ADS: Algorithmen und Datenstrukturen 2

Teil VI

Peter F. Stadler & Konstantin Klemm

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University of Leipzig**

12. Mai 2010

Lempel-Ziv Algorithmen

LZ77 (Sliding Window)

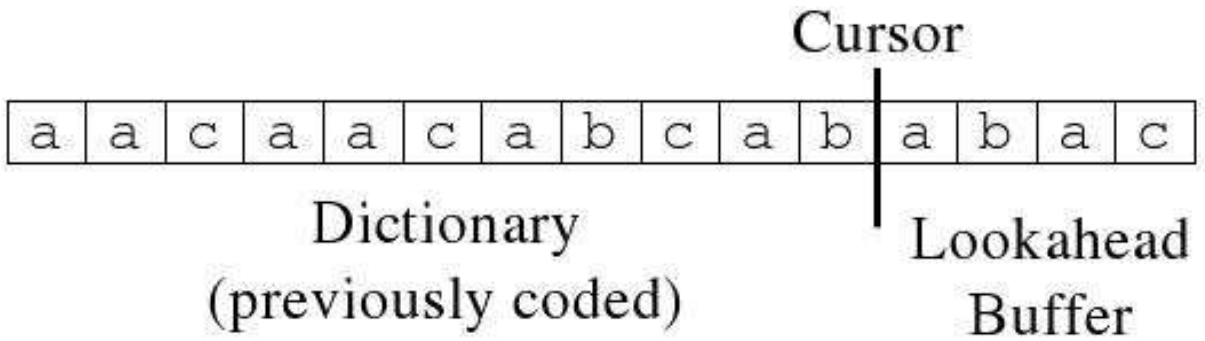
- Varianten: LZSS (Lempel-Ziv-Storer-Szymanski)
- Applications: `gzip`, Squeeze, LHA, PKZIP, ZOO

LZ78 (Dictionary Based)

- Variants: LZW (Lempel-Ziv-Welch), LZC (Lempel-Ziv-Compress)
- Applications: `compress`, GIF, CCITT (modems), ARC, PAK

Normalerweise wurde LZ77 als besser und langsamer als LZ78 betrachtet, aber auf leistungsfähigeren Rechnern ist LZ77 auch schnell.

