Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2014

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 mit Hashtabellen
   - 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

Optionales C++ Beispiel: lineare Suche

Code:

```
std::vector<int> feldA{0, 8, 15, 47, 11};
int wert = 15;

// lineare Suche nach wert in feldA
auto p = std::find(feldA.begin(), feldA.end(), wert);

// ist p am Ende von feldA angelangt, wurde nichts gefunden
if (p == feldA.end())
    std::cout << "Wert␣" << wert << "␣nicht␣gefunden";
else { // gefunden!  (*p dereferenziert den Iterator)
    std::cout << "Wert␣" << *p << "␣gefunden␣an␣Stelle␣"
              << std::distance(feldA.begin(), p);
}
```

Ausgabe:

```
Wert 15 in feldA gefunden an Stelle 2
```

Programm heute

7. Fortgeschrittene Datenstrukturen

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen mit Hashtabellen
   - 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)

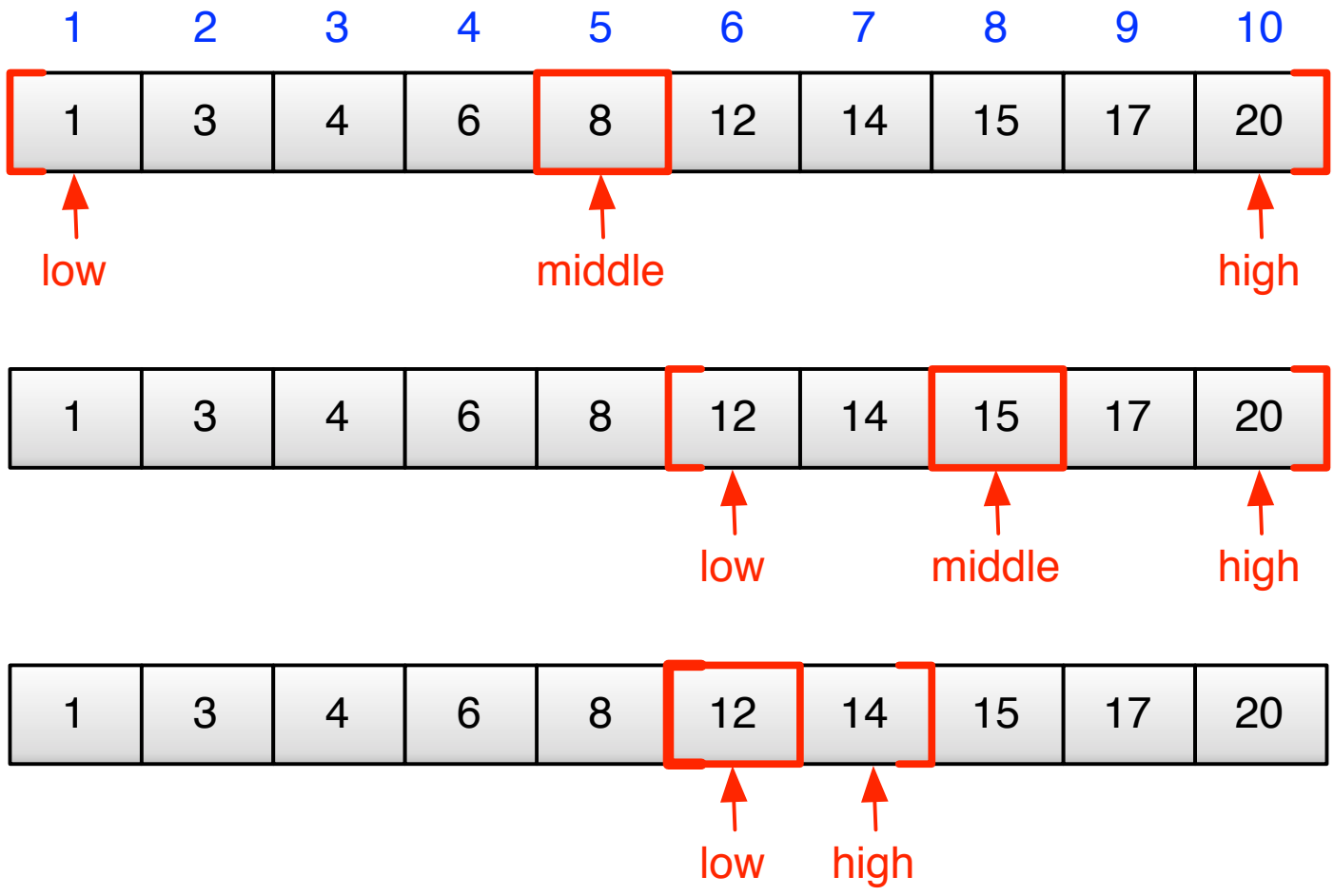
Binäre Suche: Algorithmus iterativ

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

Optionales C++ Beispiel: binäre Suche

Code:

```
std::vector<int> feldA{0, 8, 15, 47, 11};
int wert = 15;

// feldA sortieren und ausgeben
std::sort(feldA.begin(), feldA.end());
for (int x: feldA)
    std::cout << x << "␣";
std::cout << std::endl;

// binaere Suche nach wert in feldA
bool found = std::binary_search(feldA.begin(), feldA.end(),
                                wert);
if (found)
    std::cout << "Wert␣" << wert << "␣gefunden!";
else
    std::cout << "Wert␣" << wert << "␣nicht␣gefunden!";
```

Ausgabe:

```
0 8 11 15 47
Wert 15 gefunden!
```

Programm heute

7. Fortgeschrittene Datenstrukturen

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen mit Hashtabellen
   - 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

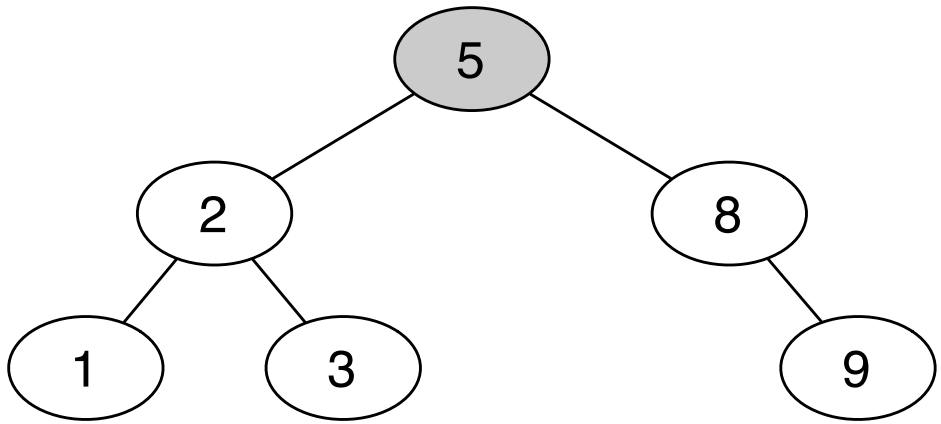
- für alle Knoten x im linken Teilbaum $v.left$ gilt

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

- 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)

