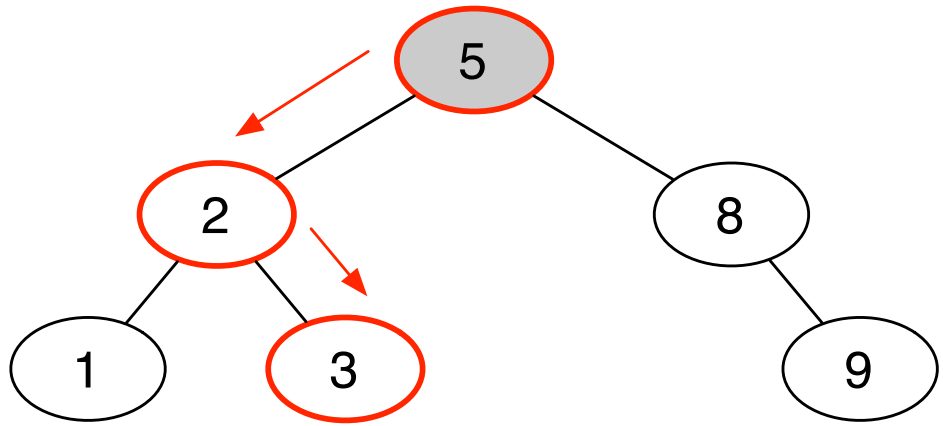
Input: Knoten v, dessen Teilbaum untersucht werden soll,
k gesuchter Schlüssel
Output: Knoten mit gesuchtem Schlüssel,
NULL falls nicht gefunden

```
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

Binärer Suchbaum: Suchen

- Beispiel: suche Schlüssel 3

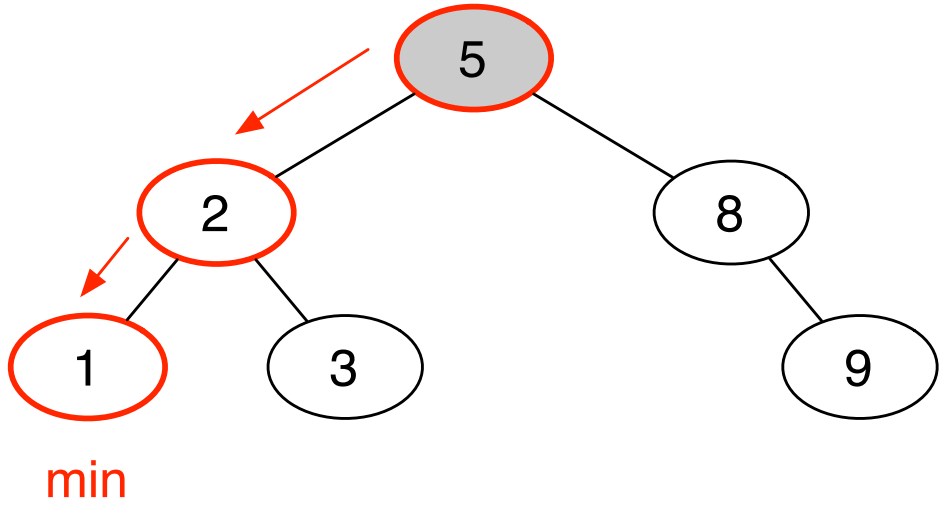


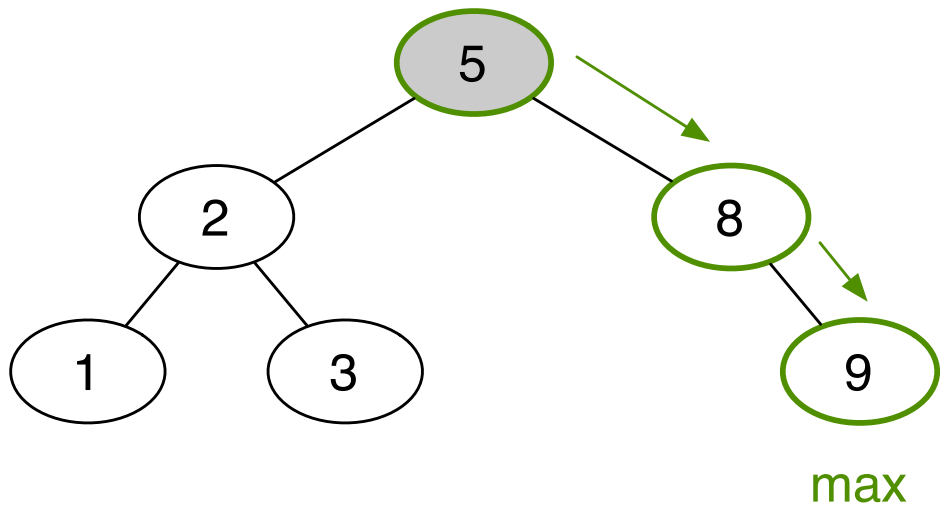
Binärer Suchbaum: Minimum/Maximum

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;

Binärer Suchbaum: Minimum/Maximum





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

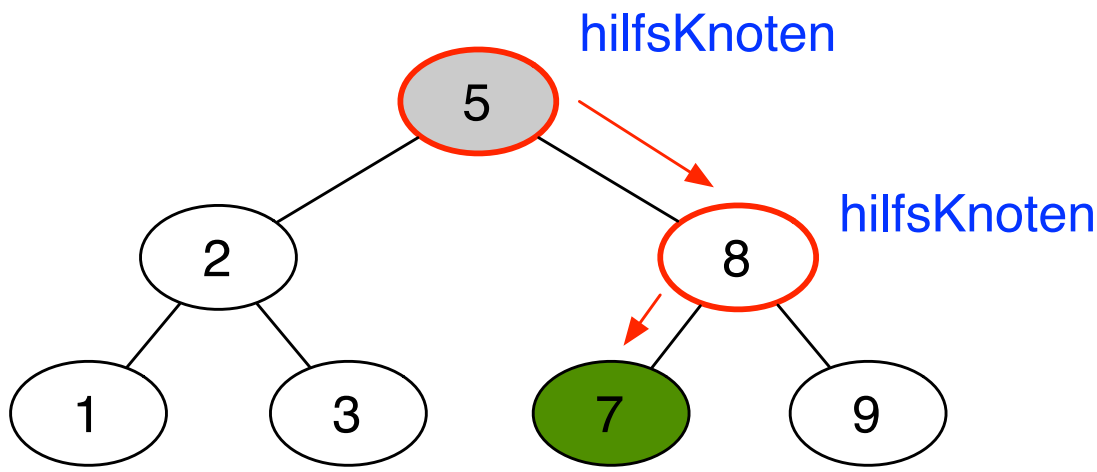
Binärer Suchbaum: Einfügen

Input: Wurzel v des Baumes, x einzufügendes Element

```
insert(v, x):
    if (v == NULL) { // Baum leer
        v = x; return;
    }
    while (v != NULL) {
        hilfsKnoten = v;
        if (key(x) < key(v))
            v = v.left;
        else
            v = v.right;