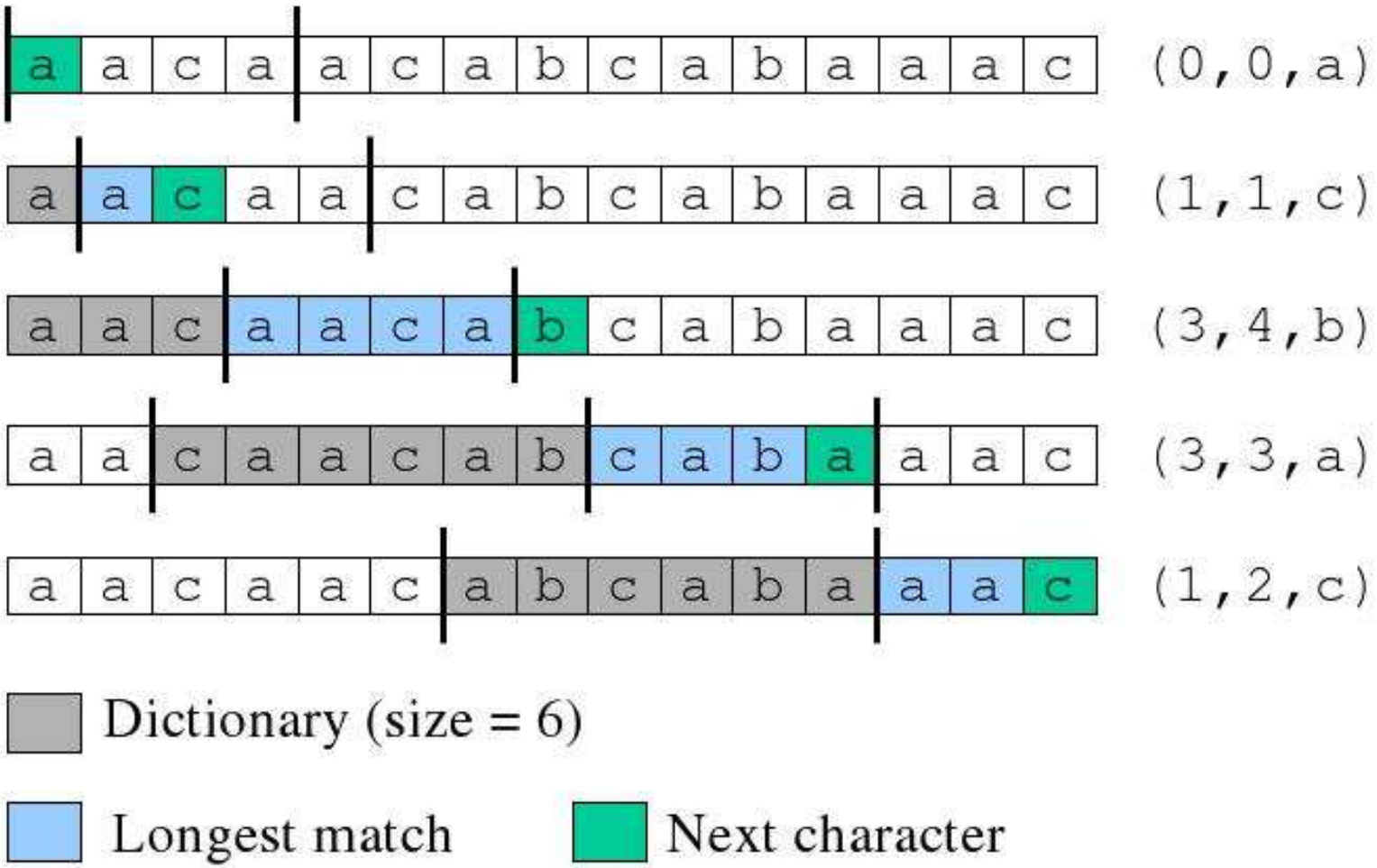
LZ77: Sliding Window Lempel-Ziv



Dictionary- und Buffer-Windows haben feste Länge und verschieben sich zusammen mit dem Cursor.
An jeder Cursor-Position passiert folgendes:

- Ausgabe des Tripels `(p,l,c)`
  - `p` = relative Position des longest match im Dictionary
  - `l` = Länge des longest match
  - `c` = nächstes Zeichen rechts vom longest match
- Verschiebe das Window um `l+1`

LZ77: Example



LZ77 Decoding

Der Decodierer arbeitet mit dem selben Dictionary-Window wie der Codierer

- Im Falle des Tripels `(p,l,c)` geht er `p` Schritte zurück, liest die nächsten `l` Zeichen und kopiert diese nach hinten. Dann wird noch `c` angefügt.

Was ist im Falle `l>p`? (d.h. nur ein Teil der zu copierenden Nachricht ist im Dictionary)