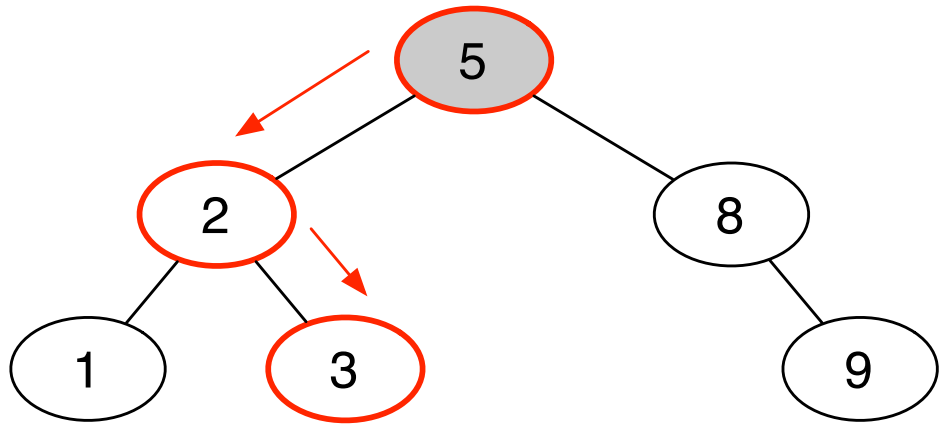
Input: Knoten v, dessen Teilbaum untersucht werden soll,
k gesuchter Schlüssel
Output: Knoten mit gesuchtem Schlüssel,
null falls nicht gefunden

```
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

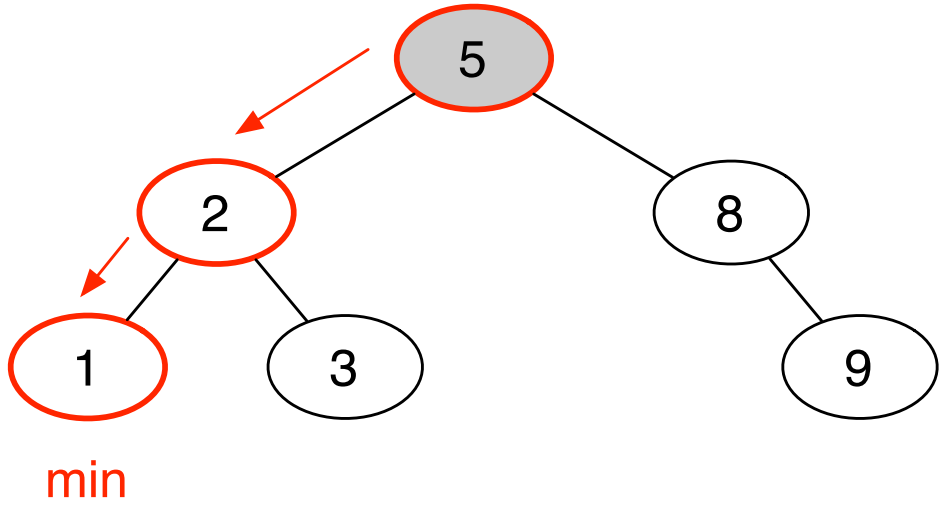
Input: Wurzel v des zu durchsuchenden Baumes
Output: Knoten mit minimalem Schlüssel
minimum(v):

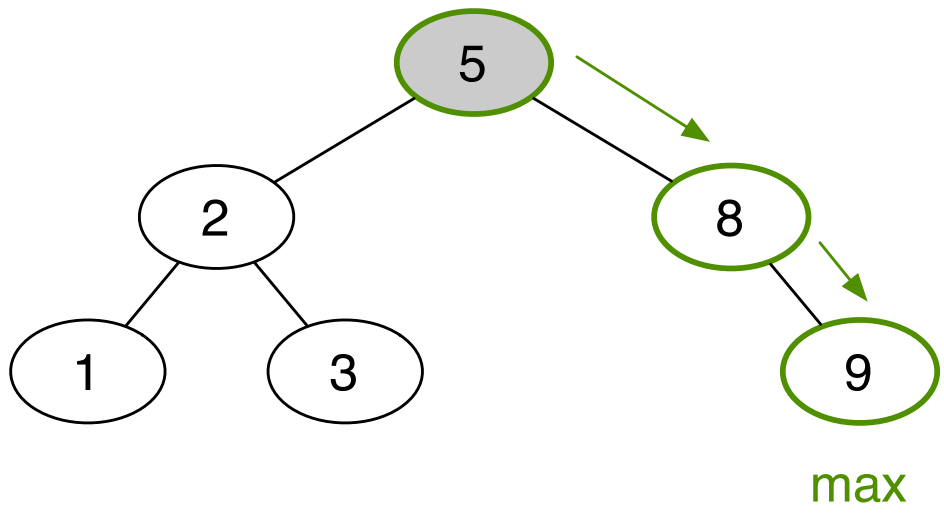
```
    while (v.left != null)
        v = v.left;
    return v;
```

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

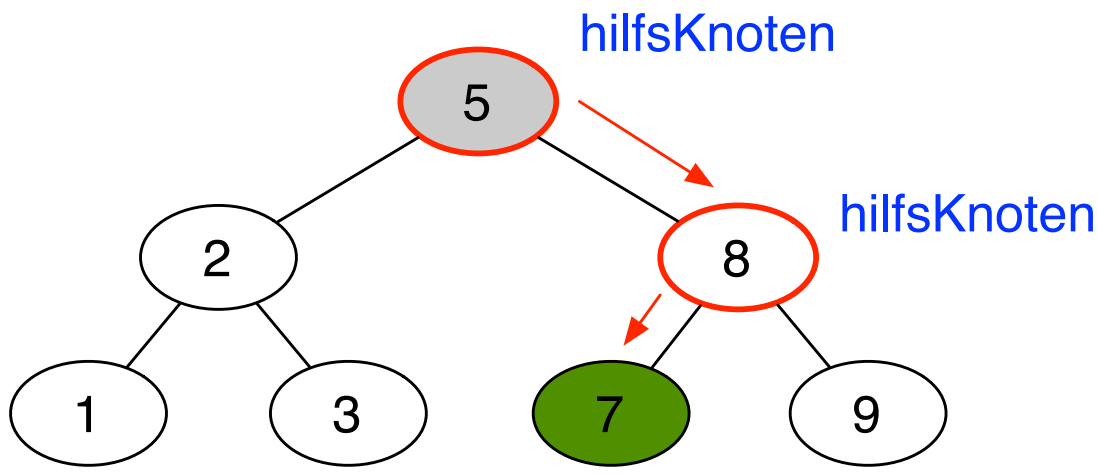
Binärer Suchbaum: Einfügen

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:



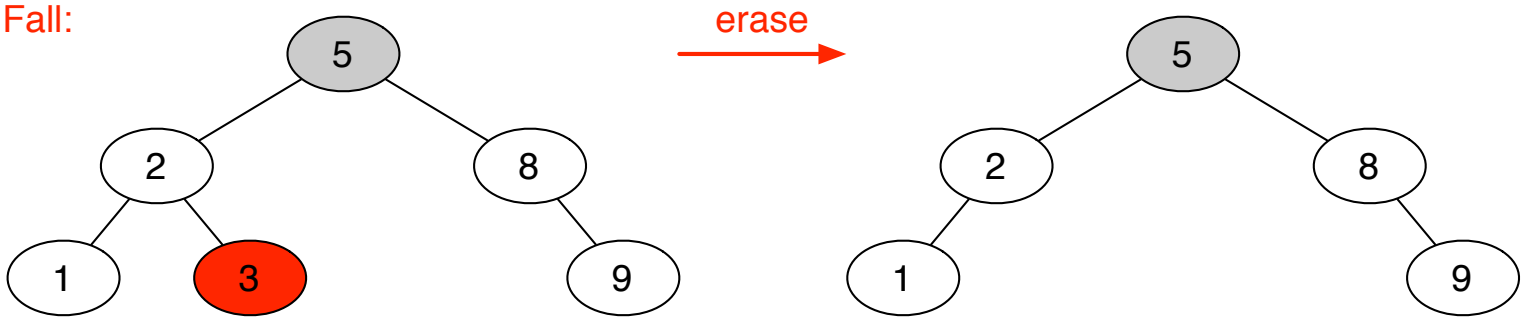
- Komplexität: $O(h)$, wobei h Höhe von Suchbaum

Binärer Suchbaum: Löschen

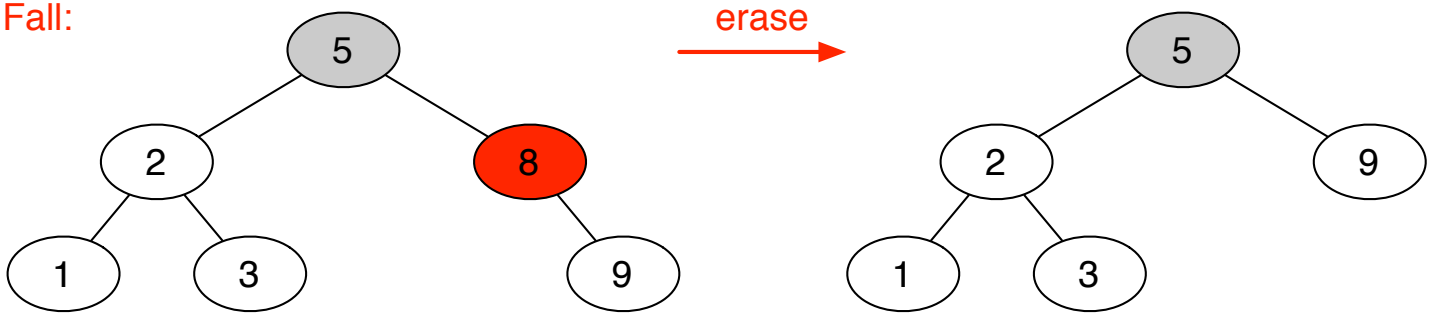
- Löschen von Knoten x in Suchbaum ist etwas komplizierter
- Drei Fälle:
  1. x ist Blatt: einfach entfernen
  2. x hat nur ein Kind: setze Kind an Stelle von x
  3. x hat zwei Kinder: setze minimales Element von rechtem Teilbaum an Stelle von x
     (alternativ: maximales Element von linkem Teilbaum)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

Binärerer Suchbaum: Löschen

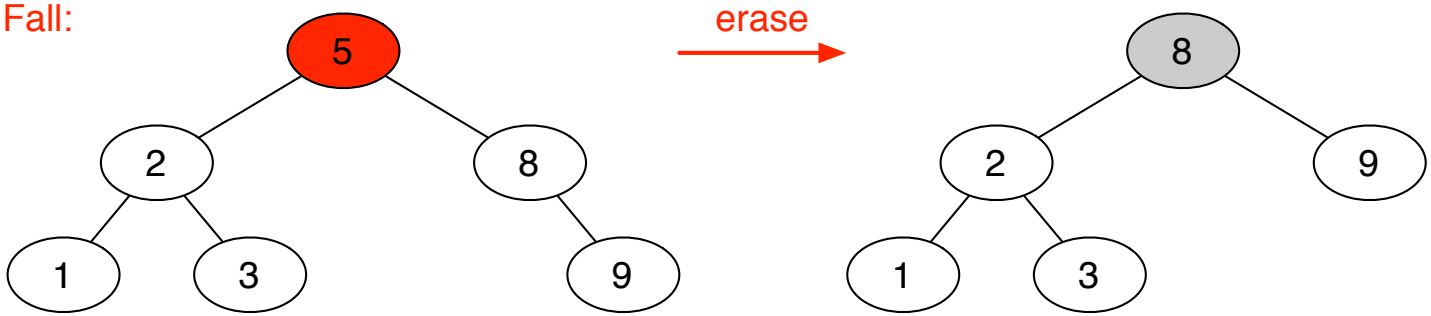
1. Fall:



2. Fall:



3. Fall:



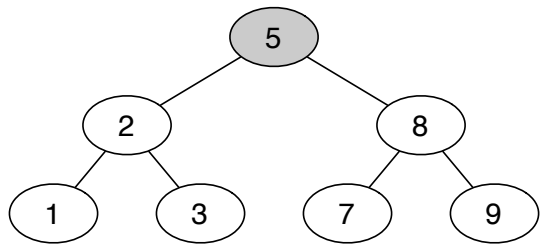
Binärer Suchbaum: Löschen

Input: Wurzel v des Baumes, x zu löschendes Element

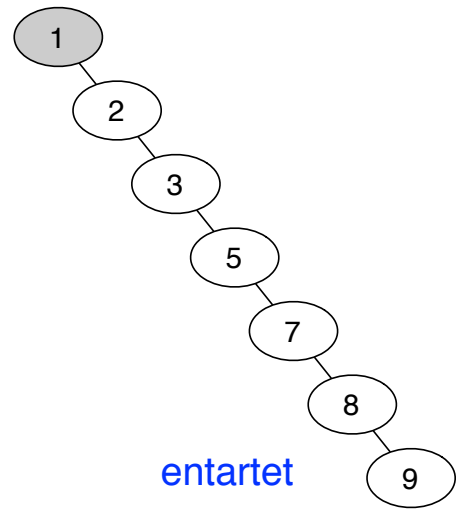
```
erase(v, x):
    if (x ist Blatt) { // 1. Fall
        if (x ist linkes Kind) x.vater.left = null;
        else x.vater.right = null;
    } else { // 2. Fall
        if (x.left == null) {
            if (x ist linkes Kind) x.vater.left = x.right;
            else x.vater.right = x.right;
       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

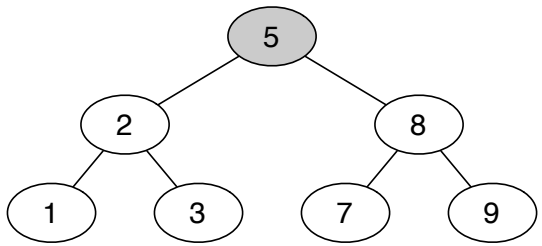
- Ausweg: automatisch balancierte Suchbäume
  (z.B. AVL Bäume, Rot-Schwarz Bäume, B-Bäume)