    }
    x.vater = hilfsKnoten;
    if (key(x) < key(hilfsKnoten))
        hilfsKnoten.left = x;
    else
        hilfsKnoten.right = x;
```

Binärer Suchbaum: Einfügen

- Einfügen von Knoten mit Schlüssel 7:



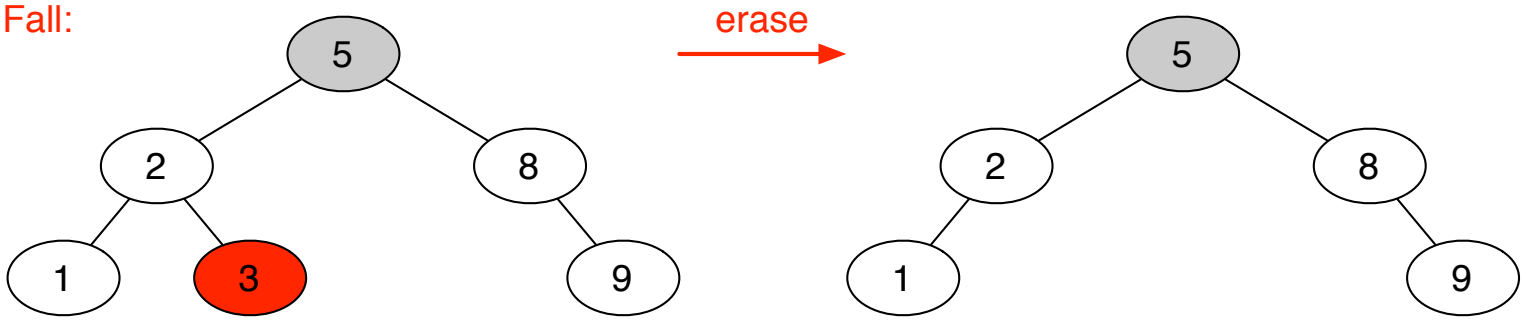
- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärer Suchbaum: Löschen

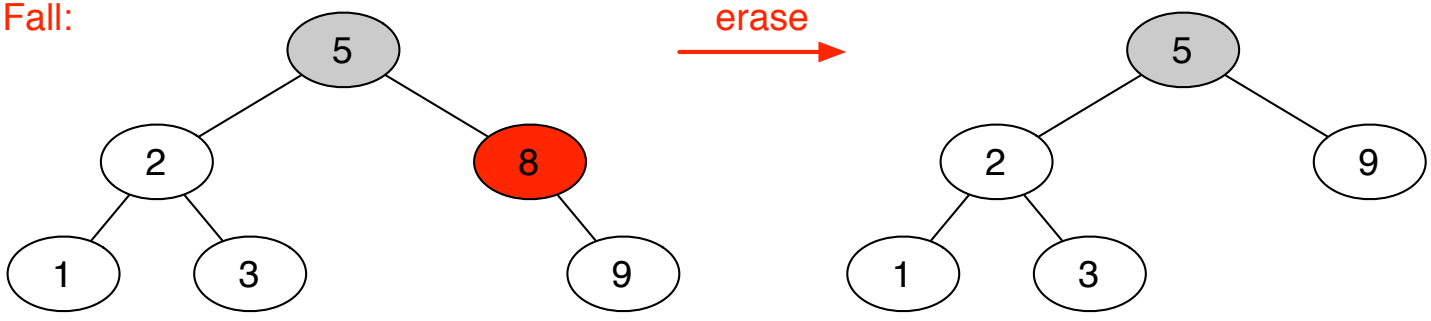
- Löschen von Knoten x in Suchbaum ist etwas komplizierter
- Drei Fälle:
  1. x ist Blatt: einfach entfernen
  2. x hat nur ein Kind: setze Kind an Stelle von x
  3. x hat zwei Kinder: setze minimales Element von rechtem Teilbaum an Stelle von x
     (alternativ: maximales Element von linkem Teilbaum)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

Binärerer Suchbaum: Löschen

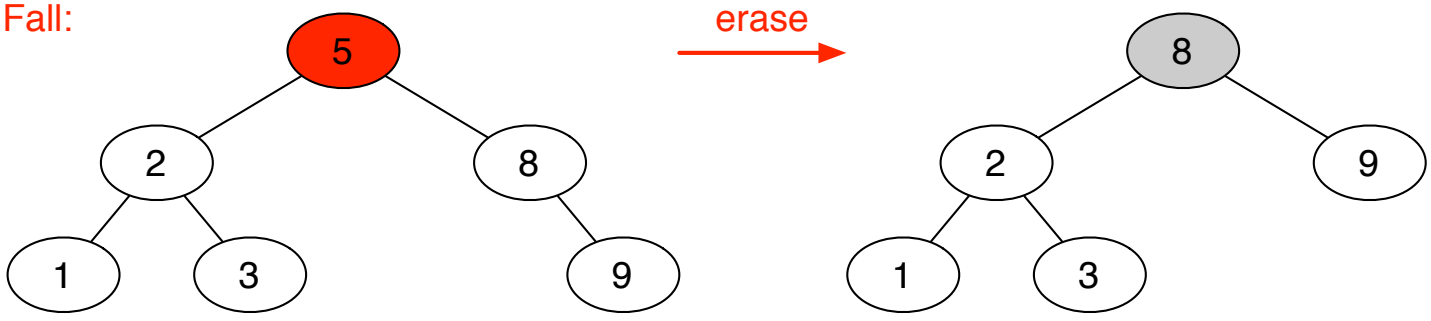
1. Fall:



2. Fall:



3. Fall:



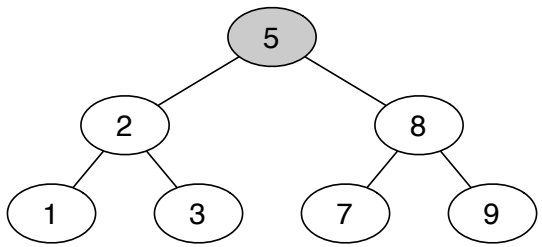
Binärer Suchbaum: Löschen

