Suchalgorithmen

Suchen

Suchen (searching)

- aus gegebener Menge bestimmtes Datum suchen

- Daten nicht vorbereitet
  - alle vorhandenen Daten ansehen
    - bis zum ersten gefundenen Objekt
  - Aufwand linear mit Anzahl der Daten
- Daten nach Suchschlüssel sortiert
  - Suchen u.U. sehr viel schneller durchführbar

Suchen

Suchen (searching)

- Suchziele: Suche ...
  - irgendein Element mit passendem Schlüssel
  - erstes Element (in der Folge)
  - letztes Element (in der Folge)
  - alle Elemente (Liste von Elementen)
  - Anzahl der Elemente mit passendem Schlüssel
  - Schlüsselbereiche
  - Weiches Suchen
  - ...

Suchen

Suchen (searching)

1. Suchen einzelner Daten in Feldern
2. Teilfelder suchen mit gewissen Eigenschaften
3. Suchen in speziellen Datenstrukturen
   Datenstrukturen zum Suchen speziell aufgebaut
4. Suchen in Texten

Suchen

- Algorithmen zum Suchen in Feldern
  - Daten in beliebiger Reihenfolge in Feld abgelegt
    - Element durch lineares Suchen finden
    - jedes Element in bestimmter Reihenfolge betrachten
  - Datenobjekte sortiert in einem Feld abgelegt
    - Suchen sehr viel effizienter

Suchen

- Einfache Suchverfahren
  - Aufwand für alle Verfahren etwa gleich groß
    - außer Linearem Suchen
  - einfachstes Suchverfahren verwenden
    - Binäres Suchen
  - exponentielles Suchen
    - bei ausgelagerten Daten

Suchen

- Lineares Suchen
  - Suche Feld vom ersten bis zum letzten Element durch, bis passendes Datenobjekt (oder Element) gefunden

  -
```
int LinearesSuchen(CDatenSatz[] Feld, int key){
  for(int i=0;i<Feld.length;i++)
    if(Feld[i].key==key) return i;
  return -1;
}
```
  - Aufwand im Mittel `(Feld.length)/2=N/2`, *O(N)*
  - Suchreihenfolge ändern?

Suchen

- Binäres Suchen
  - Teile Suchintervall in zwei Hälften
  - z.B. suche 17

Suchen

- Binäres Suchen
  - Suchbereich **`[Von,Bis]`** in zwei Teilbereiche **`[Von,Mitte-1]`** und **`[Mitte+1,Bis]`**
  - Wo liegt das gesuchte Element **`X`**?
  - im ersten Teilbereich: **`X.key < Feld[Mitte].key`**
    - Suche weiter im Bereich **`[Von,Mitte-1]`**
  - im zweiten Teilbereich: **`X.key > Feld[Mitte].key`**
    - Suche weiter im Bereich **`[Mitte+1,Bis]`**
  - Genau auf der Mitte: **`X.key == Feld[Mitte].key`**
    - Fertig

Suchen

```
static int binäresSuchen(CDatenSatz[] Feld,int key){
  int von = 0;                          // linker Index
  int bis = Feld.length - 1;    // rechter Index
  int mitte;                          // mittlerer Index
  while (true) { // Ende falls gef. oder Intervall=0
    mitte = (von + bis) / 2;    // mittleres Element
    if (key < Feld[mitte].key) // Schlüssel links
      bis = mitte - 1;
    else if (key > Feld[mitte].key)//Schlüssel rechts
      von = mitte + 1;
    else
      return mitte;                     // Schlüssel gefunden
    if (von > bis)                      // Schlüssel nicht da
      return -1;                        // ... ergibt -1
} }
```

Suchen

- Eigenschaften des Algorithmus Binäres Sortieren
  - Bei jeden Durchlauf halbiert sich Intervalllänge mindestens

$$Länge = Bis - Von + 1, \quad Mitte = \frac{Von + Bis}{2}$$

Linkes Intervall:

$$(Mitte - 1) - Von + 1 = Mitte - Von = \frac{Von + Bis}{2} - Von =$$

$$= \frac{Bis - Von}{2} \leq \frac{Bis - Von + 1}{2} = \frac{Länge}{2}$$

Rechtes Intervall:

$$Bis - (Mitte + 1) + 1 = Bis - Mitte = Bis - \frac{Von + Bis}{2} =$$

$$= \frac{Bis - Von}{2} \leq \frac{Bis - Von + 1}{2} = \frac{Länge}{2}$$

Suchen

- Eigenschaften des Algorithmus Binäres Sortieren
  - Maximale Anzahl von Schleifendurchläufen

$$Feldlänge = N$$

$$Intervalllänge\ nach\ k\ Durchläufen \leq \frac{N}{2^k} < 1$$

$$N < 2^k$$

$$\log_2 N < k$$

$$k = \lceil \log_2 N \rceil + 1$$

Suchen

- Eigenschaften des Algorithmus Binäres Sortieren
  - Wie erkennt das Programm, dass das gesuchte Element nicht in dem Feld vorkommt?
  - Was geschieht, wenn die Länge des Feldes 1, 2 oder 3 ist?
  - Zeigen Sie, dass ein gesuchtes Element niemals außerhalb des Bereichs **`[von,bis]`** liegen kann, wenn es sich einmal in diesem Bereich befindet.
  - Zeigen Sie, dass das Programm **`mitte`** zurück gibt, sobald einmal **`Feld[mitte].key==key`**.
  - Ist das Programm in jeder Situation sicher?

Suchen

- Fibonacci-Suchen
  - Suchbereich asymmetrisch wählen
  - F_n n-te Fibonacci-Zahl
    - Suchbereich Länge F_n
    - Teilbereiche Längen F_{n-1} und $F_{n-2} = F_n - F_{n-1}$.
  - Teilung durch Subtraktion berechnen
  - keine Division
  - Laufzeit nicht effizienter als Binäres Suchen
  - Fibonacci-Zahlen
    - Algorithmus wird komplizierter

Suchen

```
int fibonacciSuchen(CDatenSatz[] Feld, int key){
  int von = 0,
      bis = Feld.length - 1, // = ende
      mitte,
      fib1=1,
      fib2=1;
  while(fib1<bis) { // berechne Fibonacci-Zahlen
    fib1 += fib2;
    fib2  = fib1-fib2;
  }
  if(key<Feld[fib2].key) bis = fib2;// wähle Bereich [0,fib2]
  else                   von = bis-fib2;// Bereich [bis-fib2,ende]
  while (true) { // 0,1,von,3,mitte,5,6,bis,8,9
    fib2 = fib1-fib2; // fib1 21, 13, 8
    fib1 = fib1-fib2; // fib2 13,  8, 5
    mitte = von+fib1;     //
```

Suchen

```
  while (true) {        // 0,1,von,3,mitte,5,6,bis,8,9
    fib2 = fib1-fib2; // fib1 21, 13, 8, F_{n-2}=F_n-F_{n-1}
    fib1 = fib1-fib2; // fib2 13,  8, 5, F_{n-1}=F_n-F_{n-2}
    mitte = von+fib1; // nächste Teilung
    if (mitte>=Feld.length || fib1<=0) return -1;
      // Schlüssel > max., Intervall == 0
    if (key < Feld[mitte].key) {
      // Falls gesuchter Schlüssel links von mitte,
      bis = mitte;
    } else if (key > Feld[mitte].key) {
    // Falls gesuchter Schlüssel rechts von mitte
      von = mitte;
      fib2 = fib1 - fib2; // fib1 21, 13, 8
      fib1 = fib1 - fib2; // fib2 13,  8, 5
    } else return mitte; // gesuchter Schlüssel gefunden
    if (von > bis) return -1;//Schlüssel nicht vorhanden
} }
```

Suchen

- Suchen im Anfangsbereich eines Feldes
  - Wenn Feld sehr groß ist
  - gesuchtes Datum vermutlich am Anfang des Feldes
- Durch exponentielle Vergrößerung zunächst obere Grenze des Suchbereichs ermitteln
  - Anfang eines Felds im Speicher; Rest ausgelagert
    - u.U. im Mittel sehr viel effizienter sein.

