UA RUHR

Algorithmen auf Sequenzen

Volltext-Indexdatenstrukturen: BWT, FM-Index

Sven Rahmann

Genominformatik
Universitätsklinikum Essen
Universität Duisburg-Essen
Universitätsallianz Ruhr

Die Burrows-Wheeler-Transformation (BWT)

UA RUHR

- BWT wurde von Michael Burrows und David Wheeler 1994 zur verlustfreien Kompression von Daten entwickelt, basierend auf Vorarbeiten von 1983.
- Lange Zeit wurde die Bedeutung der BWT nicht vollständig erkannt.
- Heute: Basis für Kompression, Datenstrukturen, Algorithmen

Die Burrows-Wheeler-Transformation (BWT)

UA RUHR

- BWT wurde von Michael Burrows und David Wheeler 1994 zur verlustfreien Kompression von Daten entwickelt, basierend auf Vorarbeiten von 1983.
- Lange Zeit wurde die Bedeutung der BWT nicht vollständig erkannt.
- Heute: Basis für Kompression, Datenstrukturen, Algorithmen
- Die BWT zu einem Text T oder $T\$$ ist stets eine Permutation der Zeichen des ursprünglichen Texts.
- informell: Folge der Zeichen **vor** lexikographisch sortierten Suffixen
- Verschiedene Varianten (z.B. zyklisch oder mit Wächter)
  (Zyklisch: Nach Textende, betrachte Zeichen am Anfang des Texts)

Die zyklische Burrows-Wheeler Transformation

UA RUHR

Definition (BWT)

Gegeben T mit $|T| = n$,
bilde die $n \times n$ Matrix M, deren Zeilen zyklische Verschiebungen des Texts sind.
Sortiere die Zeilen lexikographisch. Die BWT entspricht der letzten Spalte.

Die zyklische Burrows-Wheeler Transformation

UA RUHR

Definition (BWT)

Gegeben T mit $|T| = n$,
bilde die $n \times n$ Matrix M, deren Zeilen zyklische Verschiebungen des Texts sind.
Sortiere die Zeilen lexikographisch. Die BWT entspricht der letzten Spalte.

F						L
$	b	a	n	a	n	a
a	$	b	a	n	a	n
a	n	a	$	b	a	n
a	n	a	n	a	$	b
b	a	n	a	n	a	$
n	a	$	b	a	n	a
n	a	n	a	$	b	a

(in der Regel angewendet auf Texte ohne Wächter)

Die BWT mit Wächter oder Stringende

UA RUHR

Definition (BWT)

Sei Text $T\$$ mit $n := |T\$|$ gegeben und pos das Suffixarray.
Dann ist die BWT die Abbildung $r \mapsto bwt[r] := T[pos[r] - 1]$ mit $T[-1] := \$$.

Die BWT mit Wächter oder Stringende

UA RUHR

Definition (BWT)

Sei Text $T\$$ mit $n := |T\$|$ gegeben und pos das Suffixarray.
Dann ist die BWT die Abbildung $r \mapsto bwt[r] := T[pos[r] - 1]$ mit $T[-1] := \$$.

r	$pos[r]$	$bwt[r]$	$T[pos[r]:]$
0	6	a	$
1	5	n	a$
2	3	n	ana$
3	1	b	anana$
4	0	$	banana$
5	4	a	na$
6	2	a	nana$

Eigenschaften der BWT

- Da das Suffixarray in Linearzeit konstruiert werden kann, beträgt die Laufzeit für die Konstruktion des BWT auch $\mathcal{O}(n)$.
- Bei natürlichsprachlichen Texten enthält die BWT häufig lange Läufe desselben Zeichens, was sich gut komprimieren lässt.
- Das k-te Auftreten eines Zeichens in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffixarray.
- Die BWT lässt sich ohne zusätzliche Informationen wieder in ihren ursprünglichen Text rücktransformieren.

