

# Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2016

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

# 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**  $O(n)$

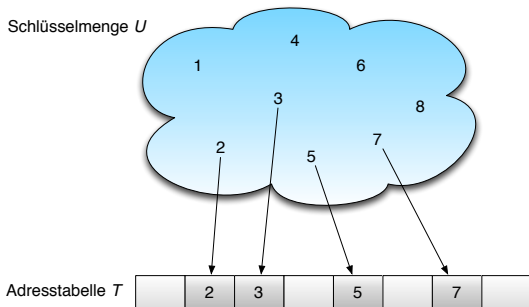
# Adresstabellen

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  $|U|$ .

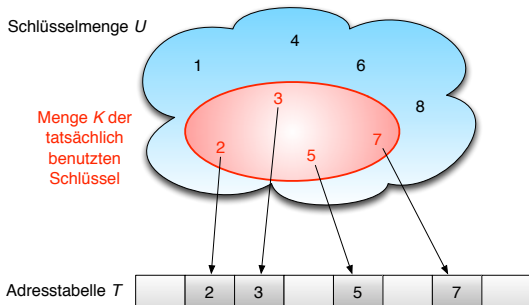
- **Input:** Tabelle  $T$ , Schlüssel  $k$   
**Output:** Element mit Schlüssel  $k$   
**search**( $T, k$ ):  
    **return**  $T[k]$ ;
- **Input:** Tabelle  $T$ , Element  $e$   
**insert**( $T, e$ ):  
     $T[\text{key}(e)] = e$ ;
- **Input:** Tabelle  $T$ , Schlüssel  $k$   
**erase**( $T, k$ ):  
     $T[k] = \text{null}$ ;

Alle Operationen: Laufzeit  $O(1)$

Speicherkomplexität:  $O(|U|)$  → Problem falls  $U$  groß!

# Reduktion von Adresstabellen mittels Hashfunktion

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

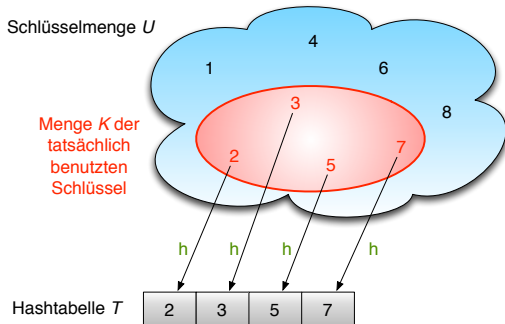


- **Idee:** wende **Hashfunktion**  $h$  auf  $key(e)$  an, um Adresstabelle auf  $|K|$  Elemente zu reduzieren  $\rightarrow$  **Hashtabelle**

# Hashtabellen

## Wörterbuch mit Hashtabelle:

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





# Operationen auf Hashtabellen

Hashtabelle  $T$  mit Länge  $m$  und Hashfunktion  $h$ .

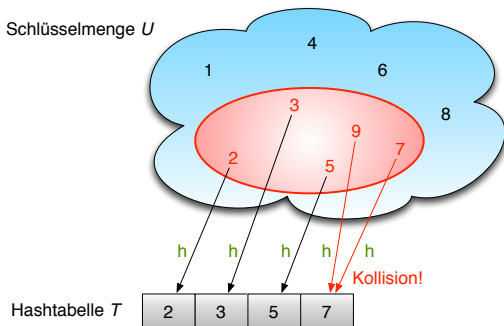
- **Input:** Tabelle  $T$ , Schlüssel  $k$   
**Output:** Element mit Schlüssel  $k$   
**search**( $T, k$ ):  
    **return**  $T[h(k)];$
- **Input:** Tabelle  $T$ , Element  $e$   
**insert**( $T, e$ ):  
     $T[h(\text{key}(e))] = e;$
- **Input:** Tabelle  $T$ , Schlüssel  $k$   
**erase**( $T, k$ ):  
     $T[h(k)] = \text{null};$