Suchen

- Binäres Anfangssuchen

```
int binäresAnfangsSuchen(CDatenSatz[] Feld, int key) {
  int von,
      bis = 1,
      mitte;  // mitte nicht initialisieren
  while (bis < Feld.length) {
    if (Feld[bis].key >= key) break;
    bis *= 2;
  }
  von = bis/2;
  if (bis >= Feld.length)
    bis = Feld.length - 1;
  ...
```

Suchen

- Fibonacci Anfangssuchen

```
int fibonacciAnfangsSuchen(CDatenSatz[] Feld, int key){
  int von = 0, bis = Feld.length - 1, mitte;
  int fib1=1, fib2=1;
  while(fib1<bis) { // berechne Fibonacci-Zahlen
    fib1 += fib2;
    fib2  = fib1-fib2;
    if (fib1>=bis) { // am Feldende höre auf
      von = bis-fib2;
      break;
    }
    if (key<=Feld[fib1].key) {//Schlüssel<Feld, höre auf
      bis = fib1;
      von = fib2;
      fib2 = fib1 - fib2; // fib1 21, 13, 8
      fib1 = fib1 - fib2; // fib2 13,  8, 5
      break;
  } } ...
```

Suchen

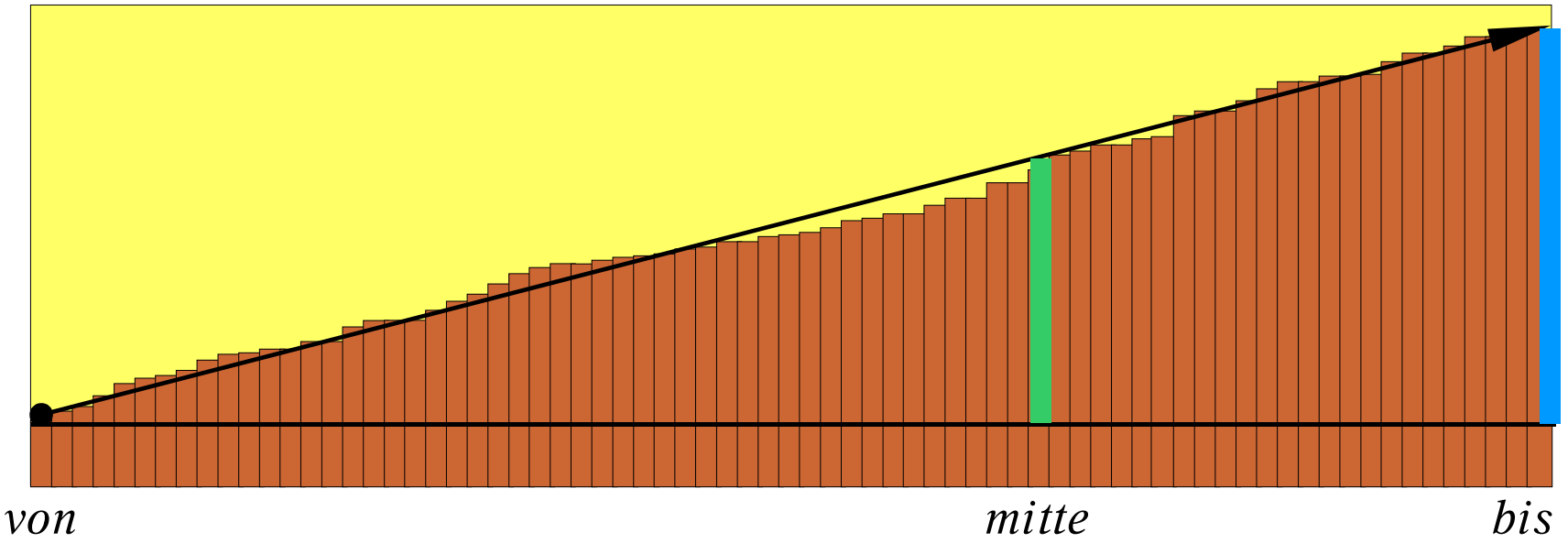
- Interpolationssuchen
  - Falls Feldindex proportional Suchschlüssel
    - Faktor bestimmt Feldelement aus Formel
      Feldindex* = *Faktor* · *Suchschlüsselwert
    - Bestimmt beim Binären Suchen mittleren Wert `Mitte` proportional zum gesuchten Element im Suchbereich
      Mitte =

Suchen

- Interpolationssuchen

```
index = (int) ( ( (bis - von) / (float)
      (Feld[bis].key-Feld[von].key))*(key-Feld[von].key));
```

$$mitte = von + \frac{key - Feld[von].key}{Feld[bis].key - Feld[von].key} \times (bis - von)$$



Suchen

- Interpolationssuchen

```
int interpolationssuchen(CDatenSatz[] Feld, int key) {
  int von = 0, bis = Feld.length - 1, mitte;
  int index = (int) ( ( (bis - von) / (float)
       (Feld[bis].key-Feld[von].key))*(key-Feld[von].key));
  mitte = Math.min(Math.max(von, von + index), bis);
   // Setze mittleres Element
  if (key < Feld[mitte].key) bis = index;
  else if (key > Feld[mitte].key) von = index;
  else return mitte; // sonst gesuchter Schlüssel gefunden
  while (true) { // kehre aus Schleife zum Aufruf zurück
    index = (int) ( ( (bis - von) / (float)
     (Feld[bis].key-Feld[von].key))*(key-Feld[von].key));
     // Berechne proportionalen Abstand
    mitte = Math.min(Math.max(von, von + index), bis);
     // Setze mittleres Element
```

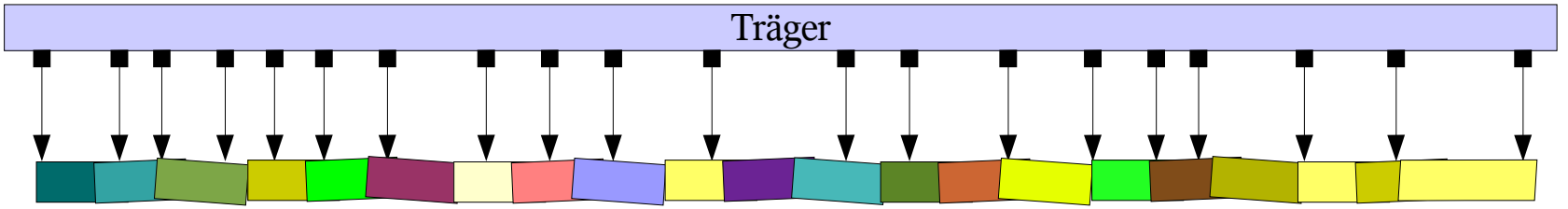
Suchen

- Interpolationssuchen

```
while (true) { // Durchlaufe Schleife
  index = (int) ( ( (bis - von) / (float)
   (Feld[bis].key-Feld[von].key))*(key-Feld[von].key));
   // Berechne proportionalen Abstand
  mitte = Math.min(Math.max(von, von + index), bis);
   // Setze mittleres Element
  if (key < Feld[mitte].key)
    // Falls gesuchter Schlüssel links von mitte
    bis = mitte - 1;
  else if (key > Feld[mitte].key)
    // Falls gesuchter Schlüssel rechts von mitte
    von = mitte + 1;
  else return mitte; //sonst gesuchten Schlüssel gefunden
  if (von > bis) return -1;
        // Falls von > bis ist Schlüssel nicht vorhanden.
} }
```

Suchen von Teilfeldern

- Statt einzelner Elemente in Feldern suchen
- Teilfelder eines Feldes ermitteln
  - bestimmte Eigenschaft in Bezug auf gesamtes Feld
  - Beispiel: Teilfeld Summe der Werte maximal
    - Belastbarkeit eines tragenden Teils
      Brücke, Ankerbefestigung eines Gebäudes
    - Differenz Massen; Tragkräfte der stützenden Träger
    - Bereich maximaler Belastung



Suchen von Teilfeldern

- Teilfelder eines Feldes ermitteln
  - bestimmte Eigenschaft in Bezug auf gesamtes Feld
  - Investition
    - Bau einer größeren Anlage, Straße
    - Kapital zur Verfügung stellen
    - Differenz der Kapitalzuflüsse aus Finanzierung Kapitalabflüsse zur Kostendeckung
    - Finanzierungsbedarf

Suchen von Teilfeldern

- 1. Lösungsansatz
  - Berechne die Summe alle möglichen Teilfelder
  - Aufwand?
    N Felder: [1..N]
    N-1 Felder: [2..N-1]
    1 Feld: [N..N]
    = N·(N-1)/2
    Aufwand quadratisch?
    Oder kubisch?