Programm heute

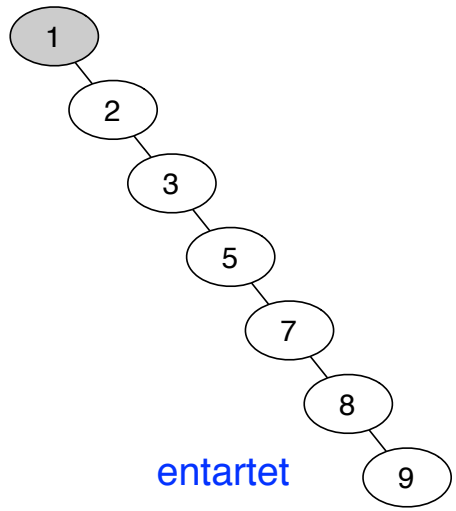
7. Fortgeschrittene Datenstrukturen

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

Entartete Suchbäume



ausgeglichen

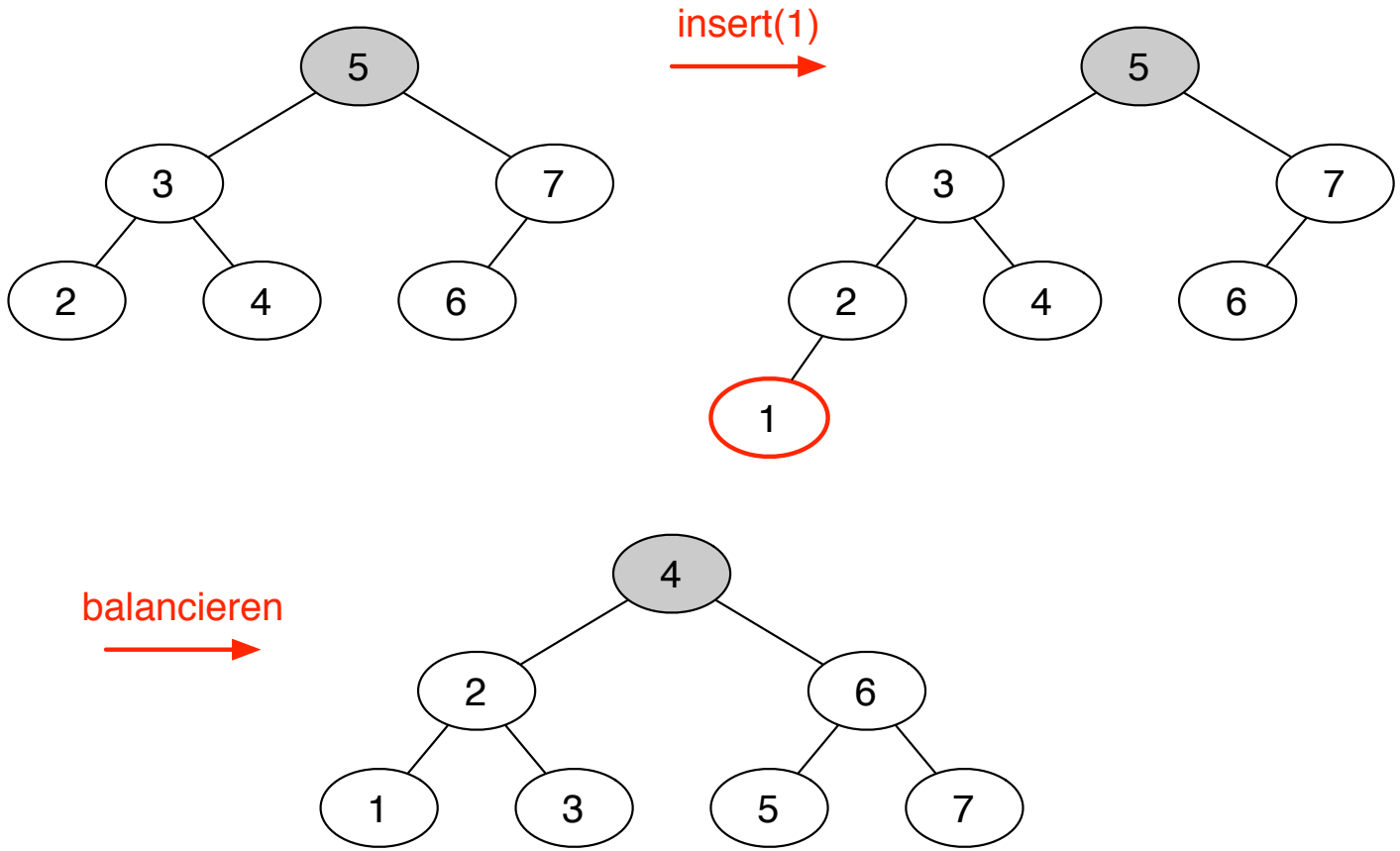


entartet

- 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 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

Definition AVL-Baum

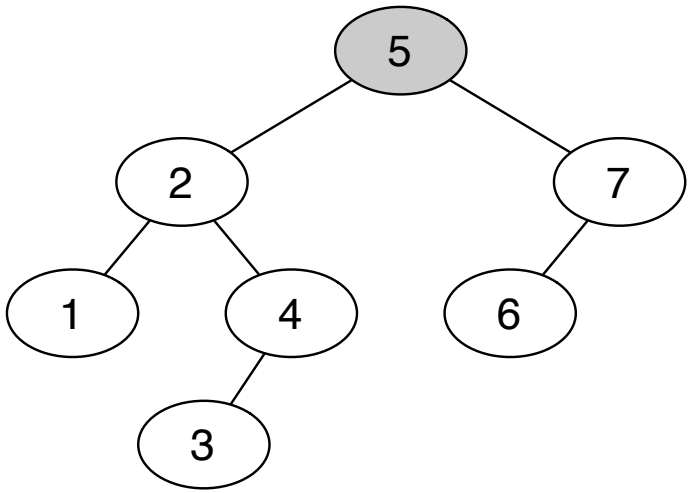
Definition AVL-Baum

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

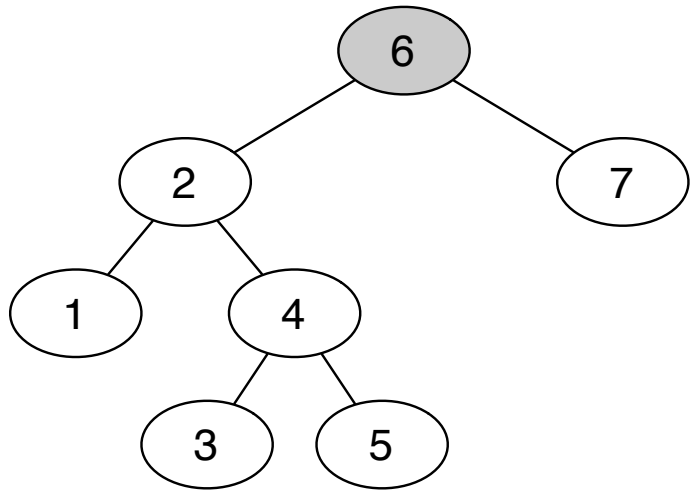
- für jeden inneren Knoten $v \in V$ gilt: Höhe von linkem und rechtem Teilbaum von v unterscheidet sich maximal um 1.

- 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

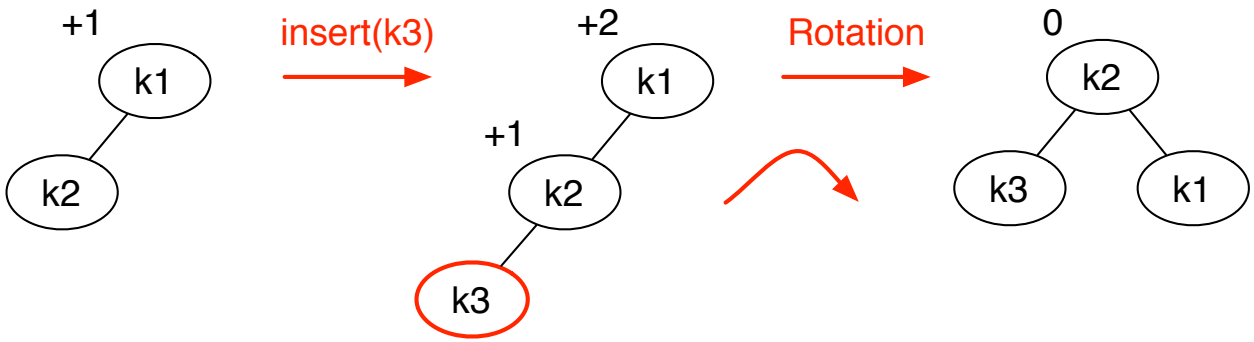
Fallunterscheidung Verletzung AVL-Bedingung bei Einfügen:

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

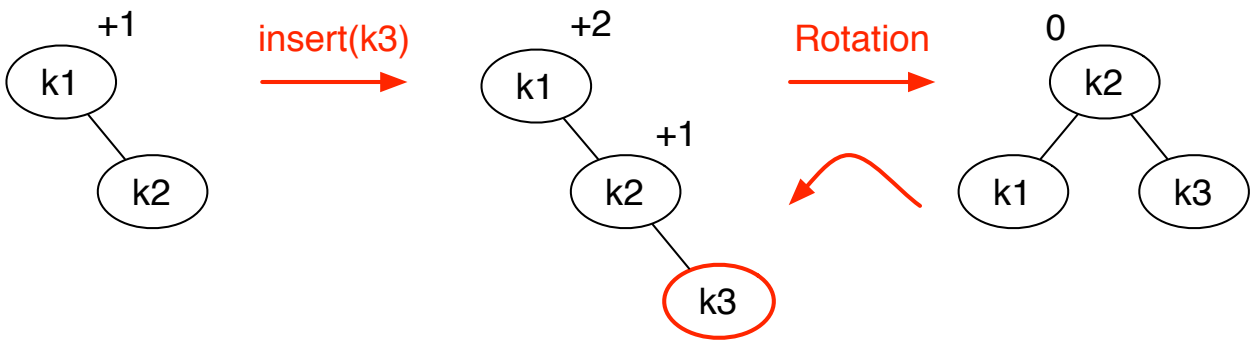
1 und 4 sind symmetrische Fälle, sowie 2 und 3