Eigenschaften der BWT

bwt	*Sortierter Text*
d	ie BWT lässt sich ohne
d	ie Indizes der sortierten
d	ie LMS-Substrings bilden
d	ie hinterste freie Stelle
d	ie selbe Anordnung
d	ie selbe Länge
d	ie sortierte Reihenfolge der
d	iese ursprüngliche Definition
d	ieser Ansatz hat eine

Eigenschaften der BWT

- Bei natürlichsprachlichen Texten kann ausgenutzt werden, dass bestimmte Buchstabenkombinationen, wie 'die', 'sch', 'tz' oder 'ie' häufig vorkommen.
- Dementsprechend beinhaltet eine BWT oft lange Blöcke von gleichen Zeichen, z.B. . . . ddddddddddddddd. . .
- Hierauf lässt sich *run-length-encoding* (RLE) sinnvoll anwenden, um solche Buchstabenblöcke mit wenig Bits darzustellen.

bzip2

Das Tool bzip2 von Julian Seward basiert auf der BWT. Als Eingabe bekommt bzip2 einen Text und eine Blockgröße und arbeitet wie folgt:
Für jeden Block separat:

1. Berechne die BWT.
2. Wende auf jede BWT die Move-to-front Transformation an. Die so entstehende Sequenz beinhalt viele Nullen und kleine Werte.
3. Spezielles Run-length encoding (RLE) der Sequenz
4. Huffman-Codierung der Sequenz: Häufige Zeichen werden durch kurze Bitfolgen kodiert, seltene Zeichen durch lange Bitfolgen.

Alle Transformationsschritte sind invertierbar.

Invertierbarkeit

UA RUHR

Lemma

Das k-te Auftreten des Zeichens $c \in \Sigma$ in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffixarray.

Invertierbarkeit

UA RUHR

Lemma

Das k-te Auftreten des Zeichens $c \in \Sigma$ in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffixarray.

r	$pos[r]$	$bwt[r]$	$T[pos[r]:]$
0	6	a	$
1	5	n	a$
2	3	n	ana$
3	1	b	anana$
4	0	$	banana$
5	4	a	na$
6	2	a	nana$

Invertierbarkeit

Lemma

Das k-te Auftreten des Zeichens $c \in \Sigma$ in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffixarray.

r	pos[r]	bwt[r]	$T[pos[r]:]$
0	6	a	$
1	5	n	a$
2	3	n	ana$
3	1	b	anana$
4	0	$	banana$
5	4	a	na$
6	2	a	nana$

Betrachte zwei Ränge $r_1 < r_2$ mit $bwt[r_1] = bwt[r_2]$. Da die Suffixe $pos[r_1]$ und $pos[r_2]$ lexikographisch sortiert sind, die jeweils auf die BWT-Zeichen folgen, entspricht die Reihenfolge der BWT-Zeichen der im Suffixarray.

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

1) Der Wächter steht bei $r = 4$. Sein Nachfolger ist b. Der Originaltext beginnt mit diesem Zeichen.

$$T = \texttt{b}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

2) Das b stand bei $r = 3$. Der Nachfolger von b ist ein a. Das Zeichen wird zum Text hinzugefügt.

$$T = \texttt{ba}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

3 - n) Es wird insgesamt n mal die Position aktualisiert und der Nachfolger zum Text hinzugefügt.

$$T = \texttt{ban}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

3 - n) Es wird insgesamt n mal die Position aktualisiert und der Nachfolger zum Text hinzugefügt.

$$T = \texttt{bana}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

3 - n) Es wird insgesamt n mal die Position aktualisiert und der Nachfolger zum Text hinzugefügt.

$$T = \texttt{banan}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

3 - n) Es wird insgesamt n mal die Position aktualisiert und der Nachfolger zum Text hinzugefügt.

$$T = \texttt{banana}$$

Invertierbarkeit der BWT

Stabile Sortierung (mit Speichern der r-Werte) der BWT liefert Nachfolger zu jedem Zeichen und erlaubt Rekonstruktion des Texts.

r	0	1	2	3	4	5	6
$bwt[r]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
$orig\_r[r]$	4	0	5	6	3	1	2

3 - n) Es wird insgesamt n mal die Position aktualisiert und der Nachfolger zum Text hinzugefügt.

$$T = \texttt{banana\$}$$

Sortierung in $\mathcal{O}(n)$ Zeit mit bucket sort, $|\Sigma| = \mathcal{O}(1)$

Exakte Mustersuche

- Bei der Mustersuche mit Hilfe der BWT sucht man die Intervallgrenzen $[L, R]$ für den Index r des Suffixarrays *pos*, so dass alle Vorkommen des Musters an $pos[r]$ mit $L \leq r \leq R$ beginnen.
- **Backward Search**: Muster wird zeichenweise rückwärts bearbeitet.
- Backward Search simuliert **nicht** die Suche im Suffixbaum.