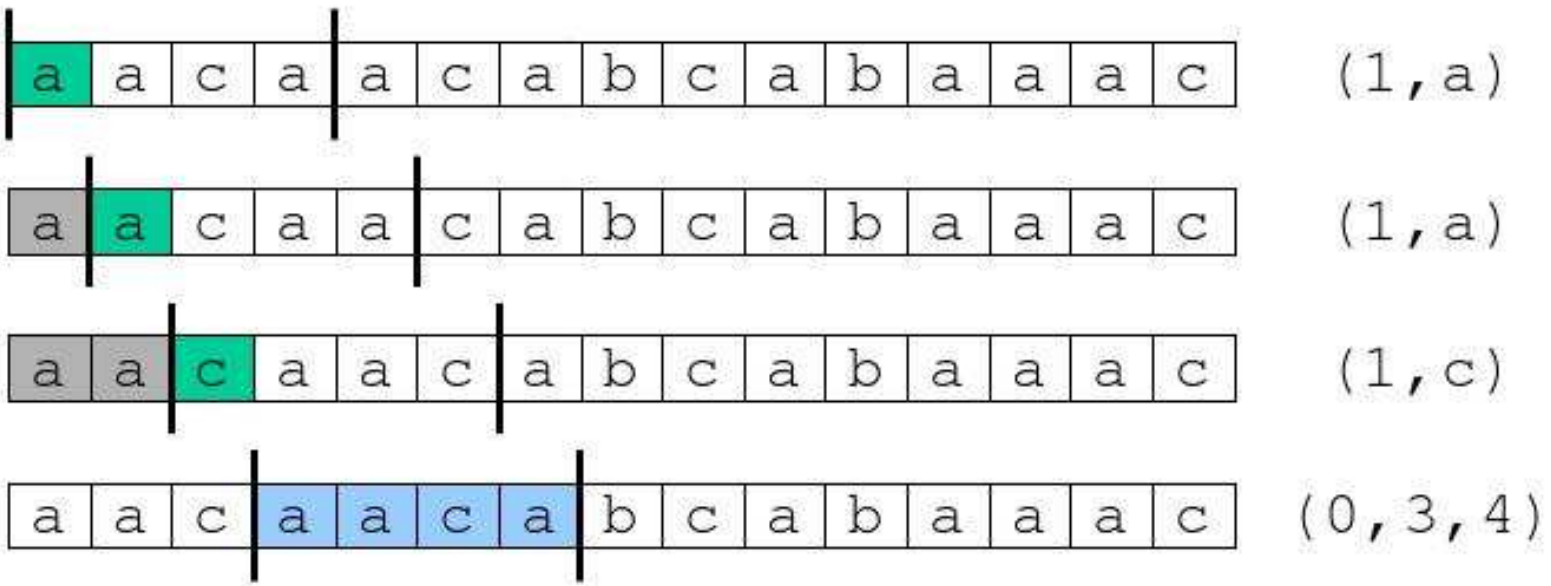
- Beispiel `dict = abcd, codeword = (2,9,e)`
- Lösung: Kopiere einfach zeichenweise:

```
for (i = 0; i < length; i++)
out[cursor+i] = out[cursor-offset+i]
```

- `Out = abcdcdcdcdcdce`

LZ77 Optimierungen bei `gzip` I

LZSS: Der Output hat eins der zwei Formate
`(0, position, length)` oder `(1,char)`
Benutze das zweite Format, falls `length < 3`.



Optimierungen bei `gzip` II

- Nachträgliche Huffman-Codierung der Ausgabe
- Clevere Strategie bei der Codierung: Möglicherweise erlaubt ein kürzerer Match in diesem Schritt einen viel längeren Match im nächsten Schritt
- Benutze eine Hash-Tabelle für das Wörterbuch.
  - Hash-Funktion für Strings der Länge drei.
  - Suche für längere Strings im entsprechenden Überlaufbereich die längste Übereinstimmung.

Theorie zu LZ77

- LZ77 ist asymptotisch optimal [Wyner-Ziv,94]
- Komprimiert hinreichend lange Strings entsprechend seiner Entropie, falls die Fenstergröße gegen unendlich geht.

$$H_n = \sum_{X \in A^n} p(X) \log \frac{1}{p(X)}$$

$$H = \lim_{n \to \infty} H_n$$

  Achtung, hier ist wirklich eine sehr große Fenstergröße nötig.
- In der Praxis wird meist ein Puffer von 216 Zeichen verwendet.

Die Burrows-Wheeler-Transformation

- Michael Burrows und David Wheeler, Mai 1994
- Verwendet in `bzip`
- Die Kompression ist sehr gut geeignet für Text, da Kontext berücksichtigt wird.
- Die eigentliche Burrows-Wheeler-Transformation ist eine vorgelagerte Umordnung von Textblöcken, die eine spätere Kompression mit einem Move-to-Front-Kodierer und einer weiteren Huffman-Codierung vorbereitet.

BWT: Erster Schritt

- Ein Eingabeblock der Länge N wird als quadratische Matrix dargestellt, die alle Rotationen des Eingabeblocks enthält.
- Die Zeilen der Matrix werden alphabetisch sortiert.
- Die letzte Spalte und die Zeilennummer des Originalblocks werden ausgegeben.
- In dieser Ausgabe sorgt ein ähnlicher Kontext von Buchstaben links davor für lange Runs gleicher Buchstaben.
- Daraus lässt sich der Originalblock wieder rekonstruieren (wird hier nicht bewiesen, sondern nur am Beispiel gezeigt).