Suchen von Teilfeldern

- 2. Lösungsansatz
  - Berechne Summe sukzessiv über gesamtes Feld
    - Wenn Zwischensumme negativ: Setze auf null
      - Negative Summen tragen nichts zur folgenden Summe bei
    - Wenn Zwischensumme maximal: merke Grenzen!
      - Größte Zwischensumme ist letzte Summe

Suchen von Teilfeldern

```
void teilfeld(Graphik graphik, Daten[] Feld) {
  int sum = 0, index, von = 0, bis = 0, summe = 0,maxSumme = 0;
  for (index = 0; index < Feld.length; index++){ // Alle Werte
    summe += Feld[index].key;      // Add. Wert auf Summe
    if (summe < 0) {
      summe = 0; // Falls Summe negativ, setze diese auf null
      von = index+1;
    } else if (summe > maxSumme) {// Falls Summe größte, merke
      bis = index;                 // Ende von größter Summe
      maxSumme = summe;            // neue größte Summe
  } }
  summe = maxSumme;
// Finde Anfang des kleinsten Feldes mit größter Summe
  for (index = bis; index >= 0; index--){// Gehe rückwärts
    summe -= Feld[index].key;      // subtrahiere von MaxSumme
    if (summe == 0) {// Falls Summe auf null, Anfang gefunden
      von = index;
      break;
} } }
```

Hashen

Suchen auf Feldern durch Hashen

- Suchzeit bislang *O(F(N))* (von *N* abhängig)
  - große Datenmengen langsamer bearbeitet als kleine
- Bearbeitungsaufwand nahezu O(1)
  - unabhängig von der Anzahl der Daten
  - auch für große Datenmengen schnelle Verarbeitung
- →**Hashen**
  - Suchzeit nach eingefügten Daten ungefähr konstant

Hashen

- Kernidee: siehe proportionales Suchen
  - aus Suchschlüssel Feldindex berechnen
  - zumindest angenähert
  - Bei 'Kollision' entsprechende Behandlung
- Perfektes Hashen
  - sämtliche Schlüssel
  - möglichst gleichmäßig über Adressraum streuen
  - verschiedene Schlüssel auf verschiedene Adressen
  - Nur in besonderen Situationen erreichbar

Hashen

- Kollision
  - unterschiedliche Schlüssel → gleicher Feldindex
- 2 Verfahren zur Kollisionsbehandlung

  1) Interne Kollisionsbehandlung
     - kollidierender Datensatz in Feld gespeichert
     - in gewissen Abstand zur errechneten Adresse

  2) Externe Kollisionsbehandlung
     - Datensätze in externer Liste angeordnet
- Feld der Länge Anzahl für Elemente
  - Schlüssel sind Zahlen oder Texte

Hashen

- Hashfunktion
  - Abbildung
    - berechnet aus Schlüssel Feldindex
  - verteilt Schlüssel gleichmäßig über Feldindizes
  - hängt von konkreter Schlüsselmenge ab
  - Gleichmäßigkeit in der Regel nicht garantierbar
  - Wahrscheinlichkeitsaussagen

Hashen

- Hashfunktion
- Schlüssel seien Zahlen
  - z.B. Divisionsrest-Funktion
    - Adresse(Zahl) = Zahl modulo Anzahl
  - Anzahl sollte Primzahl
    - gute Streuung der Adress-Werte
    - in speziellen Anwendungen auch andere Werte geeignet

Hashen

- Hashfunktion
- Schlüssel seien Zahlen
  - Anzahl Zweierpotenz
    - Berechnung durch Abtrennung
    - Verwendung der hinteren Stellen des Schlüsselwerts
    - brauchbare Hashfunktion
  - Zahlenwerte gleichverteilt aus größerem Bereich
  - gerade Anzahl
    - gerade Schlüssel gerade Feldindizes,
    - ungerade Schlüssel ungerade.
    - kann zu schlechter Feldindex-Verteilung führen

Hashen

- Schlüssel seien Zahlen
  - höhere Stellen der Schlüssel nicht berücksichtigt
    - Häufungen der Adressen
    - höheren Stellen zu niederen addieren
    - Änderung in höherer Stelle auch Änderung in niederer
  - Beispiel
    - Anzahl = 10
    - Abschneiden: $1 \rightarrow 1$, $41 \rightarrow 1$, $171 \rightarrow 1$, $901 \rightarrow 1$
    - Quersumme: $1 \rightarrow 1$, $41 \rightarrow 5$, $171 \rightarrow 9$, $901 \rightarrow 0$
  - reine Gleichverteilung der Schlüssel
    - keine besseren Ergebnisse

Hashen

- Hashfunktion
- Schlüssel seien **Texte**
  - einzelne der Buchstaben berücksichtigen
  - Liste der Bezeichner in Programm
    - Bezeichner wie x1, x2, x3
    - nur erstes Zeichens → schlechte Verteilung der Adressen
    - Häufungen (clustering) von Schlüsseln
      → verschiedene Feldindizes

Hashen

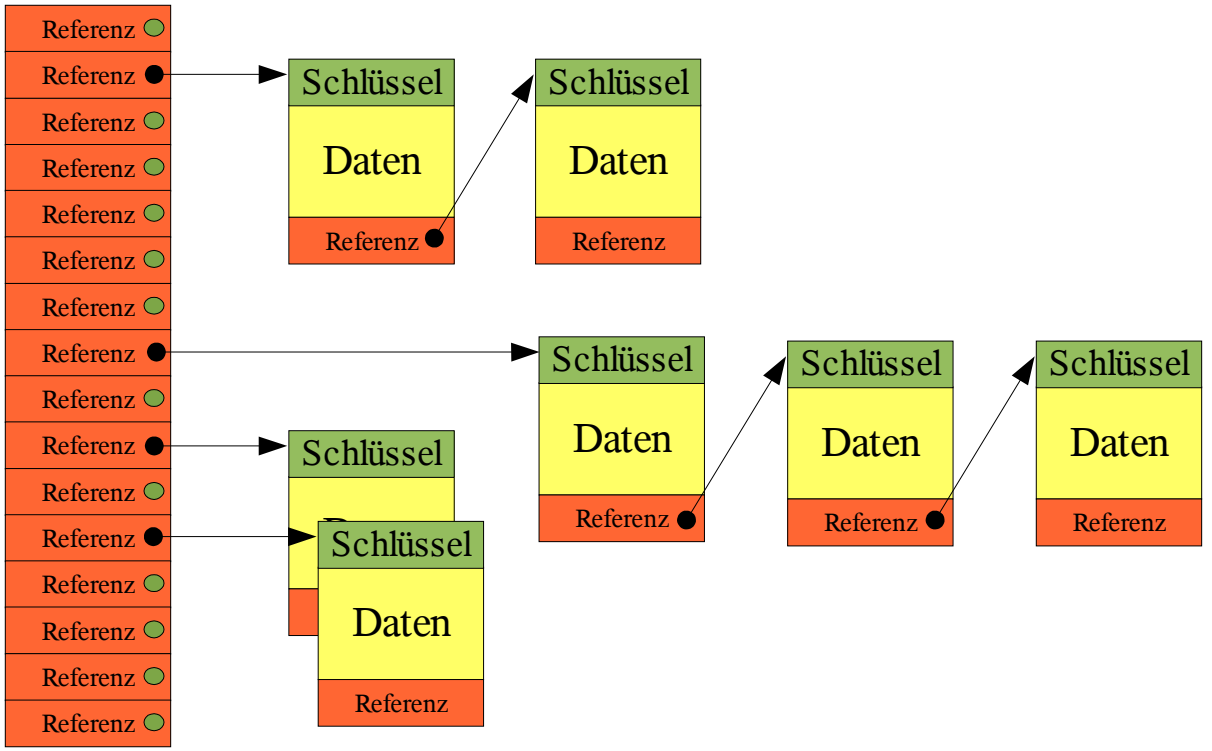
- Schlüssel seien **Texte**
  - mehrere Zeichen
    - ASCII-Kodierungen addieren
      - Großer Adressraum (10000 Adressen)
    - nebeneinanderstehende Zeichen als Zahl auffassen
    - Zeichen als Zahlen addieren
    - gewisse Zeichen eines Textes (z.B. 1., 3. und letztes)
      - Divisionsrest
  - viele andere Hashfunktionen in Literatur
    - Beurteilung nur nach statistischen Kriterien
    - wenige theoretische Begründungen für anderes Verfahren
    - Empirische Untersuchungen

Hashen

- **Externes Hashen**
  - Hashverfahren mit externer Kollisionsbehandlung
- sämtliche Datensätze in eigener Liste speichern
  - Vorteil, falls Datensätze bereits Verkettung besitzen
- heißt Externe Kollisionsbehandlung
  - da Daten nicht vollständig in Feld gespeichert
- Daten als Referenzen auf Datensätze
  - Feld nur Referenzen, keine Objekte
  - Allgemeinere Behandlung von Feldlisten

Hashen

- **Externes Hashen**



Hashen

- **Externes Hashen**
  - Einfügen eines Datensatzes relativ einfach
  - Entfernen eines Datensatzes relativ einfach
  - Zugriff auf Datensatzes relativ einfach
  - zuletzt gefundenen Datensatz nach vorne ketten
    - beim nächsten Suchen schneller finden
  - Aufwand = $O(1)+O(L)$ (L=Listenlänge)
  - keine Kollision, Einfügen am Anfang der Liste
    - Aufwand = $O(1)$
  - Schlimmster Fall: $O(N)$,
    - Nur bei sehr schlechter Hashfunktion