Exakte Mustersuche

- Bei der Mustersuche mit Hilfe der BWT sucht man die Intervallgrenzen $[L, R]$ für den Index r des Suffixarrays *pos*, so dass alle Vorkommen des Musters an $pos[r]$ mit $L \leq r \leq R$ beginnen.
- **Backward Search**: Muster wird zeichenweise rückwärts bearbeitet.
- Backward Search simuliert **nicht** die Suche im Suffixbaum.
- Das Intervall $[L, R]$ ist wie folgt definiert:
  ($\leq_m$: lexikographisch-kleiner-gleich auf den ersten $m = |P|$ Zeichen)

$$L := \min\{r \mid T[pos[r] :] \leq_m P\}$$
$$R := \max\{r \mid P \leq_m T[pos[r] :]\}$$

- Es gibt $R - L + 1$ Treffer des Musters im Text.
- Im Fall $L > R$ müsste man P zwischen R und L einfügen.

Exakte Mustersuche

- Ist $P = \epsilon$, dann gilt $L = 0$ und $R = n - 1$ (Initialisierung)
- Zur Suche sind zwei Hilfstabellen nötig:
  - *less*[*c*]: Anzahl Zeichen im Text, die kleiner sind als *c*
  - *Occ*[c, i]: zählt die Vorkommen von Zeichen *c* im Präfix *bwt*[..*i*].
- Beide Hilfstabellen zusammen werden als **FM-Index** bezeichnet ("Ferragina-Manzini" oder "full text in minute space")

Exakte Mustersuche

- Ist $P = \epsilon$, dann gilt $L = 0$ und $R = n - 1$ (Initialisierung)
- Zur Suche sind zwei Hilfstabellen nötig:
  - *less*[*c*]: Anzahl Zeichen im Text, die kleiner sind als *c*
  - *Occ*[c, i]: zählt die Vorkommen von Zeichen *c* im Präfix *bwt*[..*i*].
- Beide Hilfstabellen zusammen werden als **FM-Index** bezeichnet ("Ferragina-Manzini" oder "full text in minute space")

Beispiel: $T =$ `banana$`, $bwt =$ `annb$aa`

less	$	a	b	n
	0	1	4	5

Occ	0	1	2	3	4	5	6
$	0	0	0	0	1	1	1
a	1	1	1	1	1	2	3
b	0	0	0	1	1	1	1
n	0	1	2	2	2	2	2

Backward Search

Lemma

Sei $P^+ := cP$; $[L, R]$ das Intervall zu P und $[L^+, R^+]$ das gesuchte Intervall zu P^+.

$$L^+ = less[c] + Occ[c, L-1]$$
$$R^+ = less[c] + Occ[c, R] - 1$$

Backward Search

Lemma

Sei $P^+ := cP$; $[L, R]$ das Intervall zu P und $[L^+, R^+]$ das gesuchte Intervall zu P^+.

$$L^+ = less[c] + Occ[c, L-1]$$
$$R^+ = less[c] + Occ[c, R] - 1$$

- Zu Beginn ist $P = \epsilon$, somit ist das Muster im gesamten Intervall vorhanden: $L = 0$ und $R = n - 1$.
- Da $P^+ = c$, ist das Subintervall $[L_c, R_c]$ gesuchtt, in dem die Suffixe mit c beginnen.
- Das Intervall ist durch $L_c = less[c]$ und $R_c = less[c] + Occ[c, n-1] - 1$ gegeben. Das Lemma stimmt für $|P^+| = 1$.