Die Burrows-Wheeler-Transformation

Vorwärtstransformation von `HelloCello`

	0	1	2	3	4	5	6	7	8	9
0	H	e	l	l	o	C	e	l	l	o
1	e	l	l	o	C	e	l	l	o	H
2	l	l	o	C	e	l	l	o	H	e
3	l	o	C	e	l	l	o	H	e	l
4	o	C	e	l	l	o	H	e	l	l
5	C	e	l	l	o	H	e	l	l	o
6	e	l	l	o	H	e	l	l	o	C
7	l	l	o	H	e	l	l	o	C	e
8	l	o	H	e	l	l	o	C	e	l
9	o	H	e	l	l	o	C	e	l	l

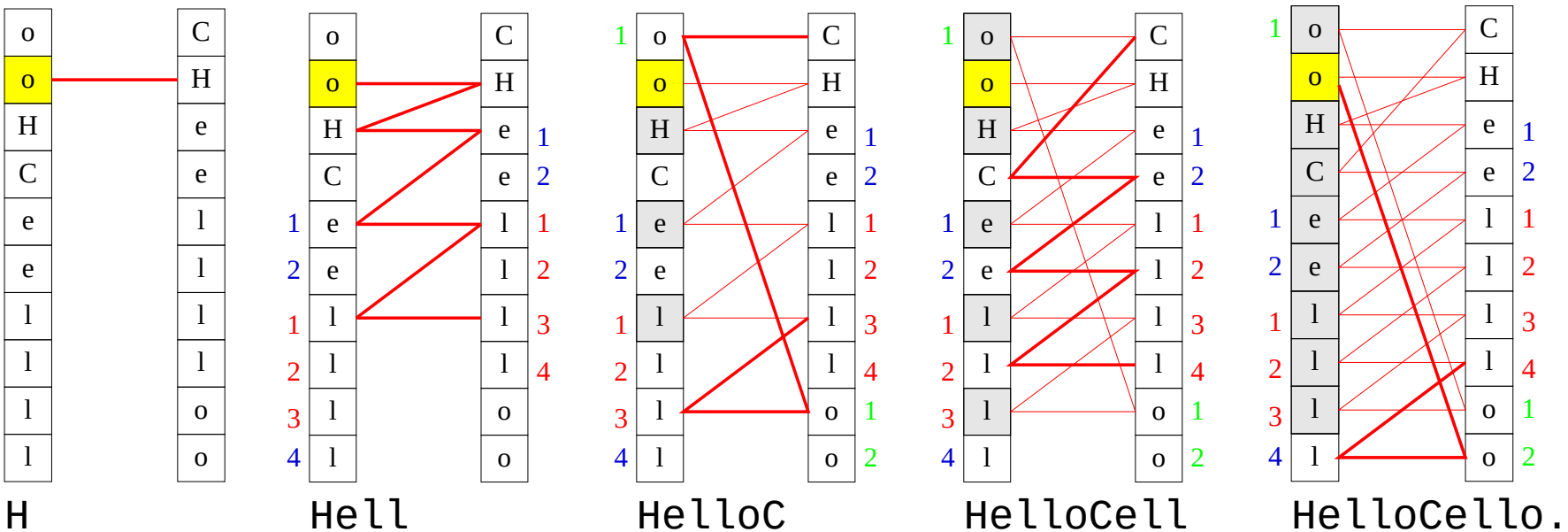
HelloCello

	0	1	2	3	4	5	6	7	8	9	
0	C	e	l	l	o	H	e	l	l	o	
1	H	e	l	l	o	C	e	l	l	o	*I*
2	e	l	l	o	C	e	l	l	o	H	
3	e	l	l	o	H	e	l	l	o	C	
4	l	l	o	C	e	l	l	o	H	e	
5	l	l	o	H	e	l	l	o	C	e	
6	l	o	C	e	l	l	o	H	e	l	
7	l	o	H	e	l	l	o	C	e	l	
8	o	C	e	l	l	o	H	e	l	l	
9	o	H	e	l	l	o	C	e	l	l	
										L	