AVL-Baum: Rotation

1. Einfügen in linken Teilbaum des linken Kindes:

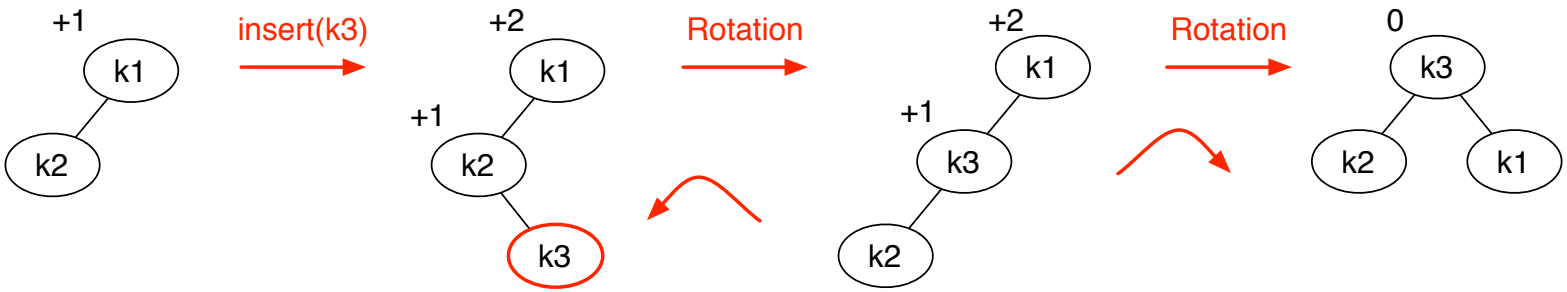


4. Einfügen in rechten Teilbaum des rechten Kindes:

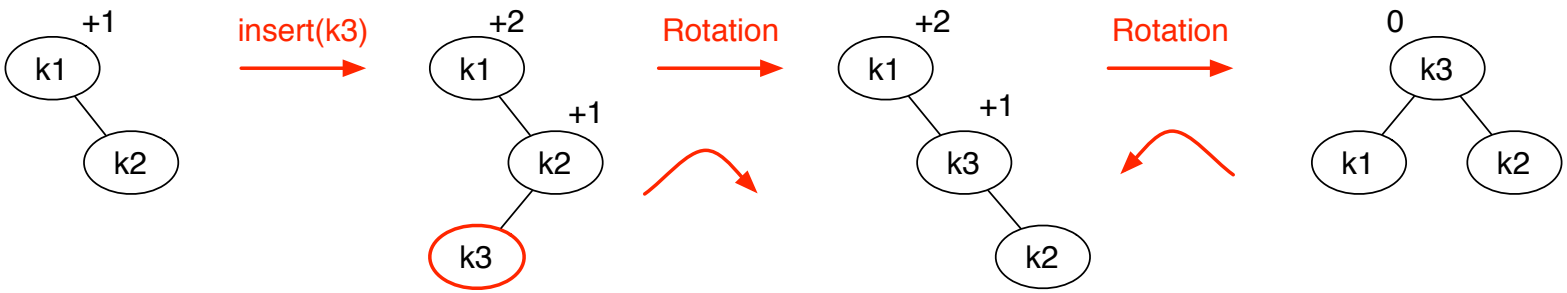


AVL-Baum: Doppelrotation

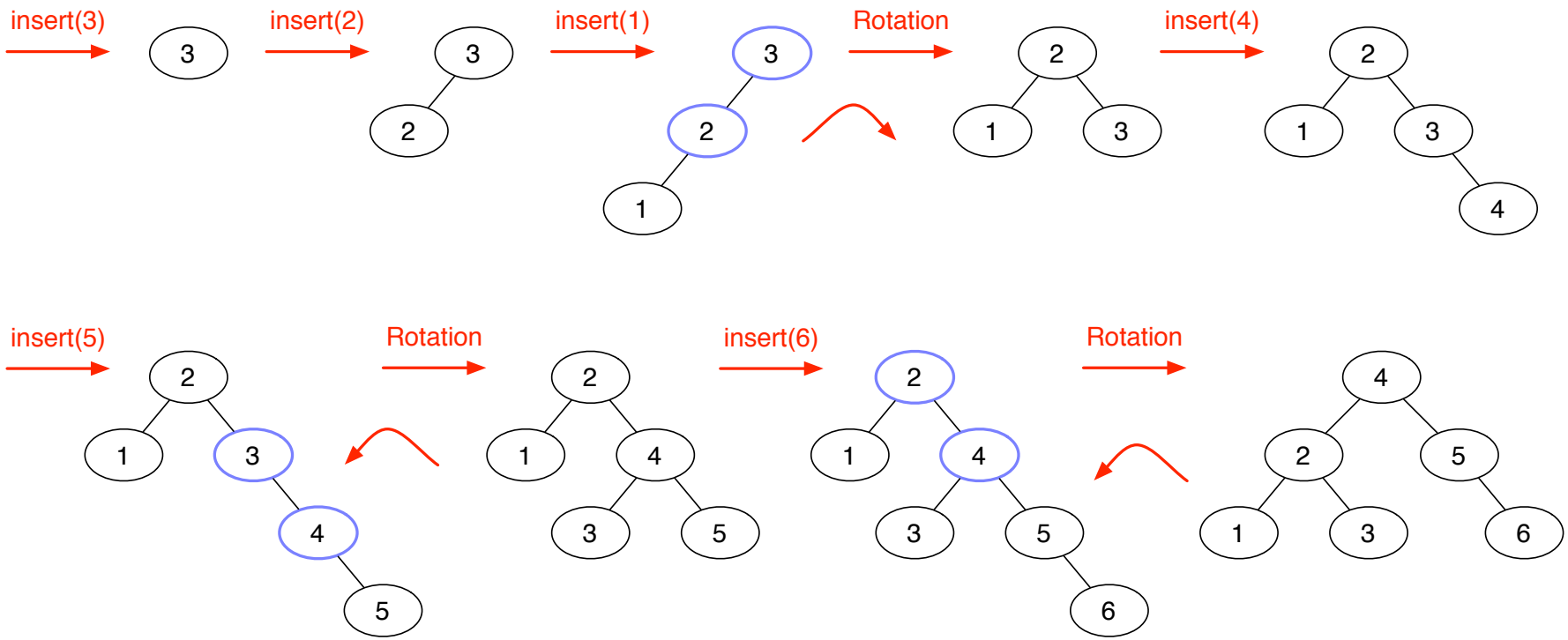
2. Einfügen in rechten Teilbaum des linken Kindes:



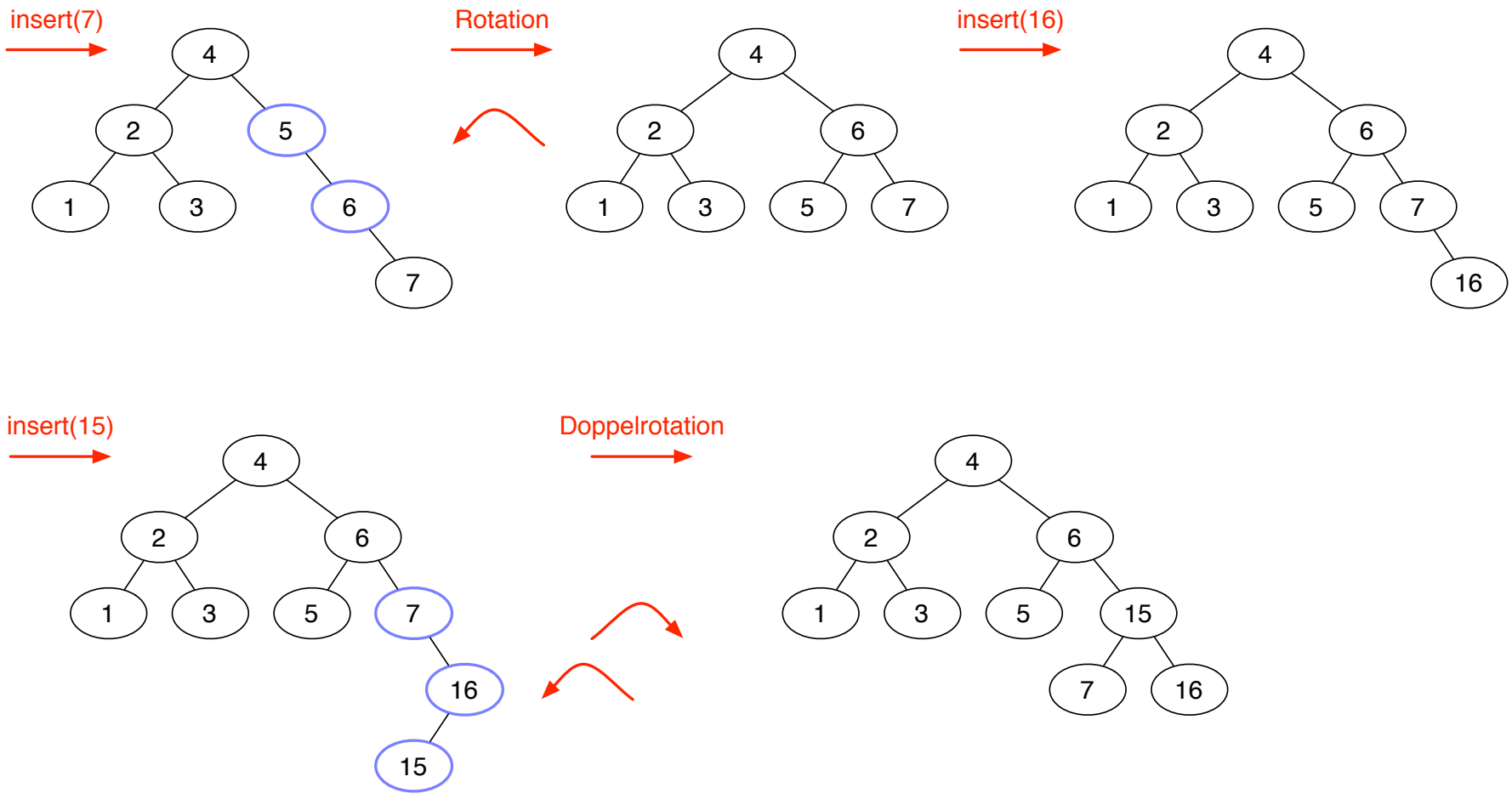
3. Einfügen in linken Teilbaum des rechten Kindes:



AVL-Baum: Beispiel-Sequenz I



AVL-Baum: Beispiel-Sequenz II

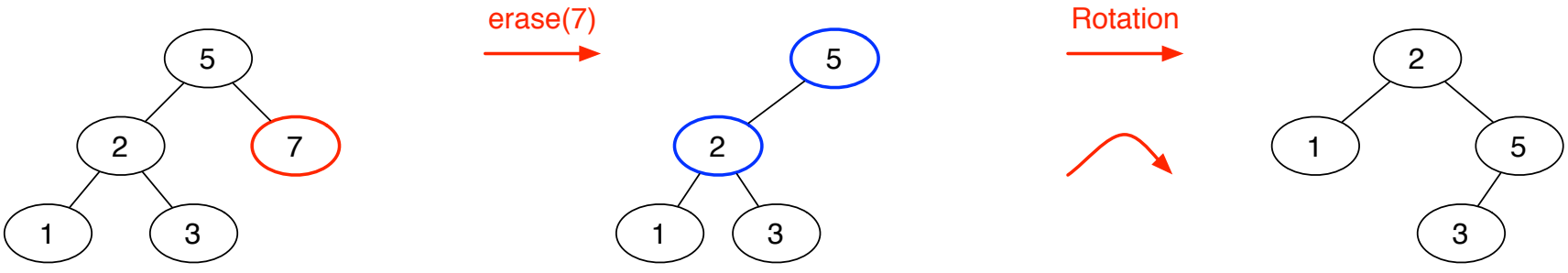


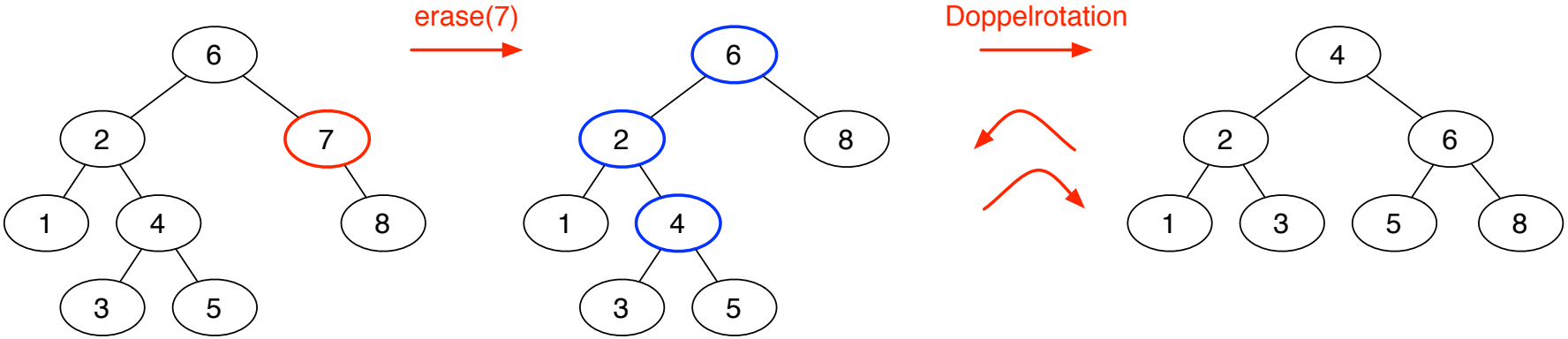
AVL-Baum: Löschen

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

AVL-Baum: Beispiel-Sequenz III





Programm heute

7. Fortgeschrittene Datenstrukturen

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

Wörterbücher

Wörterbuch

Ein Wörterbuch (auch genannt Dictionary oder Assoziatives Array) speichert eine Menge von Elementen M. Jedes Element $e \in M$ wird durch einen Schlüssel $key(e)$ eindeutig identifiziert.