Backward Search

UA RUHR

Lemma

Sei $P^+ := cP$; $[L, R]$ das Intervall zu P und $[L^+, R^+]$ das gesuchte Intervall zu P^+.

$$L^+ = less[c] + Occ[c, L-1]$$
$$R^+ = less[c] + Occ[c, R] - 1$$

- Allgemein: $[L^+, R^+]$ ist Subintervall von $[L_c, R_c]$
- Innerhalb $[L, R]$ zählen, wie viele Vorgänger ein c sind.
- Da das k-te c in der BWT auch dem k-ten c im Suffixarray entspricht, muss an Stelle L gezählt werden, wie oft c im Präfix $bwt[:L]$ vorkam, also $Occ[c, L-1]$. Diese Zahl mus nach $less[c]$ übersprungen werden, also $L^+ = L_c + Occ[c, L-1]$.
- Neue rechte Grenze: Zählen, wie oft c bis zur Position R vorkommt: $Occ[c, R]$.

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnrssssssss
L=        0
R=        32
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less
```

$bwt[r]$= nsnm$ssssgenlneeearneleienessseae

$T[pos[r]]$= $aaeeeeeeeeeeegillmnnnnnnnrssssssss

L= 0 $less[\mathrm{s}] + Occ[\mathrm{s}, -1]$

R= 32 $less[\mathrm{s}] + Occ[\mathrm{s}, 32] - 1$

Backward Search

```
                0         1         2         3
                01234567890123456789012345678901 2
T=              einsameeselessennasseneselngern$
P=              less

bwt[r]=         nsnm$ssssgenlneeearneleienessseae
T[pos[r]]=      $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=              0                 25 + Occ[s, −1]
R=              32                25 + Occ[s, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienesssseae
T[pos[r]]= $aaeeeeeeeeeeegillmnnnnnnnrssssssss
L=        0               25 + 0
R=        32              25 + Occ[s, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnrssssssss
L=        25
R=        32
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        25          less[s] + Occ[s, 24]
R=        32          less[s] + Occ[s, 32] − 1
```

Backward Search

```
                0         1         2         3
                01234567890123456789012345678901 2
T=              einsameeselessennasseneselngern$
P=              less

bwt[r]=         nsnm$ssssgenlneeearneleienessseae
T[pos[r]]=      $aeeeeeeeeeeegillmnnnnnnnrssssssss
L=              25          25 + Occ[s, 24]
R=              32          25 + Occ[s, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienesssseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        25          25 + 5
R=        32          25 + Occ[s, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        30
R=        32
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnrssssssss
L=        30          less[e] + Occ[e, 29]
R=        32          less[e] + Occ[e, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        30          3 + Occ[e, 29]
R=        32          3 + Occ[e, 32] − 1
```

Backward Search

```
          0         1         2         3
          0123456789012345678901234567890123
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienesssseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        30          3 + 8
R=        32          3 + Occ[e, 32] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnrssssssss
L=        11
R=        12
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeeegillmnnnnnnnrssssssss
L=        11          less[l] + Occ[l, 10]
R=        12          less[l] + Occ[l, 12] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeeegillmnnnnnnnrssssssss
L=        11          15 + Occ[l, 10]
R=        12          15 + Occ[l, 12] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$sssgenlneeearneleienessseae
T[pos[r]]= $aeeeeeeeeeeegillmnnnnnnnrssssssss
L=        11          15 + 0
R=        12          15 + Occ[l, 12] − 1
```

Backward Search

```
          0         1         2         3
          01234567890123456789012345678901 2
T=        einsameeselessennasseneselngern$
P=        less