Input: Wurzel v des Baumes, x zu löschendes Element

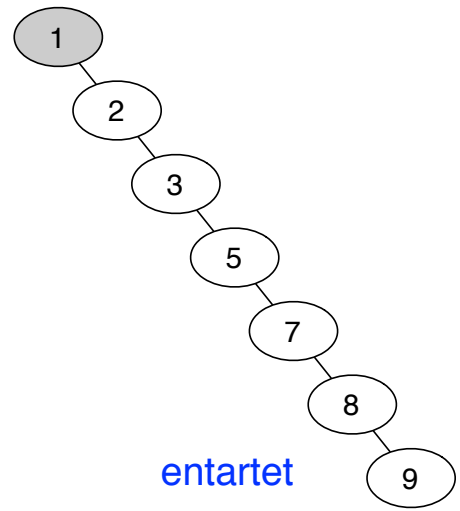
```
erase(v, x):
    if (x ist Blatt) { // 1. Fall
        if (x ist linkes Kind) x.vater.left = NULL;
        else x.vater.right = NULL;
    } else { // 2. Fall
        if (x.left == NULL) {
            if (x ist linkes Kind) x.vater.left = x.right;
            else x.vater.right = x.right;
        } 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;
            }
        }
    }
```

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: $O(n)$



ausgeglichen



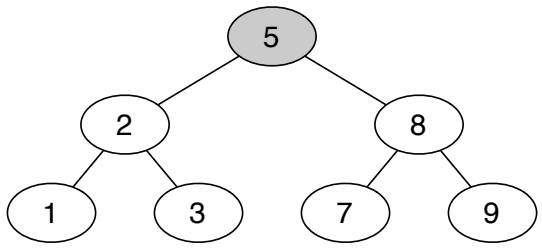
entartet

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

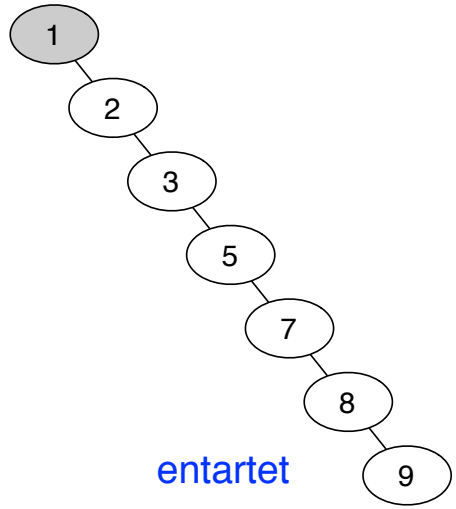
Programm

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen in Zeichenketten

Entartete Suchbäume



ausgeglichen

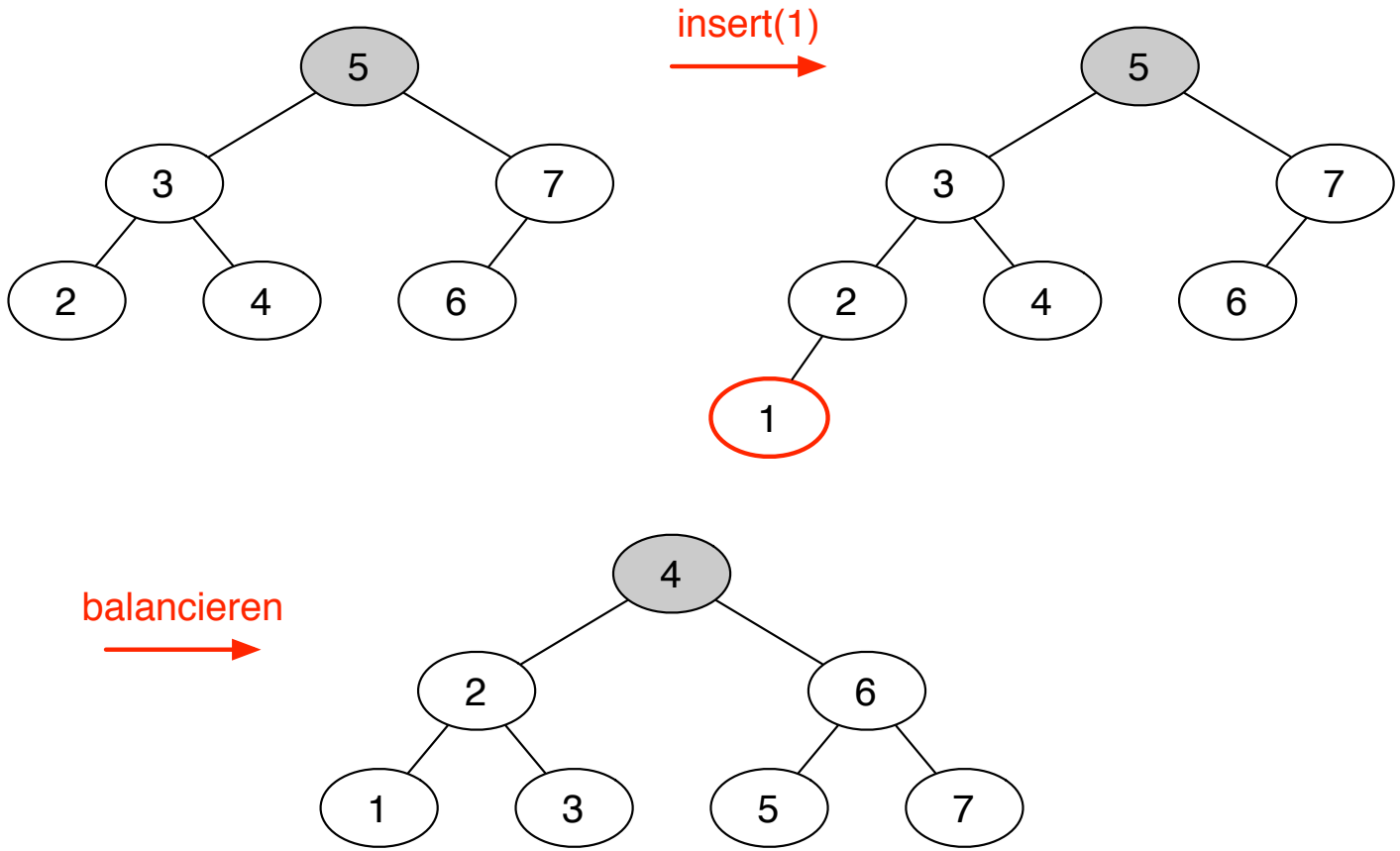


entartet

- Wie können Suchbäume entarten?