Unterstützte Operationen sind:

- search(key k): finde $e \in M$ mit $key(e) = k$
- insert(Element e): erweitere M um e
- erase(key k): entferne e aus M, wobei $key(e) = k$

Beispiele:

- Telefonbuch
  - Element: Name, Adresse und Telefonnummer
  - Schlüssel: Name
- Compiler-Symboltabelle
  - Element: Bezeichner und Typ-/Speicher-Informationen
  - Schlüssel: Bezeichner

Such-Algorithmen für Wörterbücher

Sei M Wörterbuch mit $|M| = n$.

- M als verkettete Liste mit linearer Suche
  - search O(n), insert O(1), erase O(n)
- M als sortierte verkettete Liste mit binärer Suche
  - search O(log n), insert O(n), erase O(log n)
- M als AVL-Suchbaum
  - search O(log n), insert O(log n), erase O(log n)
- M als Hashtabelle
  - search O(1), insert O(1), erase O(1) im Mittel!
  - Worst case: search O(n), insert O(1), erase O(n)

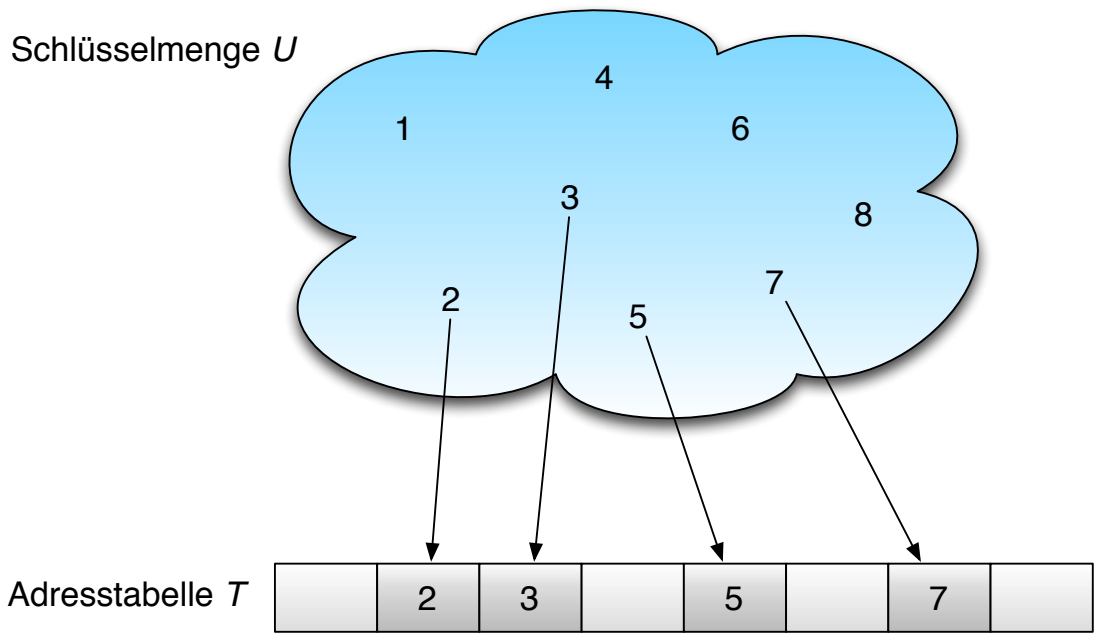
Adresstabellen

Sei M Wörterbuch.

- Setze $U := \{key(e) : e \in M\}$ die Menge aller Schlüssel
- Annahme: $key(e_1) \neq key(e_2)$ für alle $e_1 \neq e_2$, $e_1, e_2 \in M$

Wörterbuch mit Adresstabelle:

- sequentielle Liste T der Länge $|U|$



Operationen auf Adresstabellen

Adresstabelle T mit Länge $|U|$.

- **Input:** Tabelle T, Schlüssel k
  Output: Element mit Schlüssel k
  search(T, k):
  return $T[k]$;

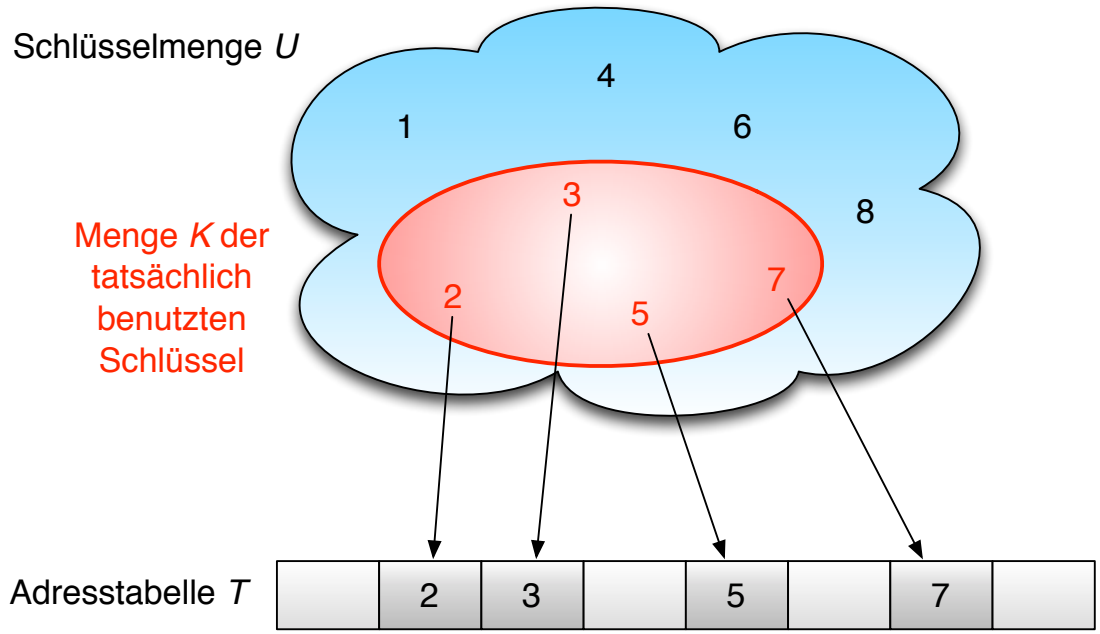
- **Input:** Tabelle T, Element e
  insert(T, e):
  $T[key(e)] = e$;

- **Input:** Tabelle T, Schlüssel k
  erase(T, k):
  $T[k] = \text{null}$;

Alle Operationen: Laufzeit $O(1)$
Speicherkomplexität: $O(|U|)$ $\rightarrow$ Problem falls U groß!

Reduktion von Adresstabellen mittels Hashfunktion

- Beobachtung: Menge K der tatsächlich benutzten Schlüssel aus U oft klein

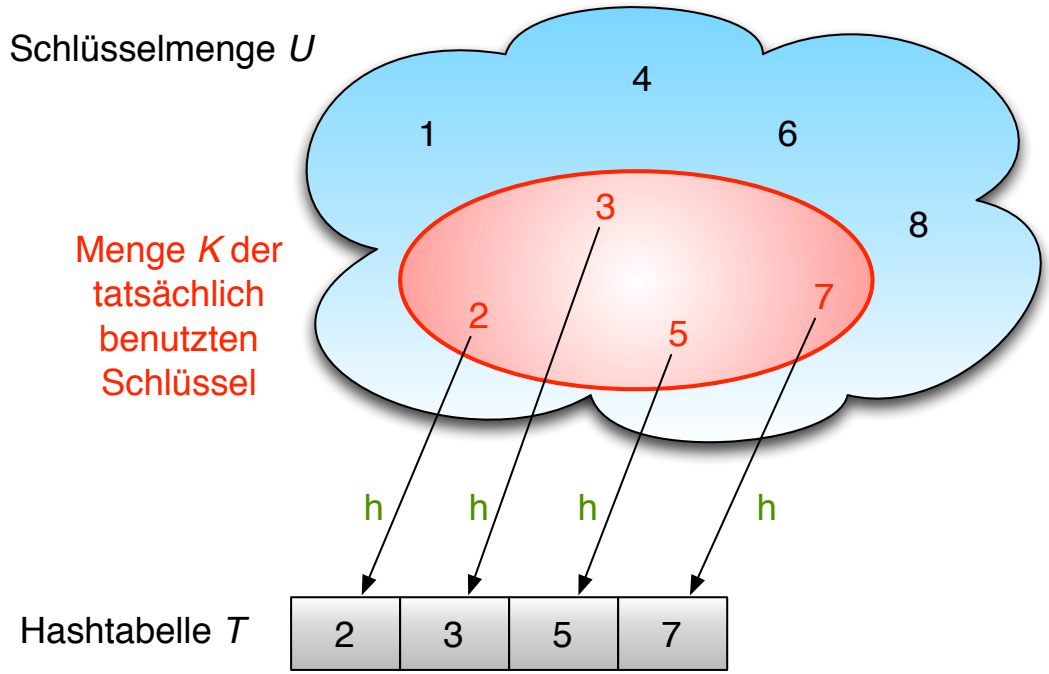


- Idee: wende Hashfunktion h auf $key(e)$ an, um Adresstabelle auf $|K|$ Elemente zu reduzieren $\rightarrow$ Hashtabelle

Hashtabellen

Wörterbuch mit Hashtabelle:

- sequentielle Liste T mit Länge $m := |K|$
- Hashfunktion $h : U \to \{0, \ldots, m-1\}$



Operationen auf Hashtabellen

Hashtabelle T mit Länge m und Hashfunktion h.