bwt[r]=   nsnm$ssssgenlneeearneleienessseae
T[pos[r]]= $aaeeeeeeeeeegillmnnnnnnnrssssssss
L=        15
R=        15
```

$pos = [32, 4, 17, 6, 0, 9, 25, 20, 14, 29, 7, 22, 11, 28, 1, 10, 26,$
$5, 31, 16, 21, 27, 15, 2, 30, 3, 8, 24, 19, 13, 23, 18, 12]$

Analyse der Backward Search

- Pro Zeichen des Musters und pro Grenze zwei Lookups, also $\mathcal{O}(1)$.
- Insgesamt erfolgt die Mustersuche in $\mathcal{O}(m)$ Zeit.
- zusätzlicher Speicherplatz:
  - *less*: $\mathcal{O}(|\Sigma| \log(n))$ Bits
  - *Occ*: $\mathcal{O}(|\Sigma| n \log(n))$ Bits

Analyse der Backward Search

- Pro Zeichen des Musters und pro Grenze zwei Lookups, also $\mathcal{O}(1)$.
- Insgesamt erfolgt die Mustersuche in $\mathcal{O}(m)$ Zeit.
- zusätzlicher Speicherplatz:
  - *less*: $\mathcal{O}(|\Sigma| \log(n))$ Bits
  - *Occ*: $\mathcal{O}(|\Sigma| n \log(n))$ Bits
- *Occ* ist nicht sehr komplex, ist der Speicher nötig?

Optimierung des Speicherplatzverbrauchs

BWT als Bitvektoren (z.B. einen für jeden Buchstaben) darstellen,
Rank-Datenstruktur für Bitsequenzen verwenden

Optimierung des Speicherplatzverbrauchs

BWT als Bitvektoren (z.B. einen für jeden Buchstaben) darstellen,
Rank-Datenstruktur für Bitsequenzen verwenden
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Bits	0	1	2	3	4	5	6
\$	0	0	0	0	1	0	0
a	1	0	0	0	0	1	1