  - Beispiel: einfügen aus sortierter Liste
- Erwünscht: Suchbäume, die immer ausgeglichen (balanciert) bleiben

  $\rightarrow$ 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

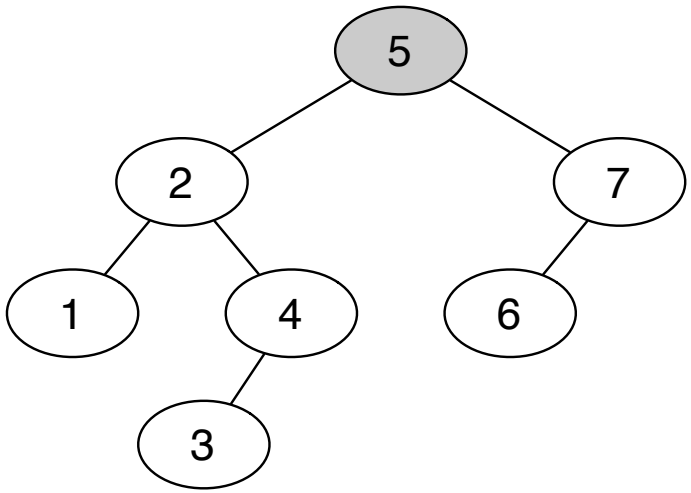
Definition AVL-Baum

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

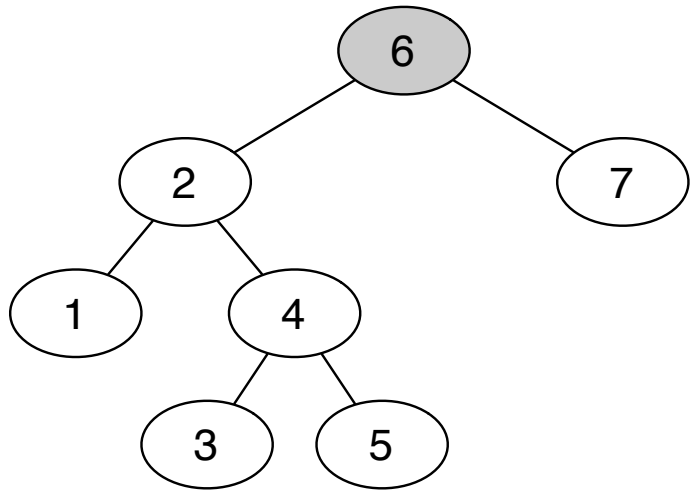
- für jeden inneren Knoten $v \in V$ gilt: Höhe von linkem und rechtem Teilbaum von v unterscheidet sich maximal um 1.