- **Input:** Tabelle T, Schlüssel k
  Output: Element mit Schlüssel k
  search(T, k):
  return $T[h(k)]$;

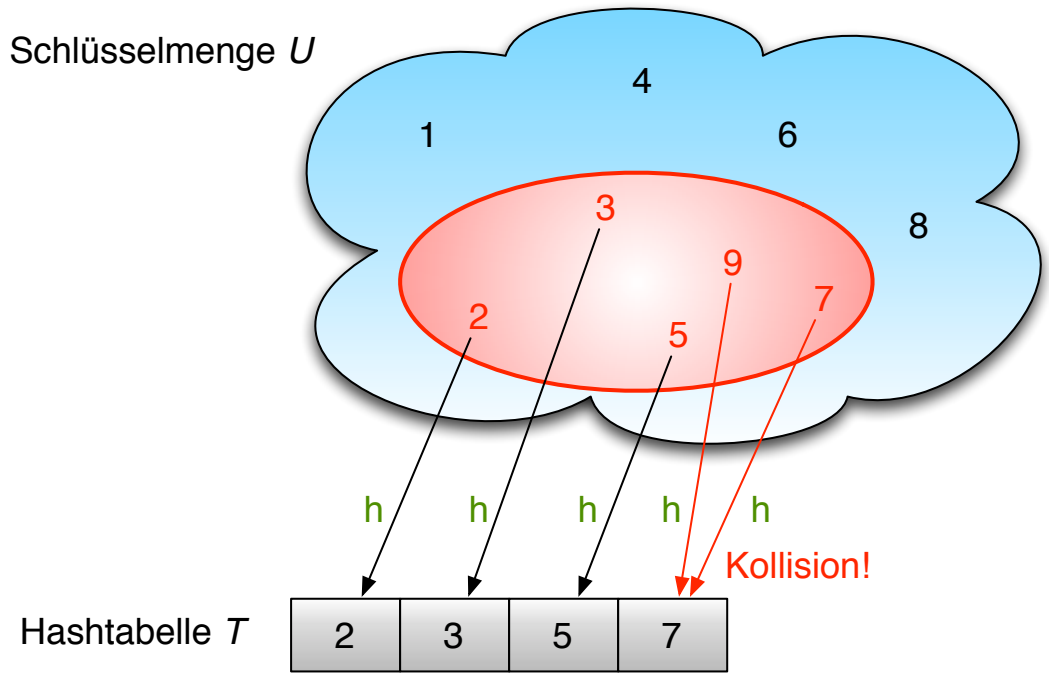
- **Input:** Tabelle T, Element e
  insert(T, e):
  $T[h(key(e))] = e$;

- **Input:** Tabelle T, Schlüssel k
  erase(T, k):
  $T[h(k)] =$ null;

Alle Operationen: Laufzeit $O(1)$ (sofern h auch $O(1)$)
Speicherkomplexität: $O(m)$

Kollisionen bei Hashtabellen

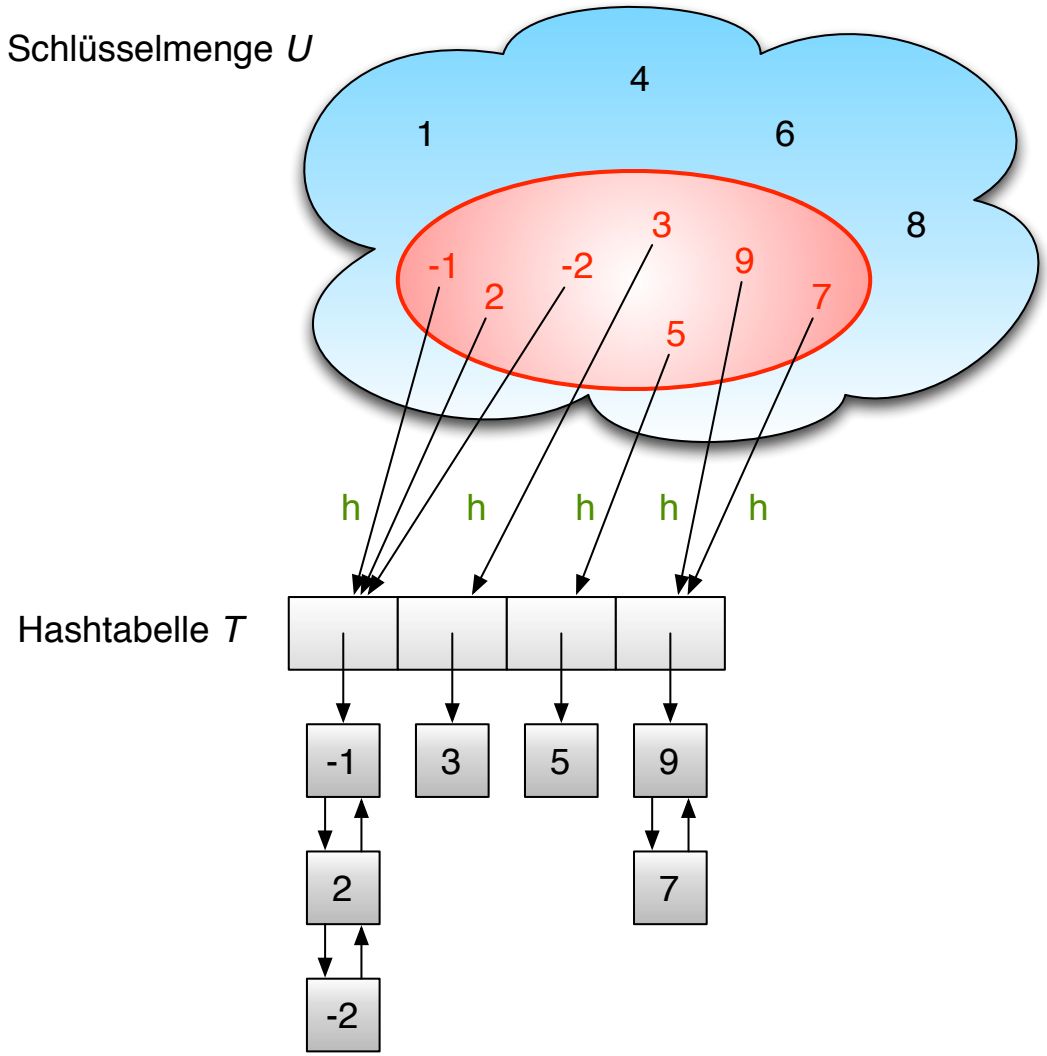
- Problem: falls h nicht injektiv $\rightarrow$ Kollision!
  - h ist nie injektiv da $|K| < |U|$



- Offene Fragen:
  - Strategie zur Kollisionsauflösung
  - Wahl von h

Verkettung zur Kollisionsauflösung

- jeder Slot in T enthält statt Element Referenz auf verkettete Liste
- bei Kollision: füge Element am Anfang der Liste ein $\rightarrow$ O(1)
- search und erase müssen nun die Liste durchlaufen $\rightarrow$ O(1) gefährdet!



Analyse von Hashing mit Verkettung

Sei T Hashtabelle mit m Slots und n gespeicherten Elementen.

- Belegungsfaktor $\alpha = n/m$
  - mittlere Anzahl von Elementen in verketteten Listen
- es kann gezeigt werden:
  Anzahl der Listendurchläufe ist $O(1+\alpha)$
- Annahme: es ist $n = cm$ mit c Konstante, dann
  $\alpha = n/m = O(m)/m = O(1)$
- Komplexität von search und erase:
  - im Mittel: O(1)
  - worst case: O(n)

Hashfunktionen

Sei U Menge aller Schlüssel, T Hashtabelle mit m Einträgen.

$$h : U \to \{0, \ldots, m-1\}$$

- Anforderungen an Hashfunktion:
  - einfache Auswertung $O(1)$
  - möglichst wenige Kollisionen
- viele Ansätze, zum Beispiel:
  - Divisionsmethode
  - Multiplikationsmethode
  - universelles Hashing
- Vereinbarung: $U = \mathbb{N}_0$ zur Vereinfachung
  - Strings z.B. via ASCII Code auf Zahlen abbilden

Divisionsmethode

Divisionsmethode

Sei $U = \mathbb{N}_0$ die Schlüsselmenge, sei T Hashtabelle mit m Einträgen. Die Divisionsmethode wählt die Hashfunktion folgendermaßen:

$$h : U \to \{0, \ldots, m-1\}, \quad h(k) = k \bmod m.$$

- Beispiel: $m = 12$, $k = 100$, dann ist $h(k) = 4$.
- Problem: geschickte Wahl von m
  - eher schlecht ist $m = 2^p$ (wertet nur die unteresten p Bits von k aus)
  - eher gut ist oft eine Primzahl, nicht nahe an Zweierpotenz
    - Beispiel: erwartete Anzahl von Einträgen in Hashtabelle: 2000, gewünschter Belegungsfaktor 3 $\to$ $m = 701$ geeignete Primzahl

Multiplikationsmethode

Multiplikationsmethode

Sei $U = \mathbb{N}_0$ die Schlüsselmenge, sei T Hashtabelle mit m Einträgen. Die Multiplikationsmethode wählt die Hashfunktion folgendermaßen:

$$h : U \to \{0, \ldots, m-1\}, \quad h(k) = \lfloor m(kA \bmod 1) \rfloor,$$

wobei $A \in (0, 1)$ Konstante.

- Vorschlag von Knuth: $A \approx \frac{1}{2}(\sqrt{5} - 1)$
- typische Wahl: $m = 2^p$
- Beispiel: $m = 2^{14} = 16384$, $A = 2654435769/2^{32}$, und $k = 123456$, dann ist $h(k) = 67$.

Universelles Hashing

- Problem: für jede feste Hashfunktion gibt es Eingabesequenzen, die worst-case Laufzeit O(n) verursachen
- Lösung:

Universelles Hashing

Sei $U = \mathbb{N}_0$ die Schlüsselmenge, sei T Hashtabelle mit m Einträgen. Sei $\mathcal{H}$ eine endliche Menge von Hashfunktionen:

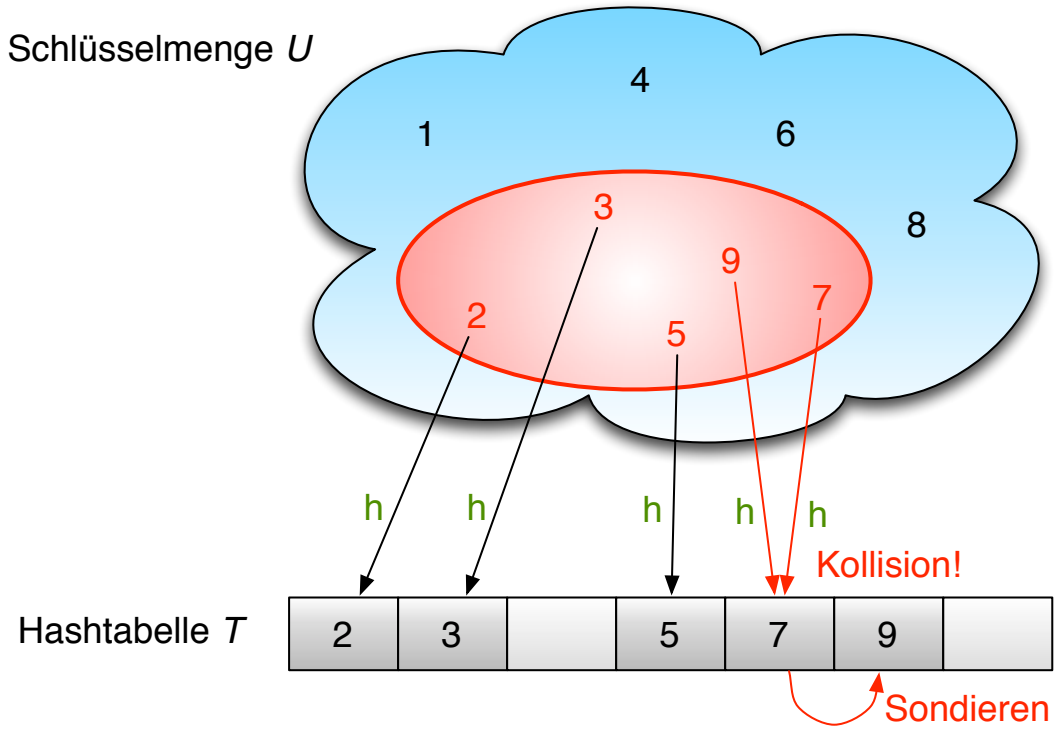
$$\mathcal{H} = \big\{h : U \to \{0, \ldots, m-1\} \mid h \text{ Hashfunktion}\big\}.$$

$\mathcal{H}$ heißt universell, falls für jedes Schlüsselpaar $k, l \in U$, $k \neq l$, die Anzahl der Hashfunktionen h mit $h(k) = h(l)$ durch $|\mathcal{H}|/m$ nach oben beschränkt ist.