Externes Hashen

```
class HashDatensatz extends CDatenSatz {
  public HashDatensatz(int key) {
    super(key);
  }
  public HashDatensatz (int key,
    HashDatensatz ref) {
    super(key);
    referenz = ref;
  }
  HashDatensatz referenz = null;
}
```

Externes Hashen

```
void EinfügenExtern(HashDatensatz Datum){
  int Index = HashKey(Datum.key);// suche Index
  Datum.referenz = Feld[Index];//setze refer.
  Feld[Index] = Datum; // kette Element ganz
}                      // vorne ein

public HashDatensatz FindenExtern(int key) {
  int Index = HashKey(key); // suche Index
  HashDatensatz datum = Feld[Index];// 1.Elem.
  while(datum!=null) {// suche Elem.
    if(datum.key==key) return datum;//gefund.
    datum = datum.referenz;// nächstes Element
  }
  return null;
}
```

Externes Hashen

```
public HashDatensatz LöschenExtern(int key) {
  int Index = HashKey(key);
  HashDatensatz datum = Feld[Index],
                datum2 = null;
  while(datum != null) {
    if(datum.key == key) {// gefunden
      if (datum2 == null) // Elem. ganz vorne
        Feld[Index] = Feld[Index].referenz;
      else // Elem. nicht ganz vorne, umketten
        datum2.referenz = datum.referenz;
      return datum; // return gefundenes Elem.
    }// nicht gefunden, setze neue Referenzen
    datum2 = datum; datum = datum.referenz;
  } }// nichts gefunden, also gib null zurück!
  return null; }
```

Hashen

- **Internes Hashen**
  - Datensätze in Feld speichern
  - bei Kollision freien Speicherplatz finden
- Sondierungsfolge
  - suche Index für freies Feld
- verschiedene Verfahren
  - Lineares Sondieren
  - Quadratisches Sondieren (0, 1, 4, 9, 16 usw.)
  - Zufälliges Sondieren: reproduzierbare Zufallsfunktion
  - Double-Hashing: neue Hashfunktion

Hashen

- **Internes Hashen**
- Sondierungsfolge
  - suche Index für freies Feld
- Verfahren vielfältig in Literatur untersucht
  - quadratisches Sondieren günstig
  - nur jeder zweite Feldeintrag
- Ein Speicherplatz in mehreren Sondierungsfolgen
  - Belegter Platz bedeutet nicht, dass Sondierungsfolge nicht zu Ende

Hashen

- Löschen in Sondierungsfolgen
  - zu löschende Einträge nicht physisch entfernen
  - als gelöscht kennzeichnen
  - bei Suche wie Eintrag behandeln
  - beim Schreiben wieder belegen
- Aufwand
  - mindestens wie bei externem Hashen
  - neue Elemente nur ans Ende der Liste
  - gelöschte Elemente verursachen weiterhin Aufwand
  - Vertauschen von Elementen kaum möglich

Internes Hashen

```
// lineares Sondieren
int linearesSondieren(int index) {
  for (int i = 0; i < Feld.length; i++) {
    if (Feld[index] == null || Feld[index].key<0)
      return index;
    index = (index + 1) % Feld.length;
  } return -1; }
// quadratisches Sondieren
int NächsterHashKeyQuad(int index) {
  int add = 1;
  for (int i = 0; i < Feld.length; i++) {
    if (Feld[index] == null || Feld[index].key<0)
      return index;
    index = (index + add) % Feld.length;
    add += 2;
  }  return -1;}
```

Internes Hashen

```
int linearesSondieren(int index, int key) {
  for (int i = 0; i < Feld.length; i++) {
    if(Feld[index]==null) break
    else if(Feld[index].key==key) return index;
    index = (index + 1) % Feld.length;
  } return -1;
}
int quadratischesSondieren(int index, int key) {
  int add = 1;
  for (int i = 0; i < Feld.length; i++) {
    if(Feld[index]==null) break
    else if(Feld[index].key==key) return index;
    index = (index + add) % Feld.length;
    add += 2;
  } return -1;
}
```

Internes Hashen

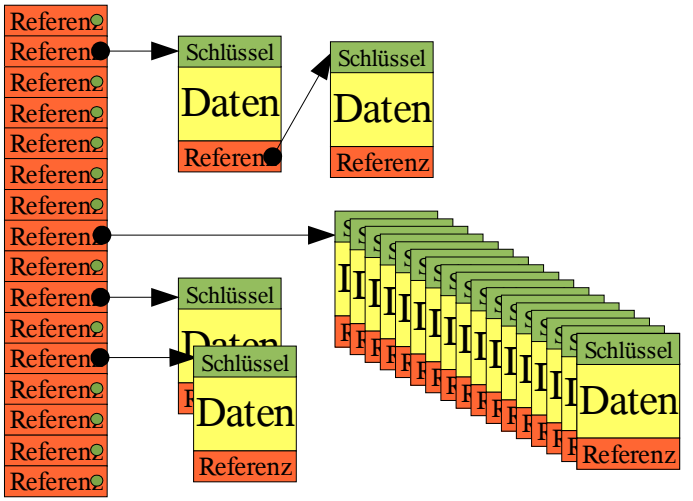
```
int Einfügen(HashDatensatz datum) {
   int key = datum.key;
   int index = HashKey(key);
   index = NächsterHashKey(index);
   if(index>=0) Feld[index] = datum;
   return index;
}

HashDatensatz Finden(int key) {
   int index = HashKey(key);
   index = NächsterHashKey(index, key);
   if(index>=0) return Feld[index];
   return null;
}
```

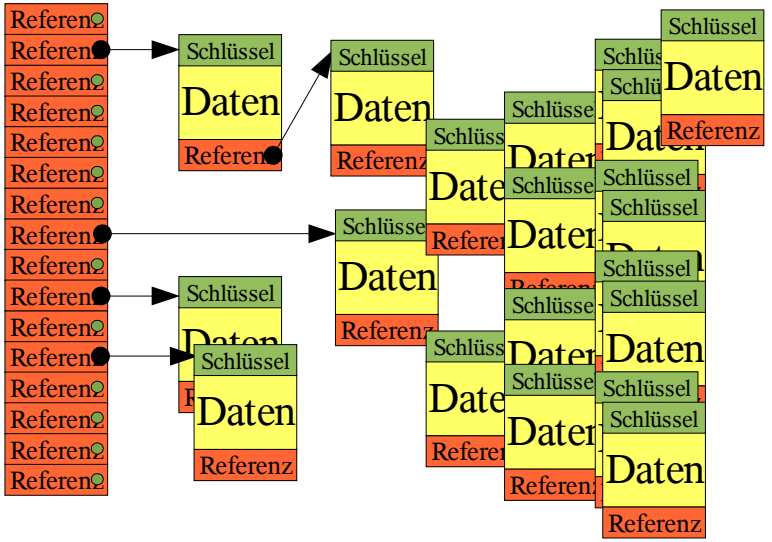
Internes Hashen

```
HashDatensatz Löschen(int key) {
  int index = HashKey(key);
  index = NächsterHashKey(index, key);
  HashDatensatz datum = null;
  if(index>=0) {
    datum = Feld[index];
    Feld[index] = nullDatensatz;
  }
  return datum;
}
```

Hashen

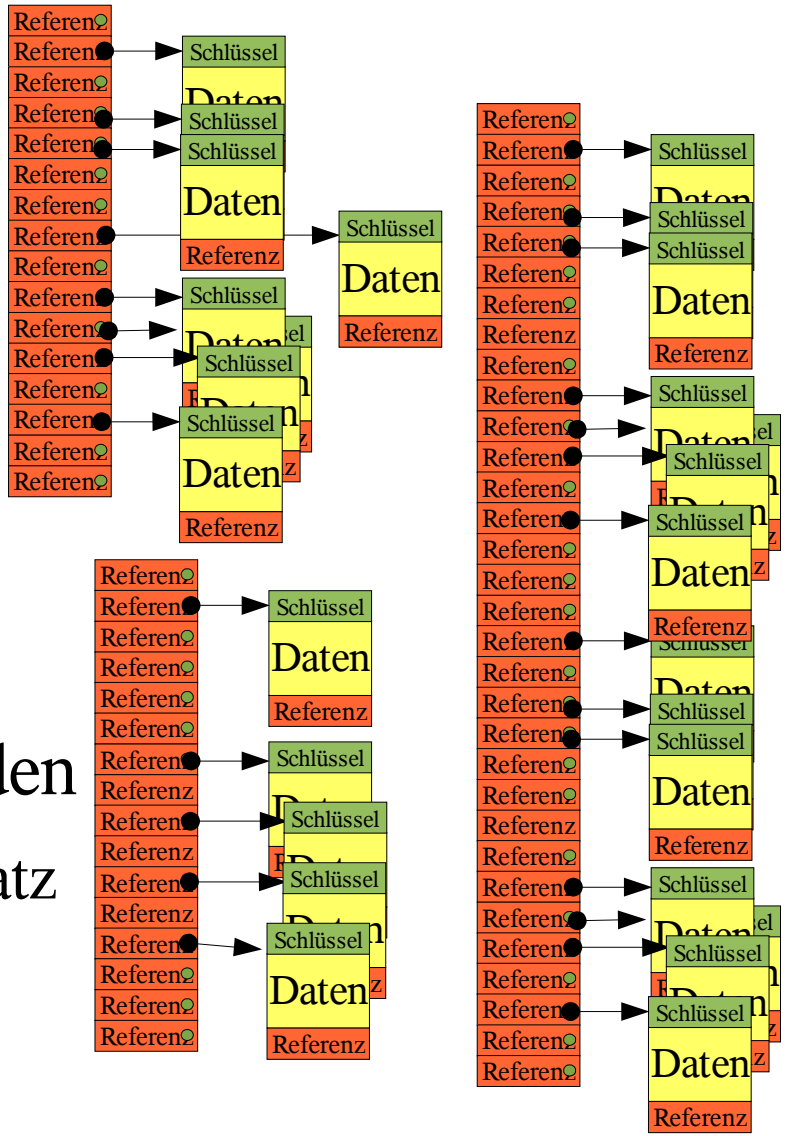
- Dynamisches Hashen
  - Feld dynamisch vergrößern
  - interne Kollisionsbehandlung
    - feste maximale Anzahl an Datensätzen
  - externe Kollsionsbehandlung
    - Flexibler
    - lange Überlauflisten → lange Bearbeitungszeiten
    - Listenstruktur verbessern
      - z.B. ein binärer Baum