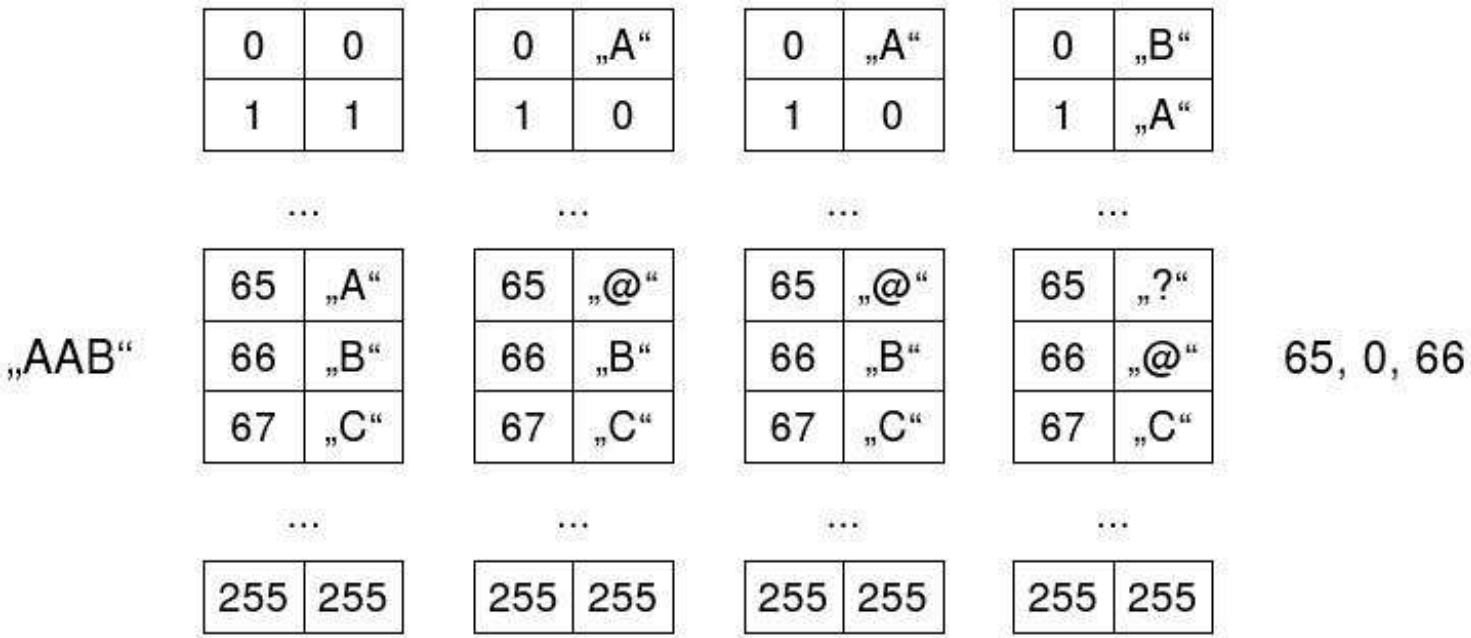
ooHCeellll

Die Burrows-Wheeler-Transformation

Rücktransformation:



MTF: Move-To-Front-Coding



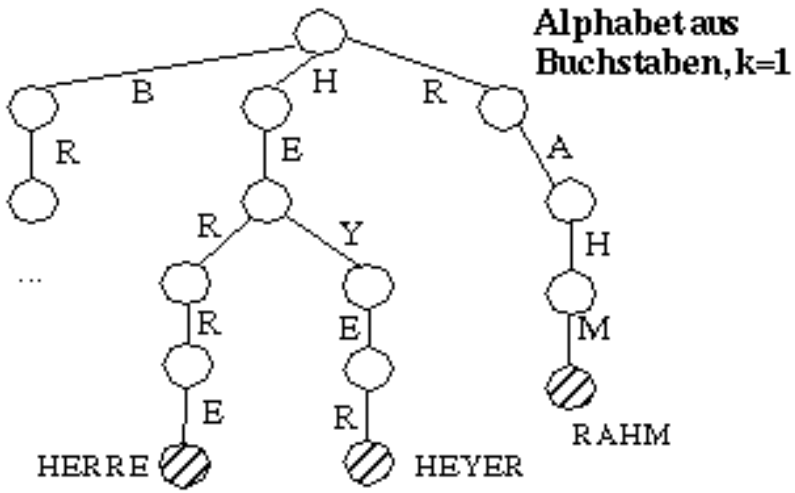
Eingabe:	A	A	B	B	C	C	C	A	A	B	C	C	C
Ausgabe:	65	0	66	0	67	0	0	2	0	2	2	0	0

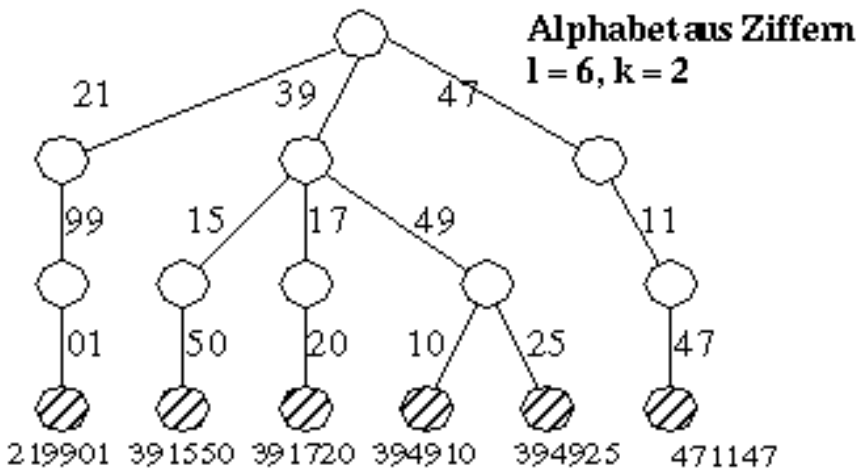
Wozu das Ganze?

- BWT erzeugt durch Sortierung lange Läufe gleicher Zeichen
- MTF erzeugt kleine (Index) Zahlen fuer häufig vorkommende, genauer: Lange Folgen von '0'.
- Lauflängen-Kodierer (evtl. speziell fuer '0') reduziert Redundanz
- Huffman-Kodierer (möglichst kurze Bitfolge)

Suchen und Finden mit Index-Strukturen

Nochmals Digitale Suchbäume:





m-ärer Trie I

- Spezielle Implementierung des Digitalbaumes:
  Trie leitet sich von Information Retrieval ab (E.Fredkin, 1960)
- Spezielle m-Wege-Bäume, wobei Kardinalität des Alphabets und Länge k der Schlüsselteile den Grad m festlegen
  - bei Ziffern: $m = 10$
  - bei Alpha-Zeichen: $m = 26$
  - bei alphanumerischen Zeichen: $m = 36$
  - bei Schlüsselteilen der Länge k potenziert sich Grad entsprechend, d. h. als Grad ergibt sich m^k

Trie-Darstellung

- Jeder Knoten eines Tries vom Grad m ist im Prinzip ein eindimensionaler Vektor mit m Zeigern
- Jedes Element im Vektor ist einem Zeichen (bzw. Zeichenkombination) zugeordnet. Auf diese Weise wird ein Schlüsselteil (Kante) implizit durch die Vektorposition ausgedrückt.
- Beispiel: Knoten eines 10-ären Trie mit Ziffern als Schlüsselteilen $|P_0|P_1|P_2|P_3|P_4|P_5|P_6|P_7|P_8|P_9|$ $m = 10$, $k = 1$
- implizite Zuordnung von Ziffer/Zeichen zu Zeiger. P_i gehört also zur Ziffer i. Tritt Ziffer i in der betreffenden Position auf, so verweist P_i auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist P_i mit `NULL` belegt
- Wenn der Knoten auf der j-ten Stufe eines 10-ären Trie liegt, dann zeigt P_i auf einen Unterbaum, der nur Schlüssel enthält, die in der j-ten Position die Ziffer i besitzen