- Idee: wähle zufällig ein $h \in \mathcal{H}$
- Resultat: Hashing mit Verkettung liefert im Mittel O(1) Laufzeit der Wörterbuch-Operationen

Kollisionsauflösung mit offener Adressierung

- Idee: statt Verkettung bei Kollision, suche systematisch nächsten freien Platz in Hashtabelle $\rightarrow$ Sondieren



- Probleme:
  - Löschen von Elementen (erase) sehr problematisch
  - Falls Hashtabelle T voll, kein insert mehr möglich

Offene Adressierung

Sei U Schlüsselmenge, T Hashtabelle mit m Einträgen.

- erweitere Hashfunktion zu

$$h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$$

- fordere, daß Sondierungssequenz

$$\big(h(k,0), h(k,1), \ldots, h(k,m-1)\big)$$

eine Permutation von $(0, \ldots, m-1)$ ist

$\to$ jeder Slot in T ist mögliches Ziel für Schlüssel k

- viele mögliche Sondierungssequenzen, zum Beispiel:
  - lineares Sondieren
  - quadratisches Sondieren
  - doppeltes Hashing

Hashing: Ausblick

- nachträgliche Änderung der Tabellengröße aufwendig
  - erfordert neue Tabelle und neue Hashfunktion
  - übertragen aller Elemente (und Berechnung neuer Hashfunktion) nötig

- ist U statisch, so ist perfektes Hashing möglich
  - auch im worst-case sind alle Operationen O(1)
  - umgesetzt durch zwei Ebenen von universellem Hashing
  - Zeitgewinn erkauft durch zusätzlichen Speicherbedarf von O(n)