- 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* = 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

Einfügen / Verletzung AVL-Bedingung

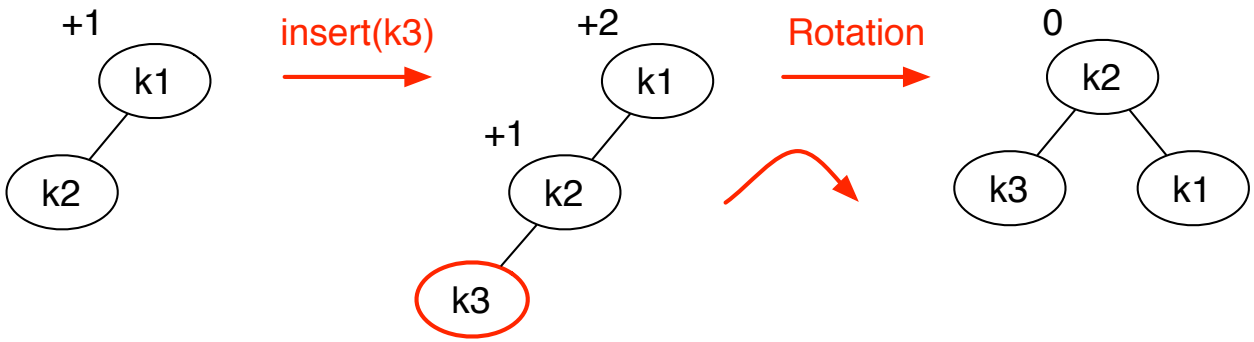
Fallunterscheidung Verletzung AVL-Bedingung bei Einfügen:

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

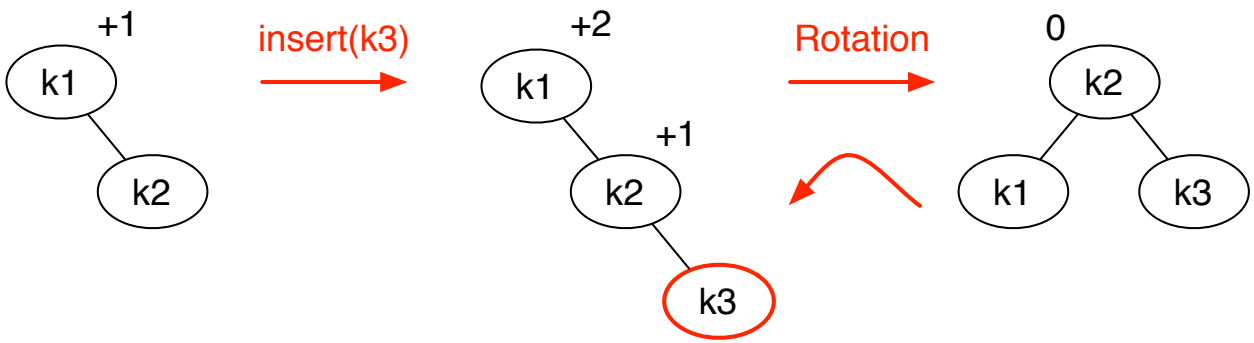
1 und 4 sind symmetrische Fälle, sowie 2 und 3

AVL-Baum: Rotation

1. Einfügen in linken Teilbaum des linken Kindes:

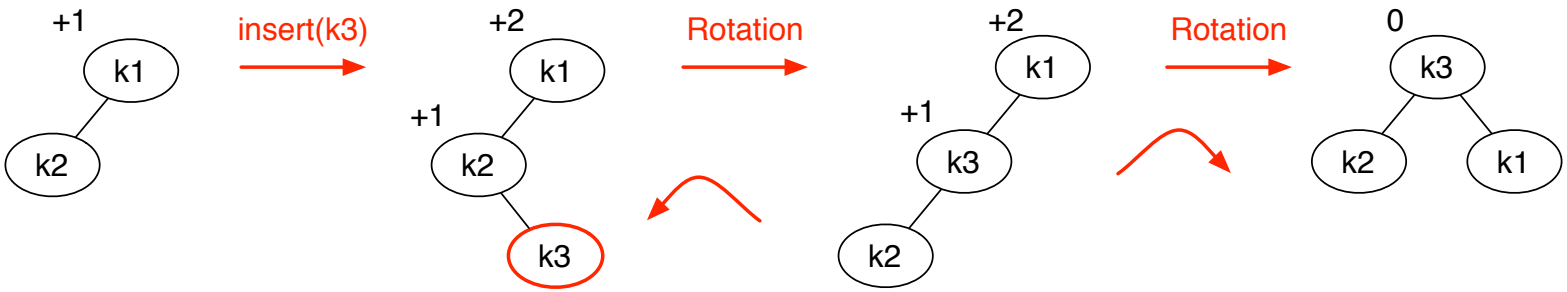


4. Einfügen in rechten Teilbaum des rechten Kindes:

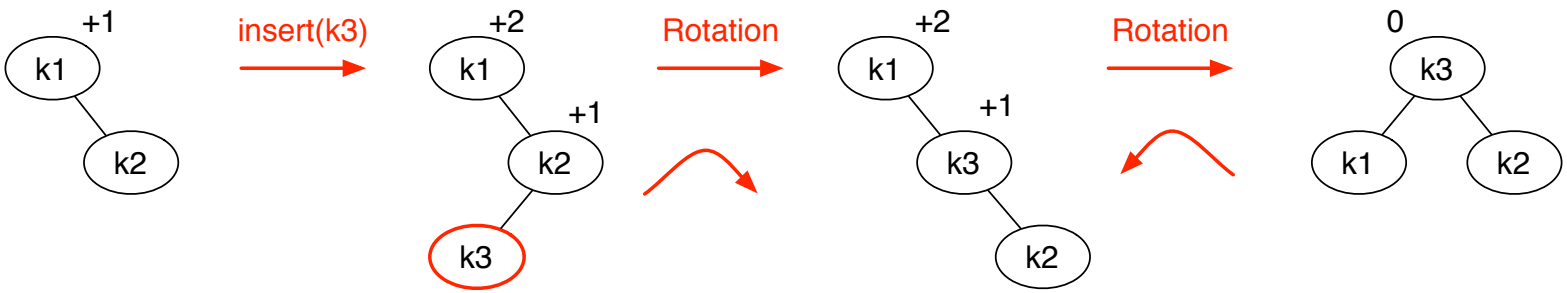


AVL-Baum: Doppelrotation

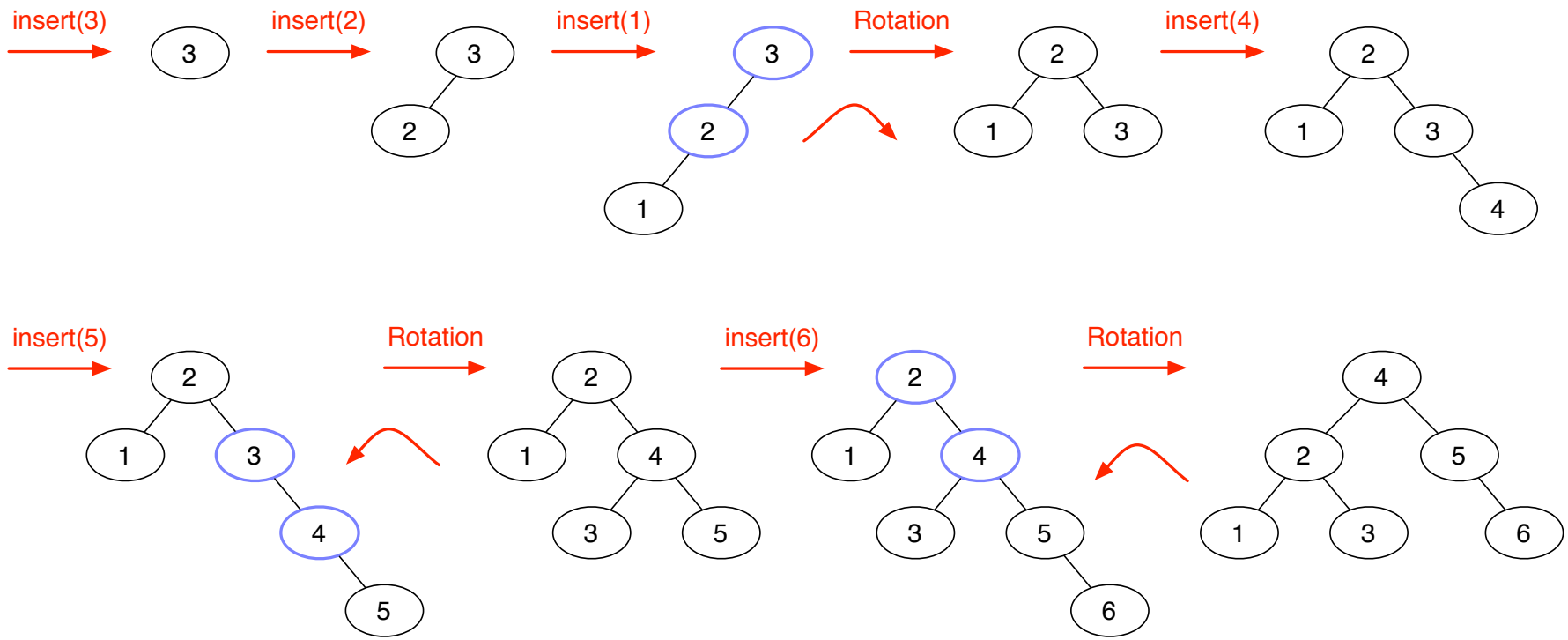
2. Einfügen in rechten Teilbaum des linken Kindes:



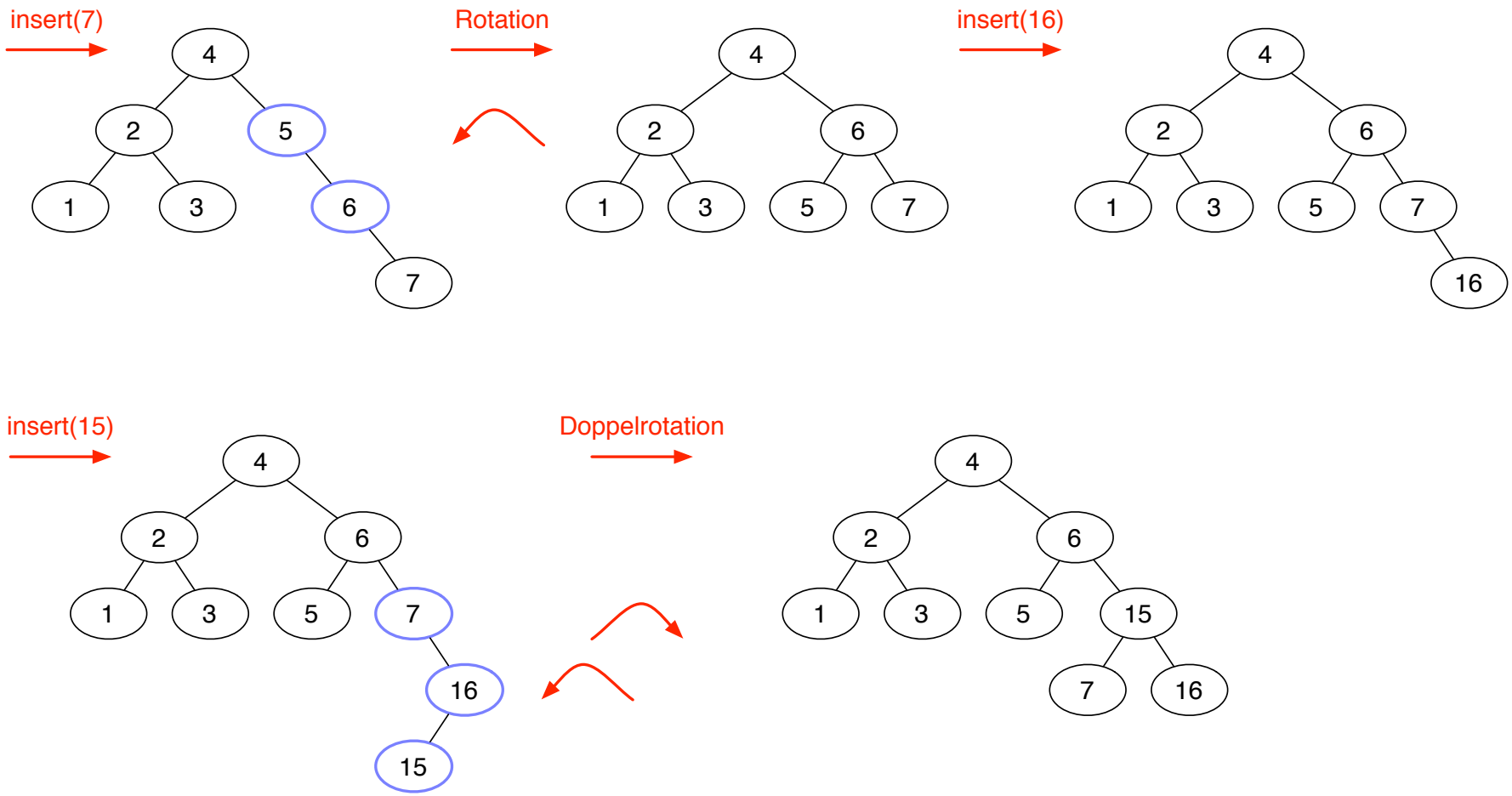
3. Einfügen in linken Teilbaum des rechten Kindes:



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

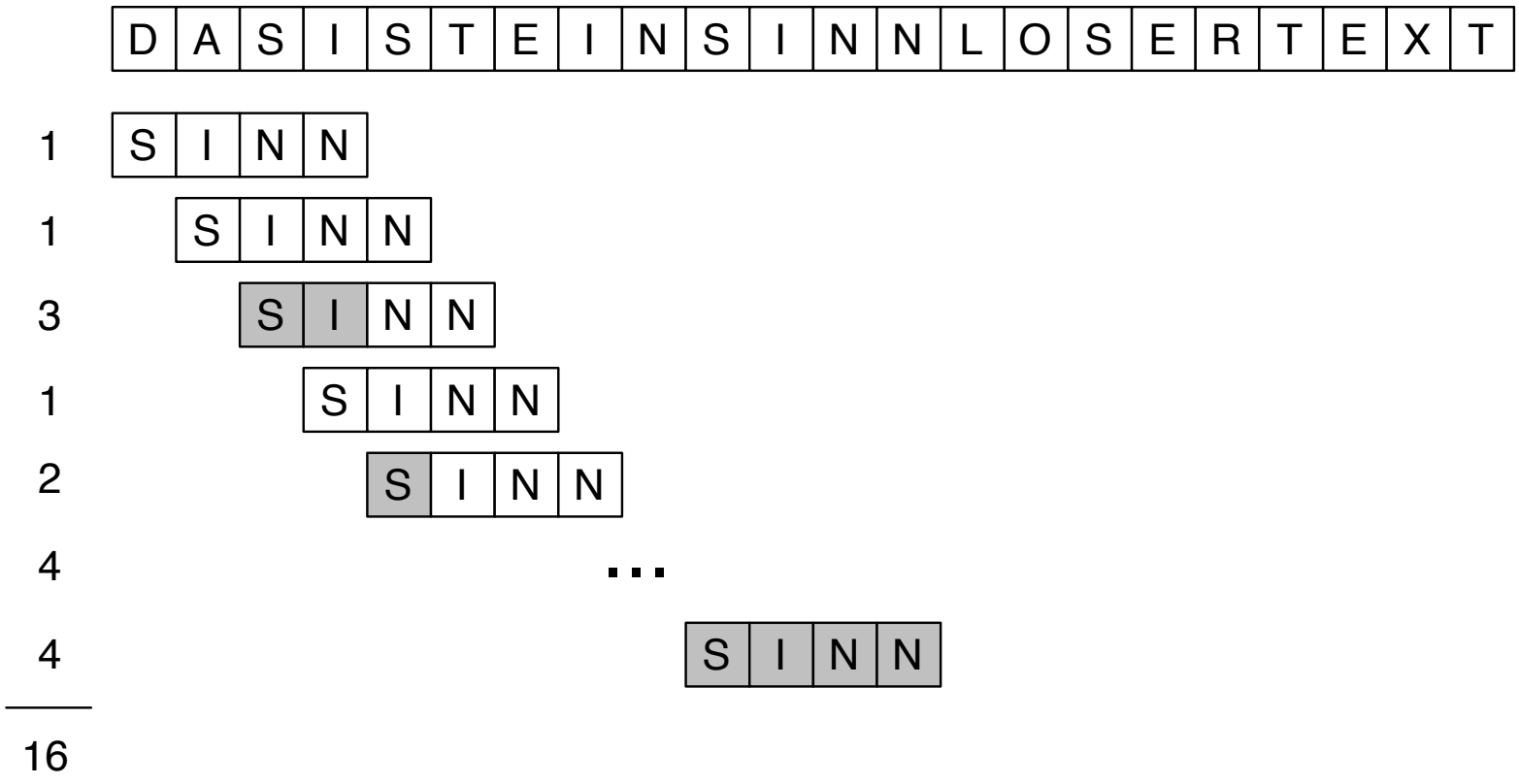