Operationen auf Tries

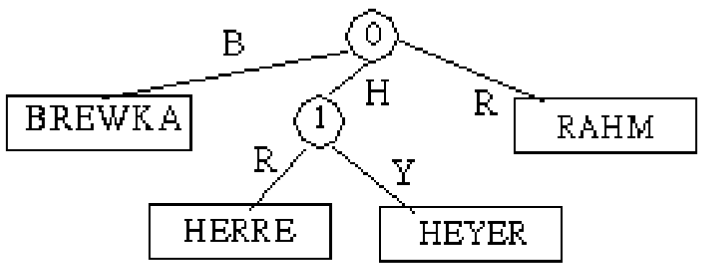
- **Direkte Suche**
  In der Wurzel wird nach dem 1. Zeichen des Suchschlüssels verglichen.
  Bei Gleichheit wird der zugehörige Zeiger verfolgt.
  Im gefundenen Knoten wird nach dem 2. Zeichen verglichen, usw.
- effiziente Bestimmung der Abwesenheit eines Schlüssels
- **Einfügen**
  Wenn Suchpfad schon vorhanden, wird NULL-Zeiger in einen Verweis auf den Datensatz umgewandelt, sonst Einfügen von neuen Knoten
- **Löschen**
  Nach Aufsuchen des richtigen Knotens wird ein Datensatz-Zeiger auf NULL gesetzt. Besitzt daraufhin der Knoten nur NULL-Zeiger, wird er aus dem Baum entfernt (rekursive Überprüfung der Vorgängerknoten)

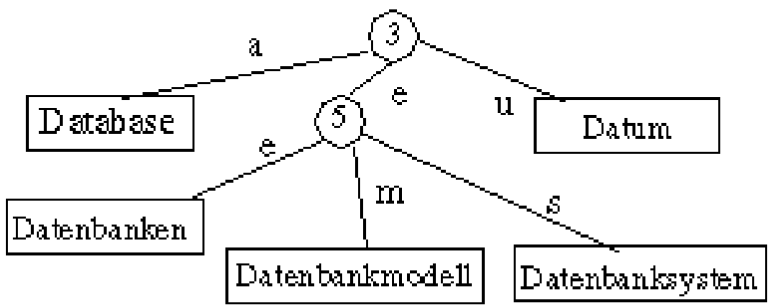
PATRICIA-Tree

Practical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric

Merkmale

- Binärdarstellung für Schlüsselwerte → binärer Digitalbaum
- Speicherung der Schlüssel in den Blättern
- innere Knoten speichern, wie viele Zeichen (Bits) beim Test zur Wegeauswahl zu überspringen sind
- Vermeidung von Einwegverzweigungen, in dem bei nur noch einem verbleibenden Schlüssel direkt auf entsprechendes Blatt verwiesen wird





PATRICIA-Tree

- speichereffizient
- sehr gut geeignet für variabel lange Schlüssel und (sehr lange) Binärdarstellungen von Schlüsselwerten
- bei jedem Suchschlüssel muss die Testfolge von der Wurzel beginnend ganz ausgeführt werden, bevor über Erfolg oder Misserfolg der Suche entschieden werden kann

Suffix Bäume

- Wichtiges Werkzeug bei der Analyse langer Zeichenketten wie sie in der Bioinformatik auftreten:
  - Menschliches Genom: 3×10^9 Zeichen
  - Frage: enthält die menschliche DNA Sequenz die Zeichenkette GATACCAGATACCAGATACCAGATACCA
- Sequenzdatenbanken erhalten Millionen solcher Anfragen. Wir hätten die Antwort daher gerne mit einem Aufwand proportional zur Anfrage aber nicht der Länge des Genoms in dem gesucht wird.

Suffix Bäume

Suffix Baum = PATRICIA Trie in dem alle **Suffixe** der Genom-Sequenz eingetragen werden

... und das sehen wir uns nächste Woche genauer an.