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

Speicherkomplexität:  $O(m)$

# Kollisionen bei Hashtabellen

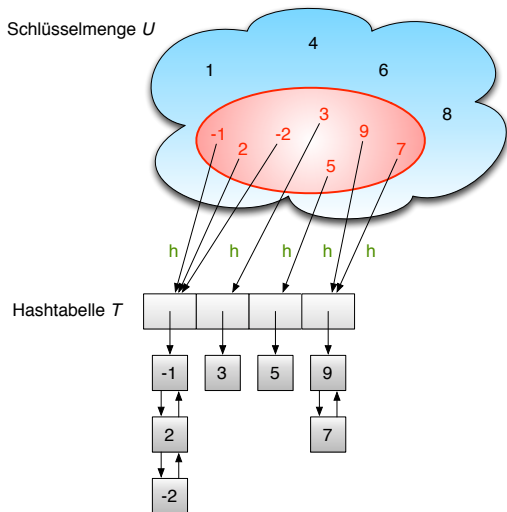
- **Problem:** falls  $h$  nicht injektiv  $\rightarrow$  **Kollision!**
  - $h$  ist nie injektiv da  $|K| < |U|$



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

# Verkettung zur Kollisionauflösung

- 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.

$$h : U \rightarrow \{0, \dots, m - 1\}$$

- **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:

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

- **Beispiel:**  $m = 12$ ,  $k = 100$ , dann ist  $h(k) = 4$ .
- **Problem:** geschickte Wahl von  $m$ 
  - eher schlecht ist  $m = 2^p$  (wertet nur die untersten  $p$  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  $\rightarrow m = 701$  geeignete Primzahl

# Multiplikationsmethode

## Multiplikationsmethode

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

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

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:

$$\mathcal{H} = \{h : U \rightarrow \{0, \dots, m-1\} \mid h \text{ Hashfunktion}\}.$$

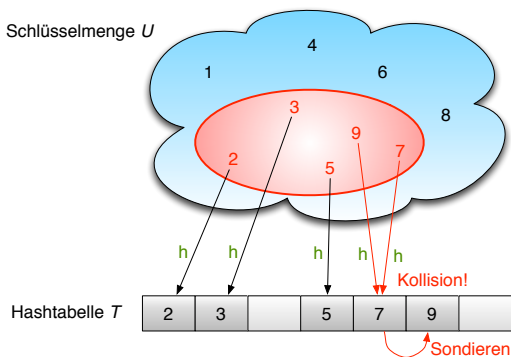
$\mathcal{H}$  heißt **universell**, falls für jedes Schlüsselpaar  $k, l \in U$ ,  $k \neq l$ , die Anzahl der Hashfunktionen  $h$  mit  $h(k) = h(l)$  durch  $|\mathcal{H}|/m$  nach oben beschränkt ist.

- **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 → 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

$$h : U \times \{0, \dots, m - 1\} \rightarrow \{0, \dots, m - 1\}$$

- fordere, daß **Sondierungssequenz**

$$(h(k, 0), h(k, 1), \dots, h(k, m - 1))$$

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

→ 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

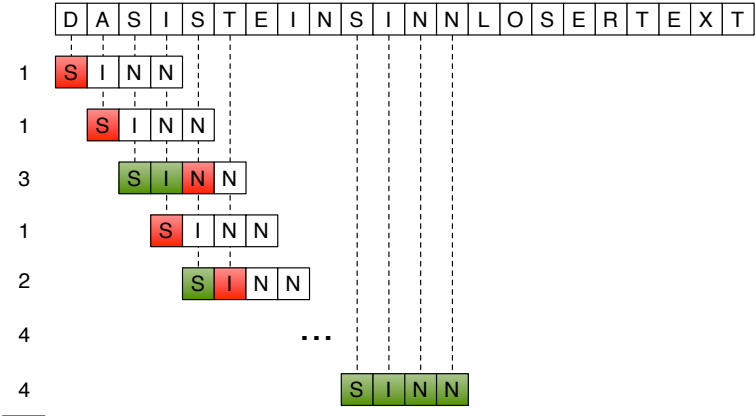
Suchen mit Hashtabellen

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



16 Vergleiche

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

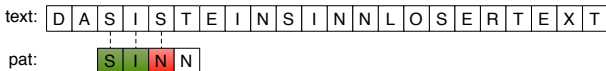


# Knuth-Morris-Pratt Algorithmus

## 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**  $\Sigma$  ist eine endliche Menge von Symbolen. Ein **Wort**  $w$  der Länge  $n$  über  $\Sigma$  ist eine endliche Folge von Symbolen

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

### Beispiel:

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

Das **leere Wort** wird mit  $\varepsilon$  bezeichnet.