Programm

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



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
```

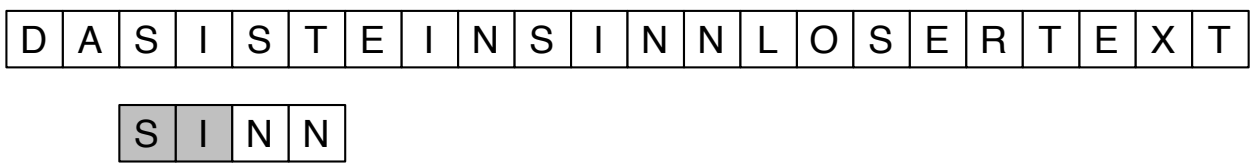
- Komplexität: $O((n-m)m) = O(nm)$

Knuth-Morris-Pratt Algorithmus

Knuth-Morris-Pratt Algorithmus (kurz: KMP)

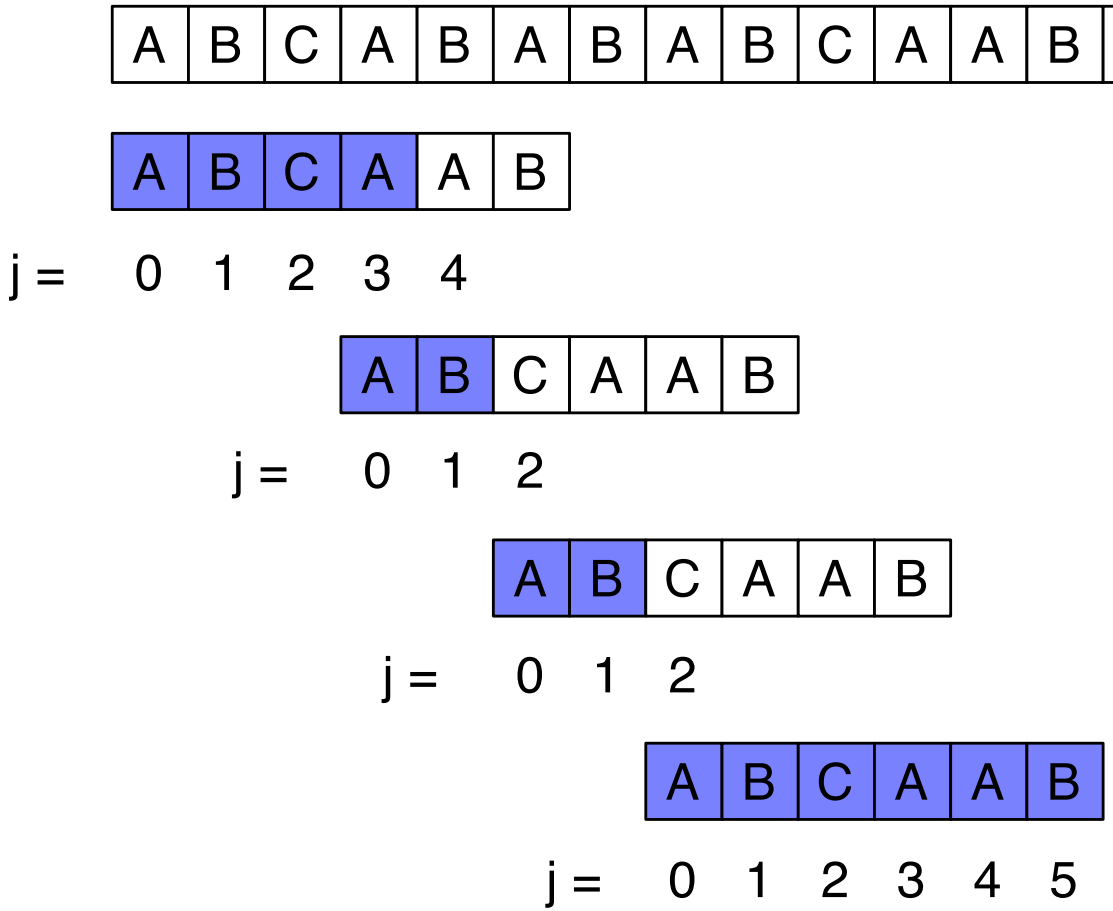
- Idee: verbessere Brute-Force Algorithmus durch Ausnutzung der bereits gelesenen Information bei einem Mismatch