b	0	0	0	1	0	0	0
n	0	1	1	0	0	0	0

Optimierung des Speicherplatzverbrauchs

BWT als Bitvektoren (z.B. einen für jeden Buchstaben) darstellen,
Rank-Datenstruktur für Bitsequenzen verwenden
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Bits	0	1	2	3	4	5	6
\$	0	0	0	0	1	0	0
a	1	0	0	0	0	1	1
b	0	0	0	1	0	0	0
n	0	1	1	0	0	0	0

Für jeden Buchstaben eigenes Bitfeld und Rank-Datenstruktur: $\mathcal{O}(n|\Sigma|) + o(n|\Sigma|)$ Bits.

Optimierung des Speicherplatzverbrauchs

BWT als Bitvektoren (z.B. einen für jeden Buchstaben) darstellen,
Rank-Datenstruktur für Bitsequenzen verwenden
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Bits	0	1	2	3	4	5	6
\$	0	0	0	0	1	0	0
a	1	0	0	0	0	1	1
b	0	0	0	1	0	0	0
n	0	1	1	0	0	0	0

Für jeden Buchstaben eigenes Bitfeld und Rank-Datenstruktur: $\mathcal{O}(n|\Sigma|) + o(n|\Sigma|)$ Bits.
Text/bwt selbst benötigt eigentlich nur $\mathcal{O}(n \log |\Sigma|)$ Bits!

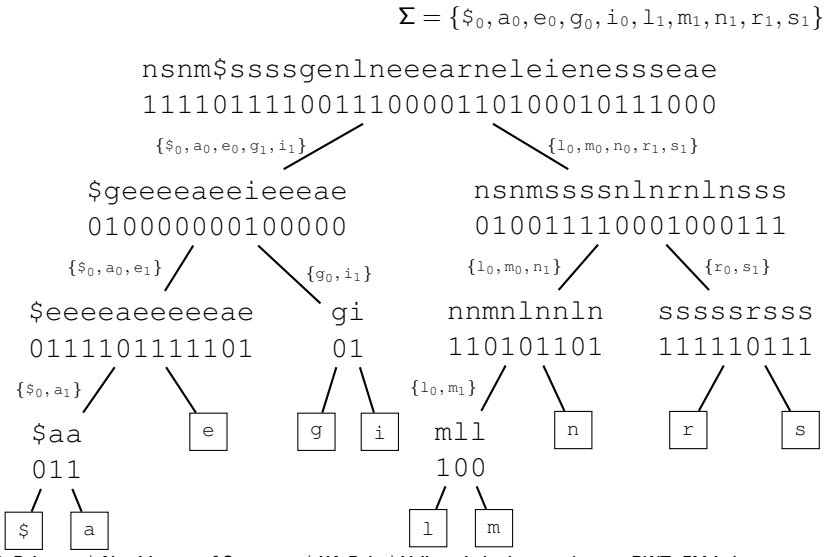
Waveletbaum

UA RUHR

Definition (Waveletbaum)

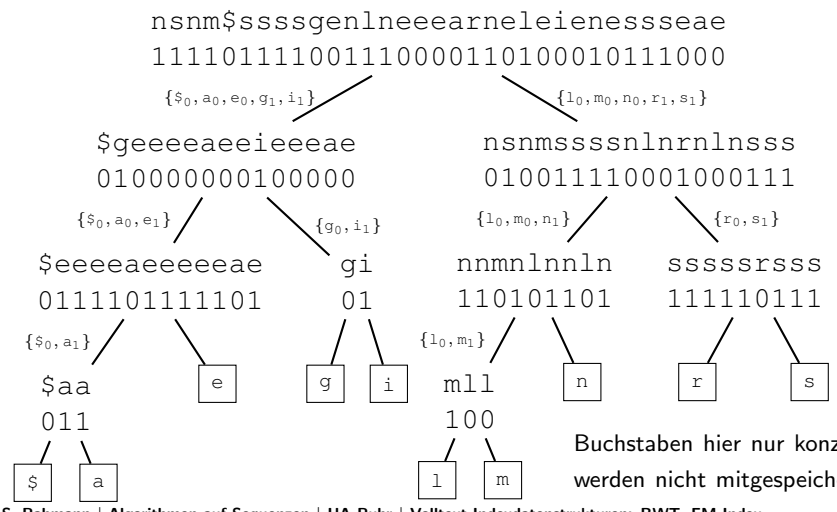
Sei T ein Text mit $|T| = n$ und Alphabet $\Sigma = \{0, \ldots, s-1\}$ mit $|\Sigma| = s$.
Der Waveletbaum zu T ist ein balancierter Binärbaum mit $|\Sigma|$ Blattknoten.
Ein innerer Knoten repräsentiert die zu einem Teilalphabet $\{a, \ldots, b\}$ gehörende Teilsequenz von T (Wurzel entspricht Σ).