# Präfix und Suffix von einem Wort

## Präfix, Suffix

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

- 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  $w = \text{Algorithmen}$ :

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

# Rand von einem Wort

## Rand

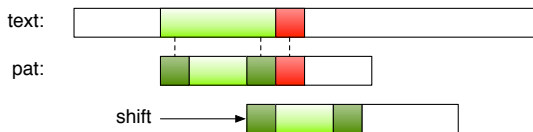
Sei  $w$  Wort über einem Alphabet  $\Sigma$ . Ein Wort  $r$  heißt **Rand** von  $w$ , falls  $r$  sowohl Präfix als auch Suffix von  $w$  ist.

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  $w = \text{aabaabaa}$  hat folgende Ränder:

- $\varepsilon$
- a
- aa
- aabaa ← **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  
→ **eigentlicher Rand**

# Shift-Tabelle

## Shift-Tabelle

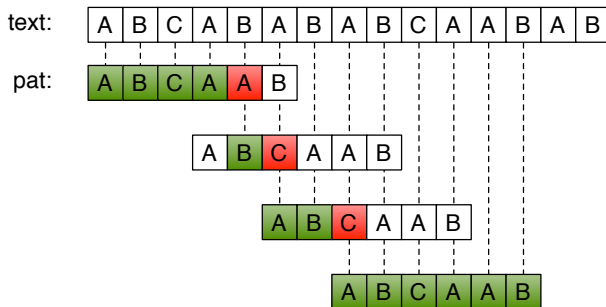
Sei  $pat = s_0 \cdots s_{m-1}$  Wort der Länge  $m$ . Die **Shift-Tabelle von  $pat$**  gibt die Länge des eigentlichen Randes des Präfixes der Länge  $j$  von  $pat$  an. Sie hat folgende Gestalt:

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

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

- 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



<i>j</i>	0	1	2	3	4	5
<i>shift[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*):

**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

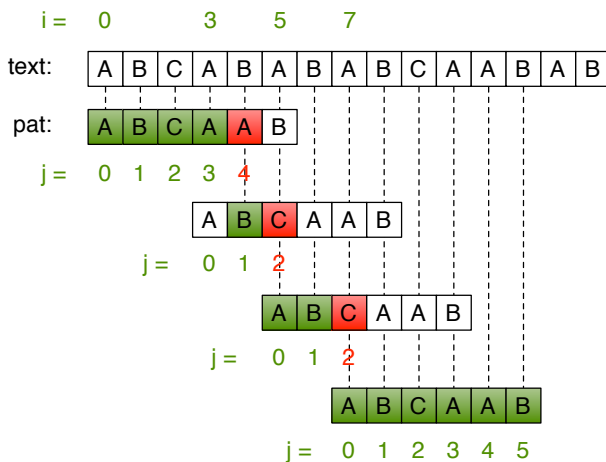
*j* = max(0, *shift*[*j*]);

}

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

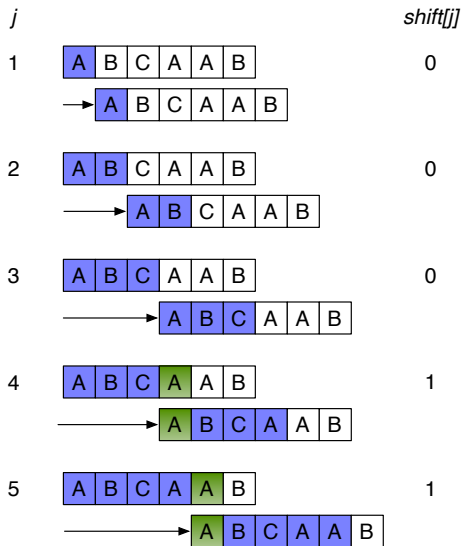


# KMP Algorithmus: Beispiel II



$j$	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** ( $i \geq 0$  &&  $pat[i] \neq pat[j - 1]$ )

$i = shift[i];$

$i = i + 1;$

$shift[j] = i;$

}

# KMP Algorithmus: Komplexität

## Komplexität von KMP Algorithmus:

- KMPSearch:
  - erfolglose Vergleiche (äußere while-Schleife): maximal  $n - m + 1$  Vergleiche
  - erfolgreiche Vergleiche (innere while-Schleife): insgesamt 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