- Mismatch an Stelle j von *pat* impliziert

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



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

KMP Algorithmus: Beispiel



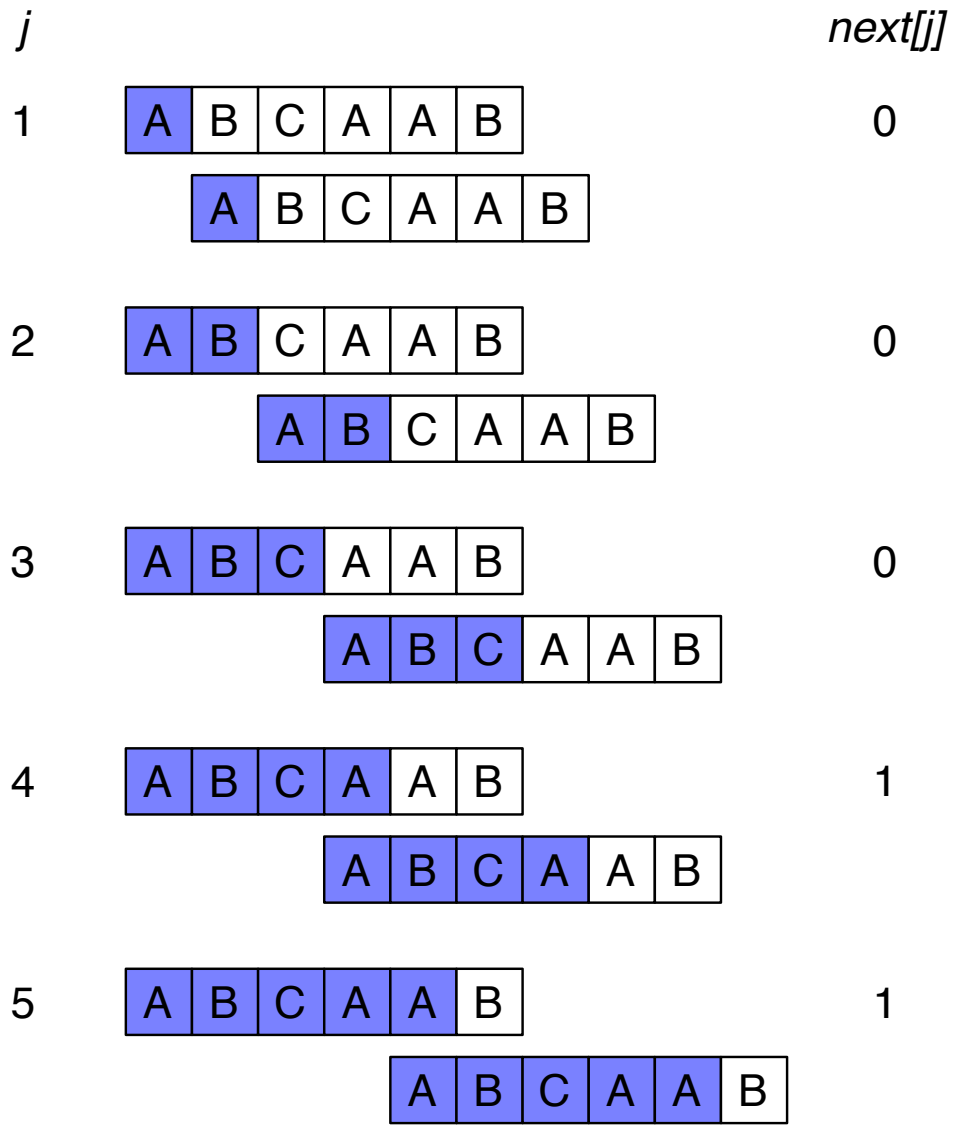
j	0	1	2	3	4	5
$next[j]$	-1	0	0	0	1	1

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



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

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen in Zeichenketten