Ein innerer Knoten teilt die repräsentierte Sequenz in die Teilsequenzen zu zwei (etwa) gleich großen Teilalphabeten (das niedere $\{a, \ldots, \lfloor (a+b)/2 \rfloor\}$ und das obere $\{\lfloor (a+b)/2 \rfloor + 1, \ldots, b\}$), indem ein Bitvektor die entsprechenden Zugehörigkeiten anzeigt.

Waveletbaum: Beispiel



Waveletbaum: Beispiel

$\Sigma = \{\$_0, a_0, e_0, g_0, i_0, l_1, m_1, n_1, r_1, s_1\}$



Buchstaben hier nur konzeptionell, werden nicht mitgespeichert

Waveletbaum

- Es gibt $\log |\Sigma|$ Ebenen in einem Waveletbaum.
- Jede Ebene speichert insgesamt (verteilt auf alle Knoten) maximal n Bits.
- Ein Waveletbaum benötigt so viel Speicher wie der ursprüngliche Text T, nämlich $n \cdot \lceil \log(|\Sigma|) \rceil$ Bits.
- $\mathcal{O}(|\Sigma|)$ Bits zusätzlich für die Baumstruktur
- Der Text selbst muss nicht mehr gespeichert werden.

Waveletbaum

- Es gibt $\log |\Sigma|$ Ebenen in einem Waveletbaum.
- Jede Ebene speichert insgesamt (verteilt auf alle Knoten) maximal n Bits.
- Ein Waveletbaum benötigt so viel Speicher wie der ursprüngliche Text T, nämlich $n \cdot \lceil \log(|\Sigma|) \rceil$ Bits.
- $\mathcal{O}(|\Sigma|)$ Bits zusätzlich für die Baumstruktur
- Der Text selbst muss nicht mehr gespeichert werden.
- Rank-Datenstrukturen für jeden Knoten, insgesamt $o(n \log |\Sigma|)$ Bits