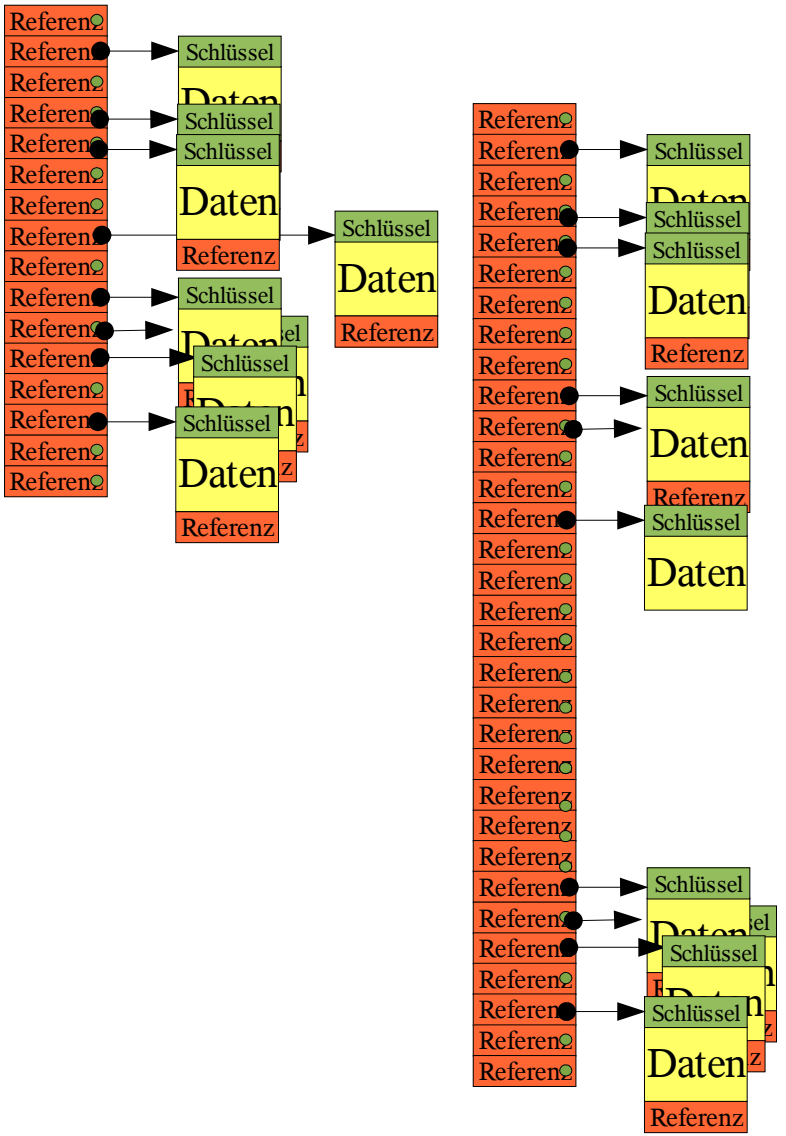
Hashen

- Dynamisches Hashen
  - alle Daten umorganisieren
  - neuen Bereich hinzufügen
- Hashfunktion ändern
  - alte Datensätze berücksichtigen
  - neuen in anderem Bereich abgelegt
  - alte und neue Hashfunktion verwenden
    - findet die erste nicht gesuchten Datensatz
      → andere verwenden



Hashen

- Dynamisches Hashen
  - Datenmengen in mehreren Schritten erweitern
  - entsprechend viele Hashfunktionen
  - Speicherbereich verdoppeln
    - „exponentielle“ Erweiterung
  - Datenmengen auf externe Speicher auslagern
    - möglichst hohe Trefferquote
    - Adressen von externen Blöcken im Feld gespeichert



Suchen in Texten

- Teiltext in längerem Text suchen
  - Teiltext ≡ Suchtext
  - Zu durchsuchender Text ≡ Text
- Texteditoren
- Suchtext nach verschiedenen Kriterien
  - Suche das erste Auftreten eines Teiltexts.
  - Suche nur ganze Worte.
  - Beachte/ignoriere Groß- und Kleinschreibung.
  - Suche ähnliche Teiltexte (***weiches Suchen***)

Suchen in Texten

- Verschieden Verfahren
  - Textlänge N
  - Suchtextlänge M
  - Aufwand i.d.R. $O(N+M)+O(N \cdot M)$
- Standardverfahren finden nur identische Texte
- Aufwandsfaktor variiert zwischen $f=1..M$
  d.h. Aufwand $\approx f \cdot O(N+M)$
- Größter Aufwand, wenn Suchtext nicht in Text!

Suchen in Texten

- Direktes Suchen
  - Für jede Stelle s im Text von s = 0..N-M
    - Vergleiche mit Suchtext
    - falls gleich, gebe s als Ergebnis zurück
  - sonst inkrementiere s
  - Falls s>N-M: beende mit Meldung: Nicht gefunden!

Suchen in Texten

- Direktes Suchen

```
int suche(String suchtext) {
  int gefunden=-1; // Index, an dem gefunden
  int bis = text.length()-suchtext.length();
  for(int von=0; von<=bis; von++) {
    gefunden = von; // wenn, dann hier gefunden
    for(int i=0;i<suchtext.length();i++) {
      if(suchtext.charAt(i)!=//suchtext[0,M-1]
         text.charAt(i+von)) {//==text[von,..]
      gefunden = -1; // nicht gleich, also -1
      break;         // noch mal mit von++
    } }
    if(gefunden>=0) return gefunden;//gefunden
  } return -1;       // nicht gefunden!
} // ENDE:: suche(String suchtext)
```

Suchen in Texten

- Direktes Suchen (Implementierung n. Sedgewick)

```
int suche(String suchtext) {//
  int vonText = 0, vonSuch = 0;//Indizes
  while(vonText < text.length() &&
        vonSuch < suchtext.length()) {
    if(text.charAt(vonText) ==
                    suchtext.charAt(vonSuch)){
      vonText++; vonSuch++; // Erhöhe Indizes
    } else { //Setze Indizes zurück
      vonText -= vonSuch-1; vonSuch = 0;
    }
  } if(vonSuch>=suchtext.length()) //gefunden
        return vonText-suchtext.length();
    else return -1; // nichts gefunden
} // ENDE:: suche(String suchtext)
```

Suchen in Texten

- Aufwand für direktes Suchen in Texten
- Minimaler Aufwand (Suchtext nicht vorhanden)
  - Erstes Zeichen in Suchtext (Suchtext nicht im Text)
  - Aufwand: *N* Vergleiche! *O(N)*
- Maximaler Aufwand (Suchtext nicht vorhanden)
  - Alle Zeichen in Suchtext im Text
    Unterschied erst beim letzten Zeichen des Suchtexts
  - Aufwand: *N·M* Vergleiche! *O(N·M)*
  - Beispiel: Suche **AAAAAAAB** in
    AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA...