Programm heute

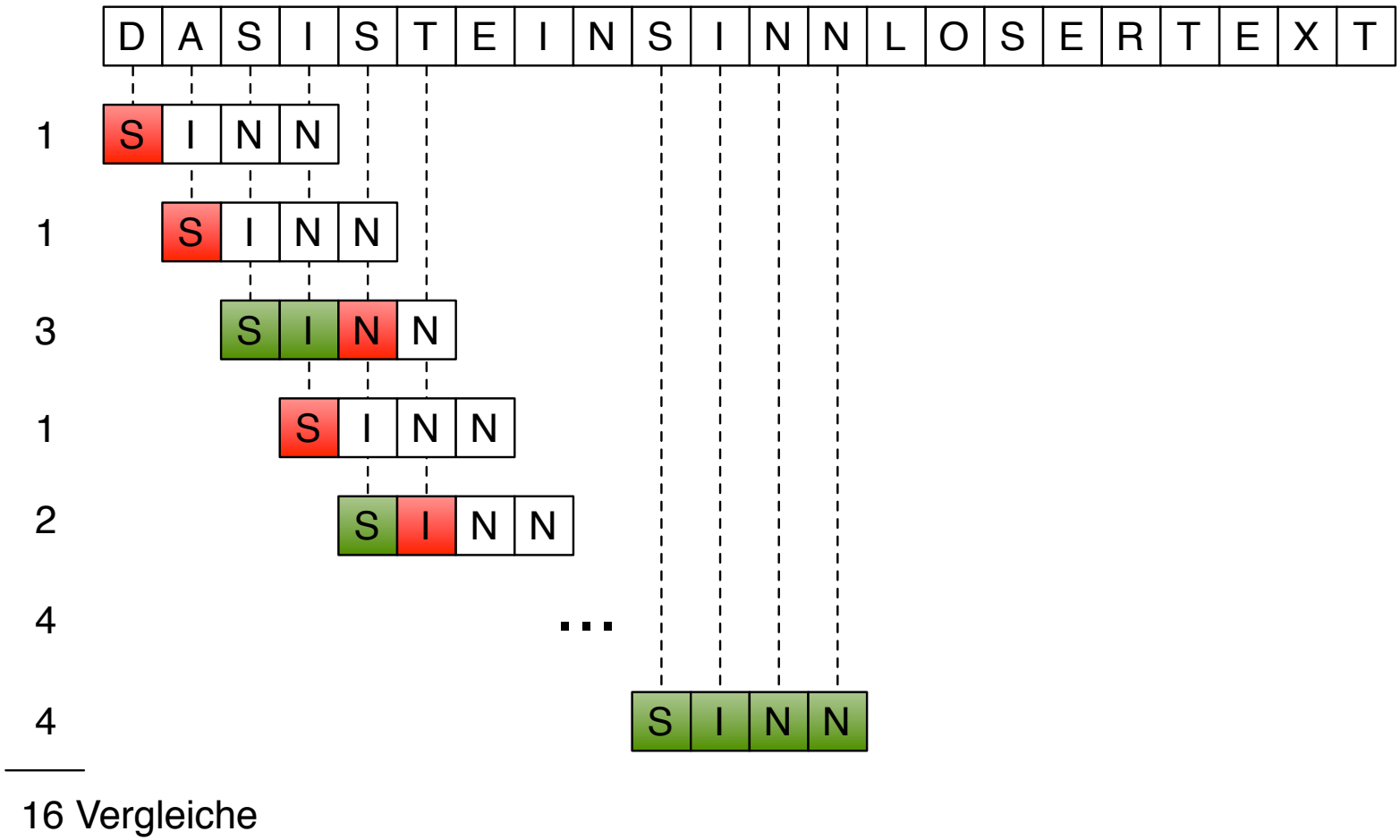
7. Fortgeschrittene Datenstrukturen

8. Such-Algorithmen
   - Lineare Suche
   - Binäre Suche
   - Binäre Suchbäume
   - Balancierte Suchbäume
   - Suchen mit Hashtabellen
   - 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
```

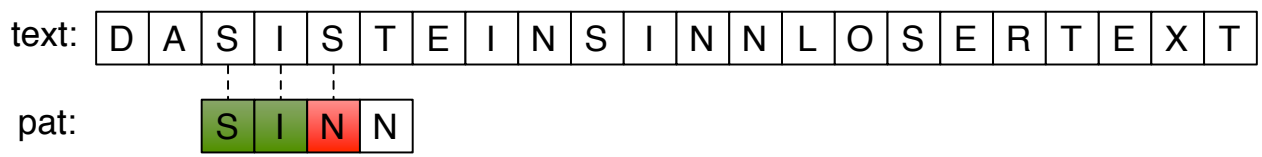
- Komplexität: $O((n-m)m) = O(nm)$

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 *shift*

Alphabet und Wörter

Wörter

Ein Alphabet Σ ist eine endliche Menge von Symbolen. Ein Wort w der Länge n über Σ ist eine endliche Folge von Symbolen

$$w = w_1 \cdots w_n, \quad w_i \in \Sigma,\ i = 1, \ldots, n.$$

Beispiel:

- Alphabet $\Sigma = \{$ `a,...,z, A,...,Z` $\}$
  - Wörter: `Daten`, `Algorithmen`, `aabb`
  - keine Wörter: `über`, `t35t`
- Alphabet $\Sigma = \{$ `0,...,9,A,B,C,D,E,F` $\}$
  - Wörter: `09FF`, `ABC`, `A3E`
  - keine Wörter: `1f`, `1gH`

Das leere Wort wird mit ε bezeichnet.

Präfix und Suffix von einem Wort

Präfix, Suffix

Sei $w = w_1 \cdots w_n$ Wort der Länge n über einem Alphabet Σ.

- Das Wort w' heißt Präfix von w, falls $w' = w_1 \cdots w_l$ für $0 \leq l \leq n$.
- Das Wort w' heißt Suffix von w, falls $w' = w_l \cdots w_n$ für $1 \leq l \leq n + 1$.

Beispiele für Wort $w =$ `Algorithmen`:

- Präfixe von w sind: `A`, `Alg`, `Algorit`, `Algorithmen`
- Suffixe von w sind: `n`, `men`, `gorithmen`, `Algorithmen`

Rand von einem Wort

Rand

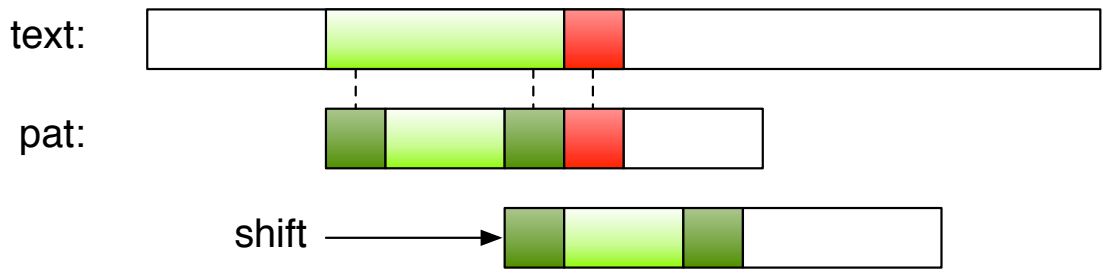
Sei w Wort über einem Alphabet Σ. Ein Wort r heißt Rand von w, falls r sowohl Präfix als auch Suffix von w ist.

Ein Rand r von w heißt eigentlicher Rand, wenn $r \neq w$ und wenn es außer w selbst keinen längeren Rand gibt. Der eigentliche Rand von w wird mit $\partial(w)$ bezeichnet.

Beispiel: Wort $w =$ `aabaabaa` hat folgende Ränder:

- ε
- `a`
- `aa`
- `aabaa` $\leftarrow$ eigentlicher Rand
- `aabaabaa` $= w$

Überspringen beim Suchen



- Idee: verschiebe Pattern so, daß im bereits geprüften Bereich wieder Übereinstimmung herrscht
  - dazu müssen Präfix und Suffix dieses Bereichs übereinstimmen
    $\rightarrow$ eigentlicher Rand

Shift-Tabelle

Shift-Tabelle

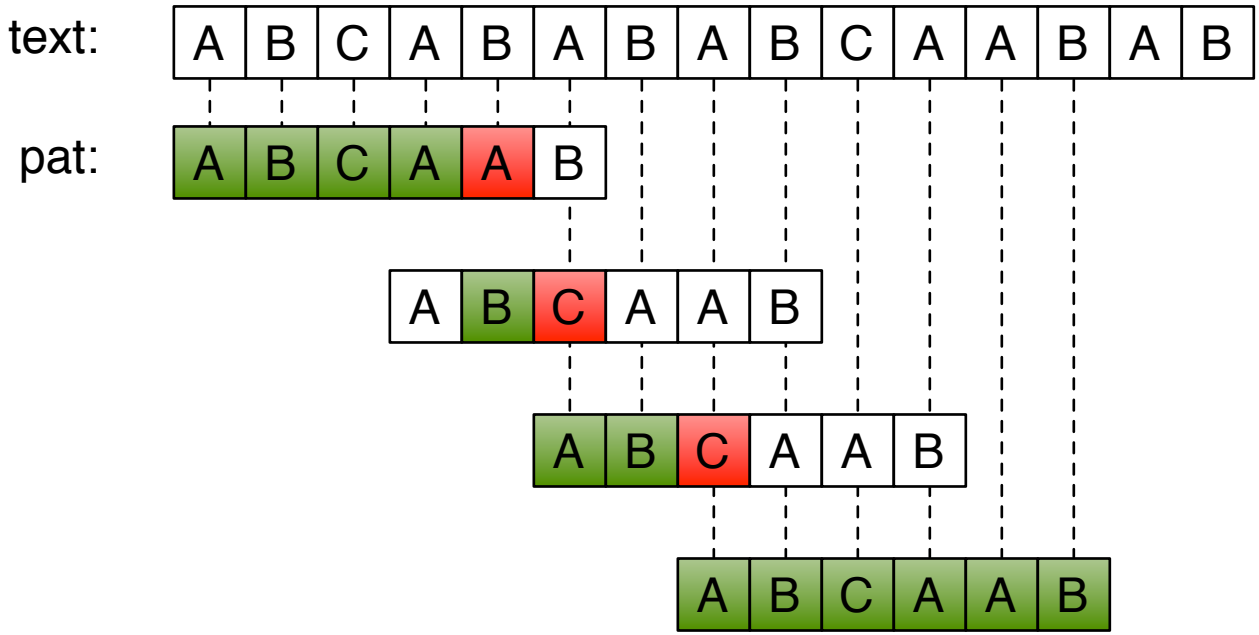
Sei $pat = s_0 \cdots s_{m-1}$ Wort der Länge m. Die Shift-Tabelle von pat gibt die Länge des eigentlichen Randes des Präfixes der Länge j von pat an. Sie hat folgende Gestalt:

$$shift[j] = \begin{cases} -1 & \text{für } j = 0 \\ \left|\partial(s_0 \cdots s_{j-1})\right| & \text{für } j \geq 1 \end{cases}$$

wobei $j = 0, \ldots, m-1$.

- Ist das Präfix der Länge j gematcht, und an Stelle j ein Mismatch, so ist der Shift $j - shift[j] > 0$ korrekt

KMP Algorithmus: Beispiel I



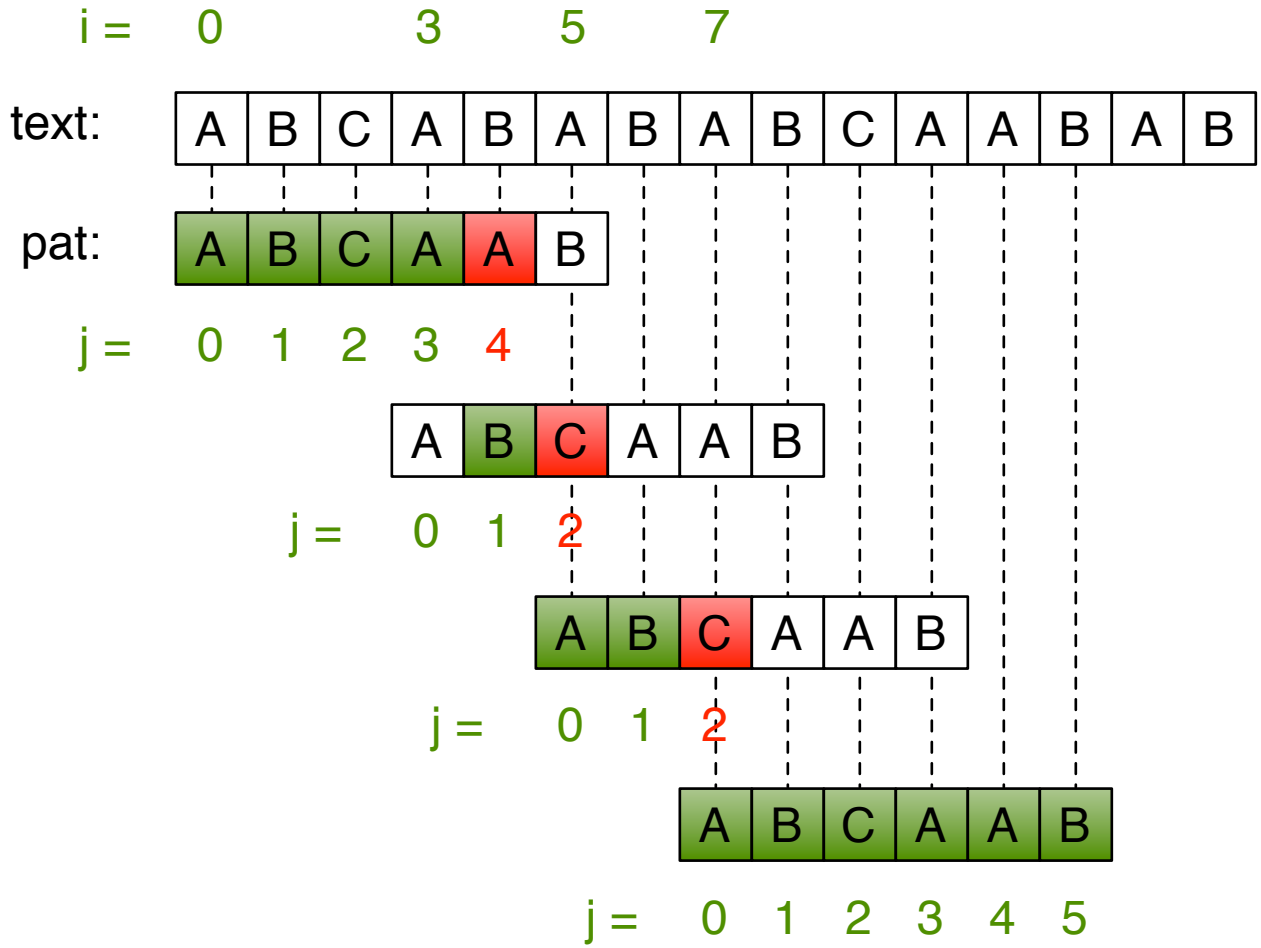
j	0	1	2	3	4	5
$shift[j]$	-1	0	0	0	1	1

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*):

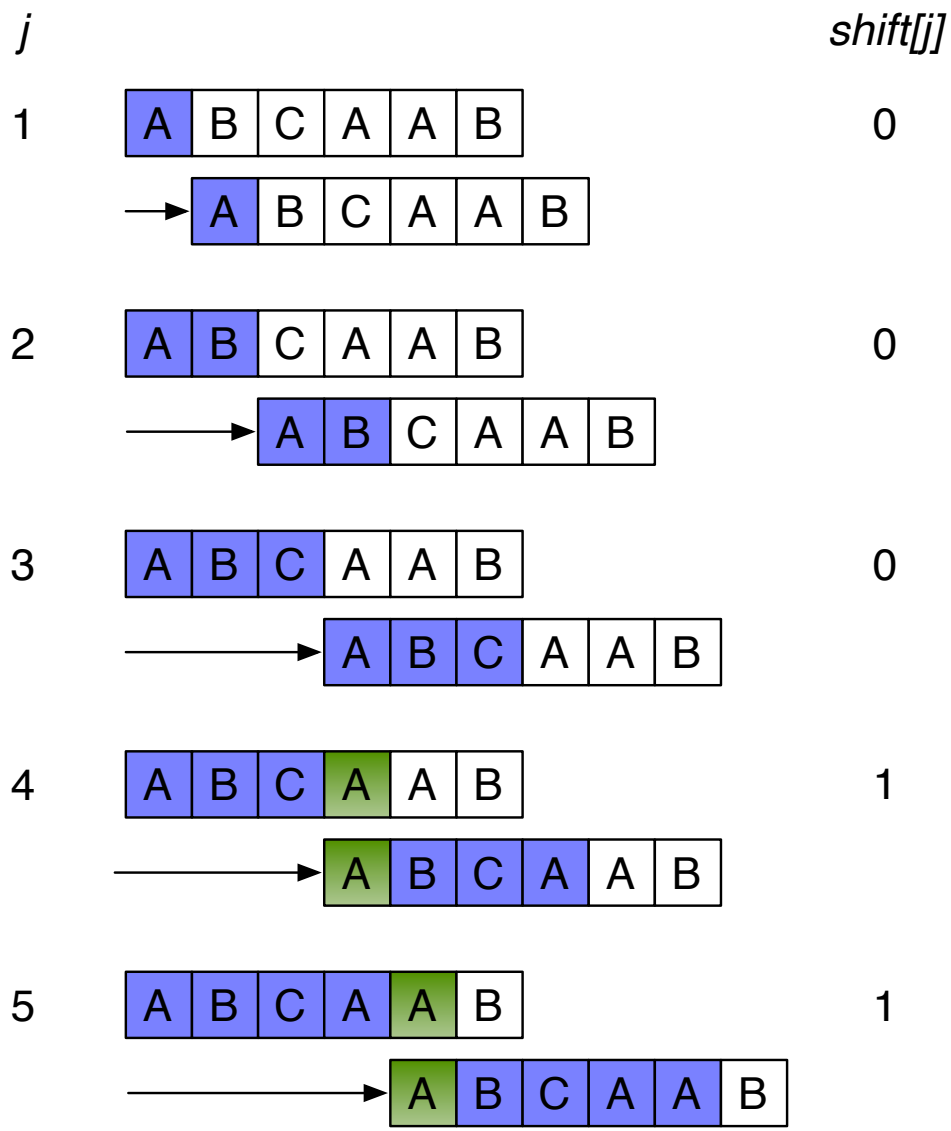
```
initShift(pat); // Initialisierung shift Tabelle, siehe später
i = 0; j = 0;
while (i ≤ n − m) {
    while (text[i + j] == pat[j]) {
        j = j + 1;
        if (j == m)
            return i; // fündig geworden
    }
    i = i + (j − shift[j]); // j − shift[j] ist immer > 0
    j = max(0, shift[j]);
}
return -1; // war wohl nix
```

KMP Algorithmus: Beispiel II



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

Shift Tabelle Beispiel I



KMP Algorithmus: Shift Tabelle

Input: Muster pat der Länge m
initShift(pat):

```
shift[0] = −1;
shift[1] = 0;
i = 0;
for j = 2 to m {
    // hier ist: i == shift[j − 1]
    while (i ≥ 0 && pat[i] ≠ pat[j − 1])
        i = shift[i];
    i = i + 1;
    shift[j] = i;
}
```

KMP Algorithmus: Komplexität

Komplexität von KMP Algorithmus:

- KMPSearch:
  - erfolglose Vergleiche (äußere while-Schleife): maximal $n - m + 1$ Vergleiche
  - erfolgreiche Vergleiche (innere while-Schleife): ingesamt maximal n Vergleiche
  - insgesamt: maximal $2n - m + 1$ Vergleiche
- initShift:
  - erfolgreiche Vergleiche (for-Schleife): maximal $m - 1$ Vergleiche
  - erfolglose Vergleiche (while-Schleife): maximal m Vergleiche
  - insgesamt: maximal $2m - 1$ Vergleiche
- insgesamt: maximal $2n + m$ Vergleiche, also $O(n + m)$
- 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 mit Hashtabellen
- Suchen in Zeichenketten