Suchen in Texten

- Aufwand für direktes Suchen in Texten
- Mittlerer Aufwand
  - Einfaches stochastisches Modell
    - Alle Zeichen gleichwahrscheinlich: p
  - 1. Zeichen: $N \cdot p$ gleich, $N \cdot (1-p)$ ungleich,
    $N \cdot p + N \cdot (1-p) = N$ Vergleiche
  - 2. Zeichen: $N \cdot p$ Fälle, $N \cdot p^2$ gleich, $N \cdot p \cdot (1-p)$ ungleich
    $N \cdot p^2 + N \cdot p \cdot (1-p) = N \cdot p$ Vergleiche
  - k. Zeichen: $N \cdot p^{k-1}$ Fälle, $N \cdot p^k$ gleich, $N \cdot p^{k-1} \cdot (1-p)$ ungleich
    $N \cdot p^k + N \cdot p^{k-1} \cdot (1-p) = N \cdot p^{k-1}$ Vergleiche

Suchen in Texten

- Aufwand für direktes Suchen in Texten
- Mittlerer Aufwand

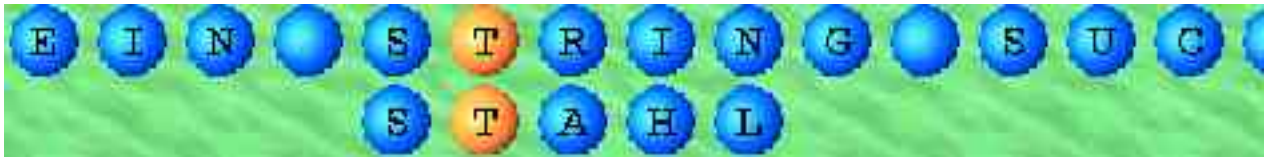
$$E[Vergleiche] = N + N \cdot p + N \cdot p^2 + \ldots + N \cdot p^{M-1} = N \cdot \frac{1 - p^{M-1}}{1 - p} \approx \frac{N}{1 - p}$$

$$\frac{N}{1 - p} \approx N + N \cdot p$$

  - Für p ≈ 1/30 ist Aufwand um ca. 4% größer als N!
  - Realistischeres Modell:
    p = P[Auftreten 1. Zeichen im Suchtext]
    z.B. $p_e \approx 0{,}15$ $p_x \approx 0{,}005$

Suchen in Texten

- Knuth-Morris-Pratt
- Ergebnis aus einer Theorie von S.A.Cook (1970)
- Suchaufwand maximel *O(N·M)*
- Idee: Nutze Information aus früheren Vergleichen
  - Lösung: Wenn an Stelle s (=3) Unterschied, schiebe Suchtext um maximales Stück



  - Hängt ab von Suchtext
  - 'vorbereitende' Arbeiten nötig

Suchen in Texten

- Beispiel

```
das MAXTAX MAXMAX ist  (Text, 6, 7, 7)
    MAXMAX             (Suchtext, 2, 3)
       MAXMAX          (Suchtext, 0)
```

- `das MTMTAX MAXMAX ist  (Text,6,7, 7)`

```
    MTMAAX             (Suchtext,2,3)
      MTMAAX           (Suchtext, 1)
```

- `das MMMMAX MAXMAX ist  (Text,6,7, 7)`

```
    MMMAAX             (Suchtext,2,3)
     MMMAAX            (Suchtext, 2)
```

Suchen in Texten

```
int sucheKMP(String suchtext){//Knuth-Morris-Pratt
  KMP kmp = new KMP(suchtext);
  int textIndex = 1, suchIndex = 1;
  while(textIndex<=text.length()&&
                  suchIndex<=suchtext.length()){
    if(suchIndex==0 || text.charAt(textIndex-1)==
                  suchtext.charAt(suchIndex-1)) {
      textIndex++; suchIndex++;
    } else {
      suchIndex = kmp.schiebe[suchIndex-1];
  } }
  if (suchIndex > suchtext.length())
       return textIndex - suchtext.length();
  else return -1;
} // Ende:: sucheKMP(suchtext)
```

Suchen in Texten

```
class KMP {
  int [] schiebe;
  String text;
  int len;
  public KMP(String text) {
    this.text = text;
    len = text.length();
    schiebe = new int[text.length()];
    wert();
  }
  void wert() { // nach Sedgewick
    int i, j;
    i=1; j=0; schiebe[0]=0;
    while(i<text.length()) {
```

Suchen in Texten

```
class KMP {
...
void wert() { //erstelle Tabelle
  int i, j;
  i=1; j=0; schiebe[0]=0;
  while( i < text.length() ) {
    if( j==0 ||
        text.charAt(i-1)==text.charAt(j-1) ){
      i++; j++;
      schiebe[i-1]=j;
    } else {
      j = schiebe[j-1];
} } }
```

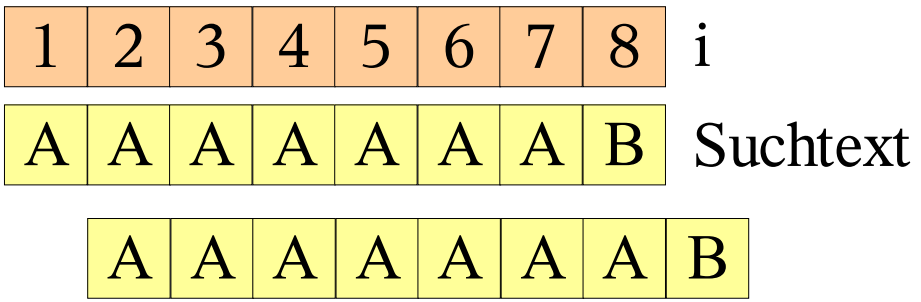
1	2	3	4	5	6	7	8	i
A	B	C	X	A	B	C	Z	Suchtext

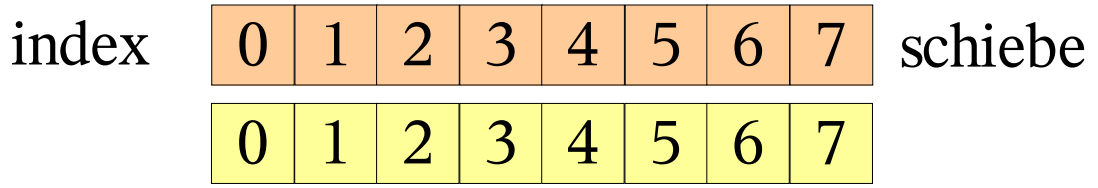
A	B	C	X	A	B	C	Z

index	0	1	2	3	4	5	6	7	schiebe
	0	1	1	1	1	2	3	4	

Suchen in Texten

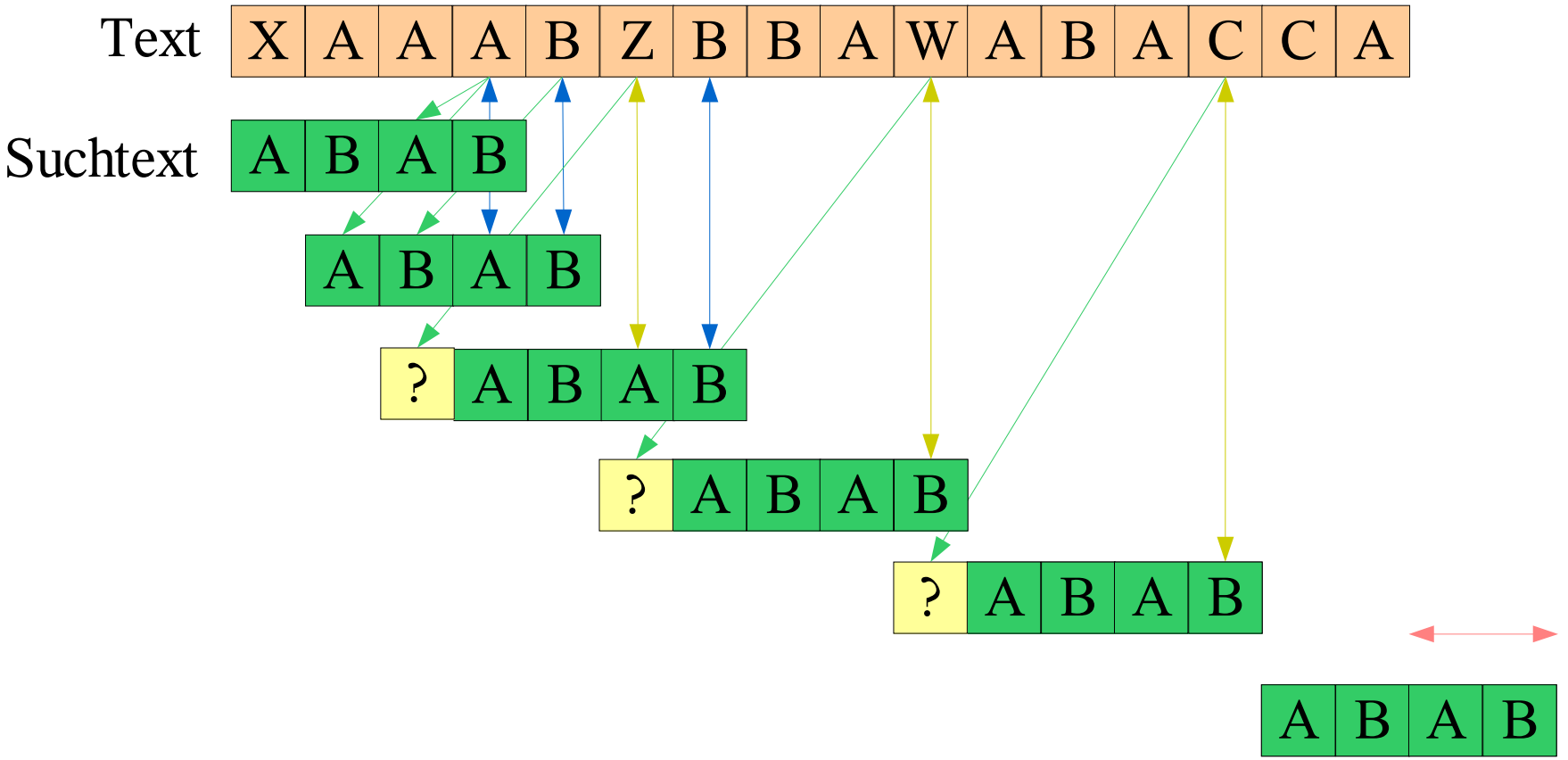
```
class KMP {
...
void wert() { //erstelle Tabelle
  int i, j;
  i=1; j=0; schiebe[0]=0;
  while( i < text.length() ) {
    if( j==0 ||
        text.charAt(i-1)==text.charAt(j-1) ){
      i++; j++;
      schiebe[i-1]=j;
    } else {
      j = schiebe[j-1];
} } }
```





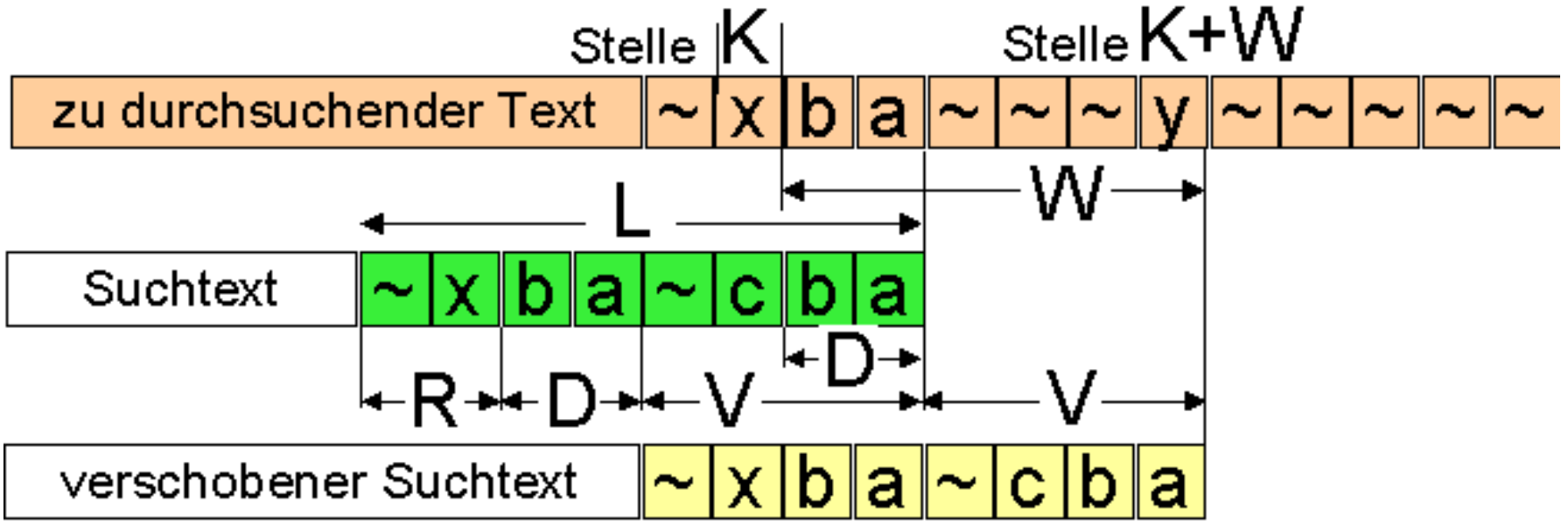
Suchen in Texten

- Suchen nach Boyer-Moore



Suchen in Texten

- Suchtext um V = L – R – D verschieben.

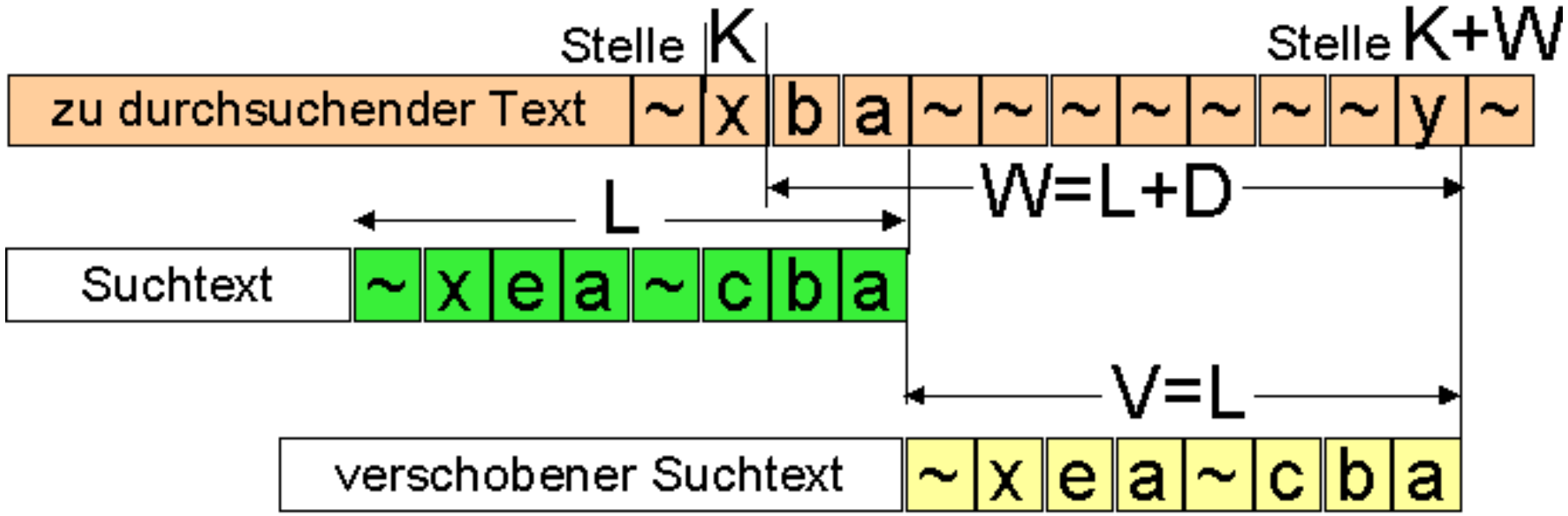


L = Länge des Suchtexts
D = Anzahl übereinstimmender Zeichen
R = Letztes Auftreten von 'x' im Suchtext
V = Verschiebung des Suchtexts
W = Verschiebung des Vergleichszeigers

V = L-R-D
W = V + D = L-R

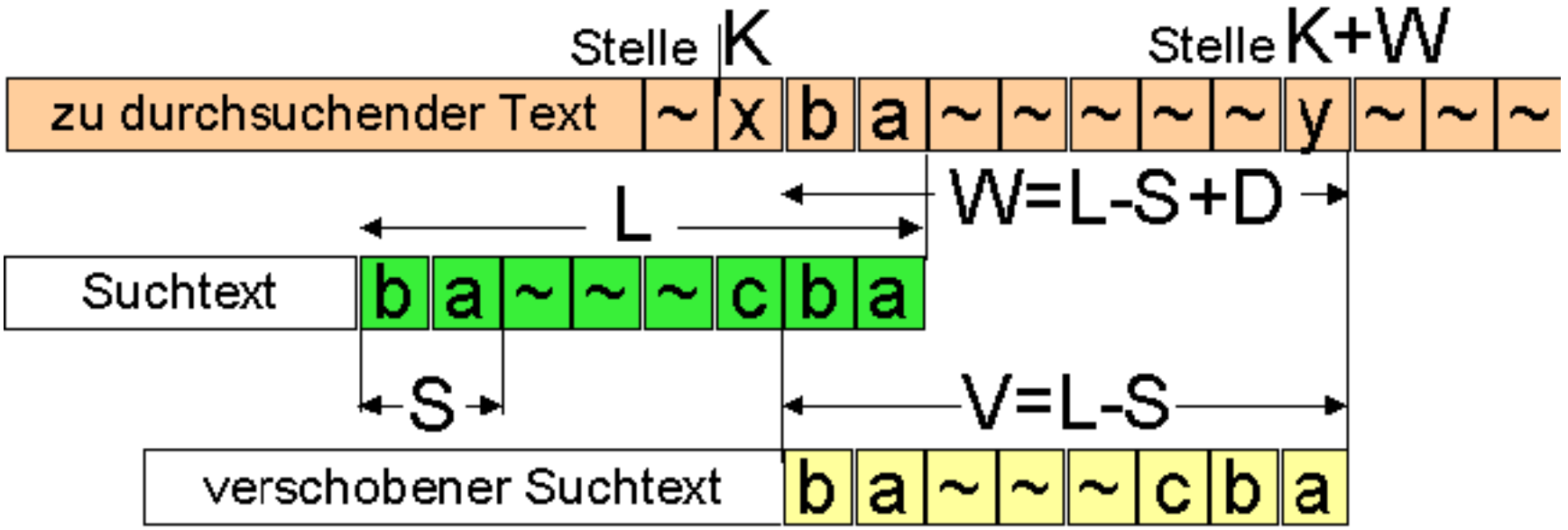
Suchen in Texten

- In Suchtext kommt nicht xba vor
- Am Anfang kommt weder ba noch a vor
- Verschiebe um L



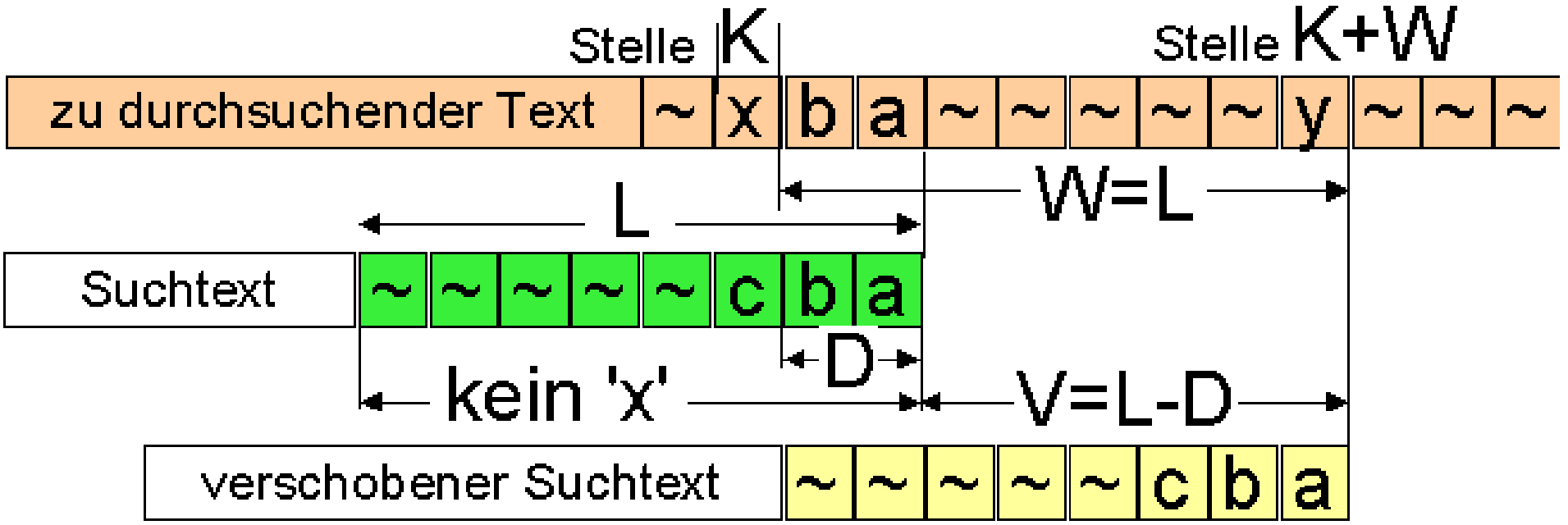
Suchen in Texten

- Kommt am Anfang ba oder a vor
- Verschiebe um L - |ba|



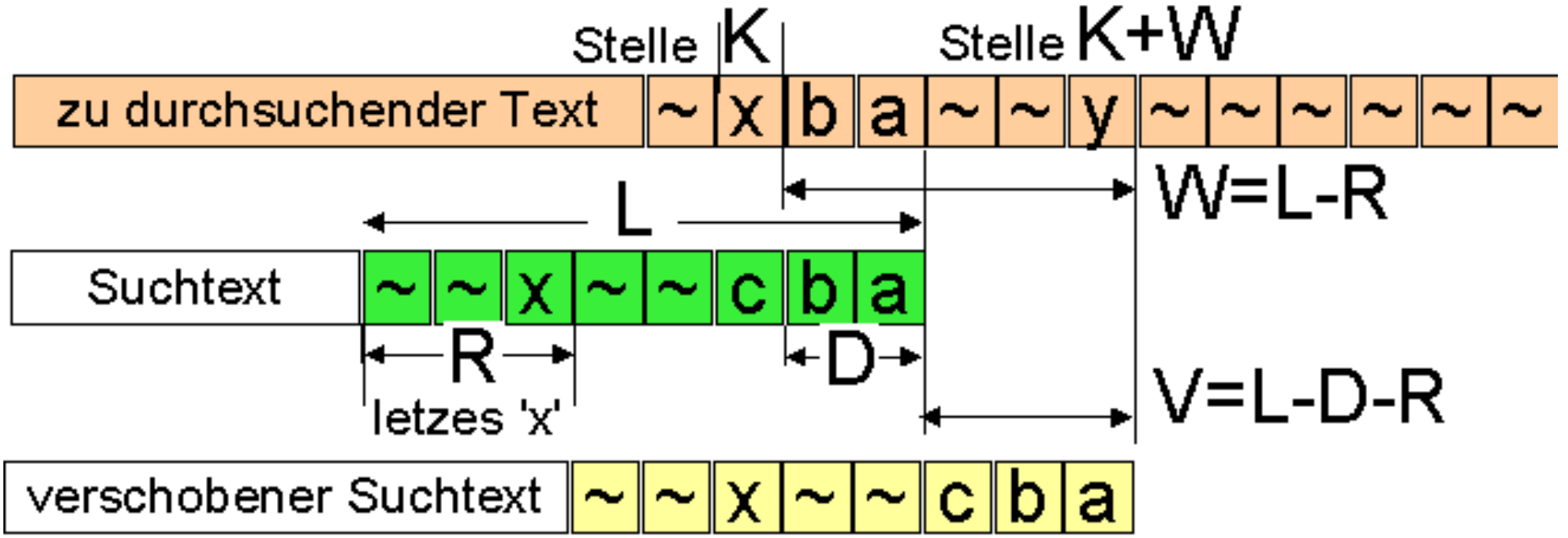
Suchen in Texten

- Vereinfachung: nur von 'x' abhängig
- 'x' kommt nicht im Text vor
- Verschiebe um L-D



Suchen in Texten

- Vereinfachung: nur von 'x' abhängig
- 'x' kommt im Text vor
- Verschiebe um L-D-R



Suchen in Texten

- Vereinfachung
- Kenntnis der geprüften Zeichen
- Erstellt Tabelle zur Verschiebung aus Suchtext

Suchen in Texten

```
// suche letztes Auftreten von c+text[len-s,len-1] in text
int tabelleBoyerMoore(int s, char c) {
  if (text.charAt(len-1-s1) == c) return 0;
  for (int i = len-2-s; i >=0; i--) { // i -> c
    if(text.charAt(i)==c && vergleich(i+1,len-s,s))
      return len-1-s-i;
  }
  for(int k=s-1; k>0;k--) // k = s-1, s-2, .. 0
    if(vergleich(0,len-k-1,k+1)) return len-k-1;
  return len; // verschiebe um den ganzen Text.
}
  schiebe = new int[text.length()][oben-unten+1];
  for(int i=0;i<text.length();i++)
    for(char c=unten; c<=oben; c++)
      schiebe[i][c-unten] = tabelleBoyerMoore(i,c);
```

Suchen in Texten

```
int sucheBM(String suchtext) {
  BM bm = new BM(suchtext,' ','z');
  int gefunden = -1;
  int bis = text.length()-suchtext.length();
  for(int von = 0; von <= bis; von++) {
    gefunden = von; int i;
    for(i = suchtext.length()-1; i>=0; i--) {
      int s = bm.schiebe[suchtext.length()-1-i]
                        [text.charAt(i+von)-' '];
      if(s>0) { // Zeichen ungleich, also schiebe
        von += s-1;     // da ja auch noch von++
        gefunden = -1; break; // nicht gefunden
    } }
    if(i<0) return gefunden;
  } return gefunden;}
```