- Zeichen- und Rank-Anfragen mit Wavelet-Baum in $\mathcal{O}(\log |\Sigma|)$ Zeit

Waveletbaum Rank-Anfrage

- In der Wurzelebene wird eine Rank-Anfrage $rank(\sigma, i)$ für $\sigma \in \Sigma, i < n$ aufgerufen.
- Es wird überprüft, durch welches Bit (`0`, `1`) in dieser Ebene σ kodiert wird.
  - `1`-Bit: Berechne $k = rank(i)$.
  - `0`-Bit: Berechne $k = i - rank(i) + 1$.

Waveletbaum Rank-Anfrage

- In der Wurzelebene wird eine Rank-Anfrage $rank(\sigma, i)$ für $\sigma \in \Sigma, i < n$ aufgerufen.
- Es wird überprüft, durch welches Bit (`0`, `1`) in dieser Ebene σ kodiert wird.
  - `1`-Bit: Berechne $k = rank(i)$.
  - `0`-Bit: Berechne $k = i - rank(i) + 1$.
- Entsprechend der Kodierung ins linke (0) oder rechte (1) Kind gehen und die Grenzen Alphabetgrenzen a, b anpassen.
- In dieser Ebene $rank(\sigma, k)$ anfragen
- In der Ebene, wo zu σ kein Kind mehr existiert, hat man das Ergebnis.

Waveletbaum Rank-Anfrage

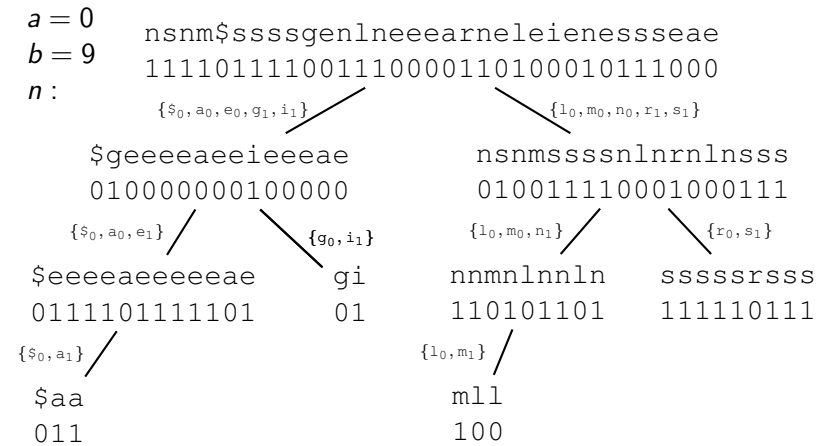
$rank(n, 15) =$

$a = 0$

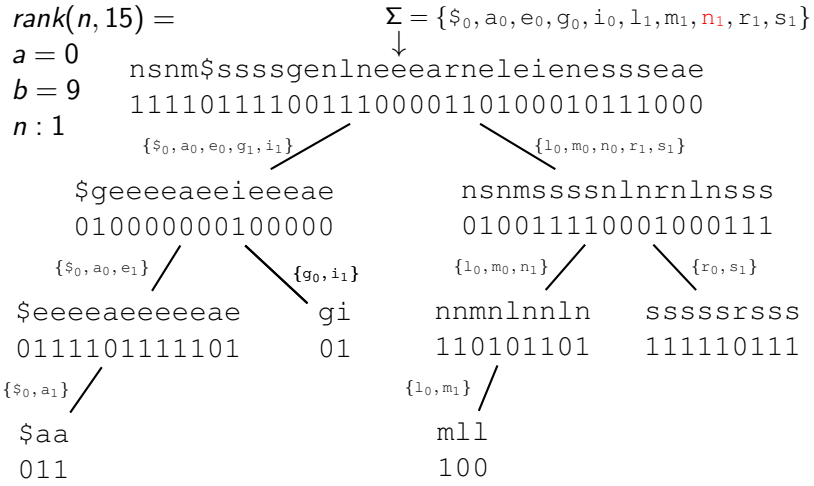
$b = 9$

n :

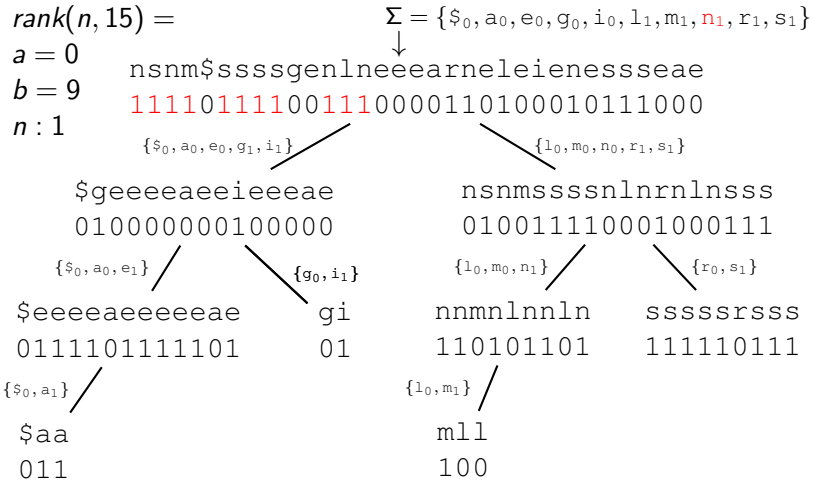
$\Sigma = \{\$_0, a_0, e_0, g_0, i_0, l_1, m_1, n_1, r_1, s_1\}$



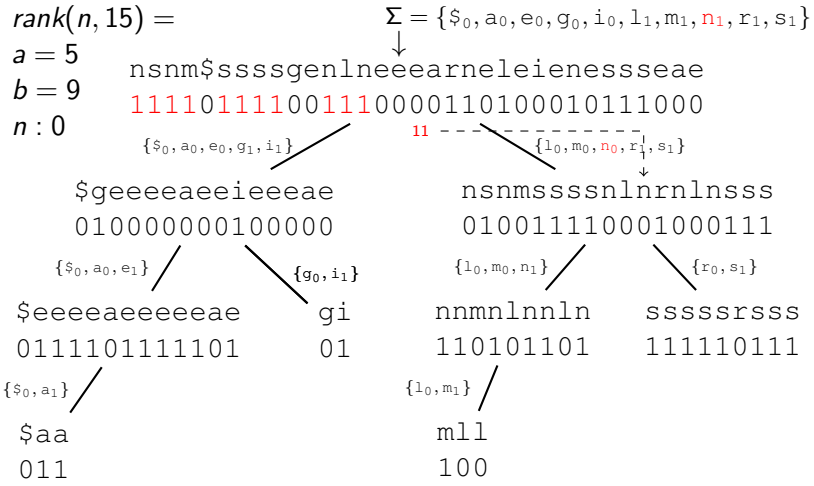
Waveletbaum Rank-Anfrage



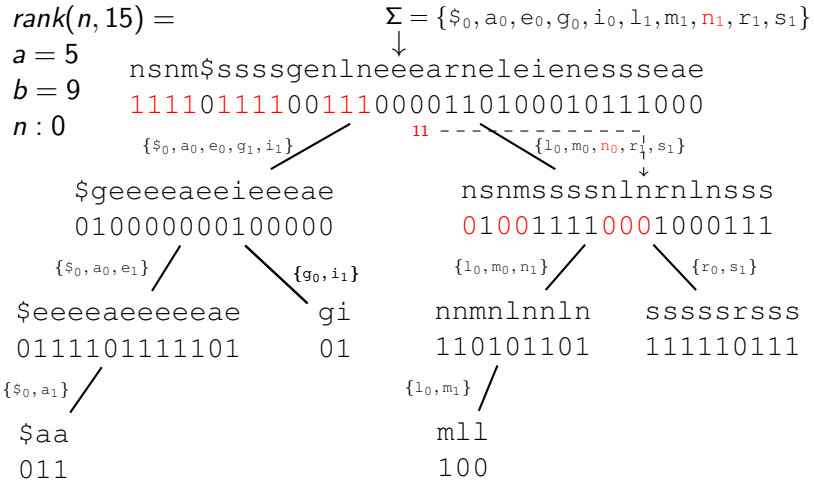
Waveletbaum Rank-Anfrage



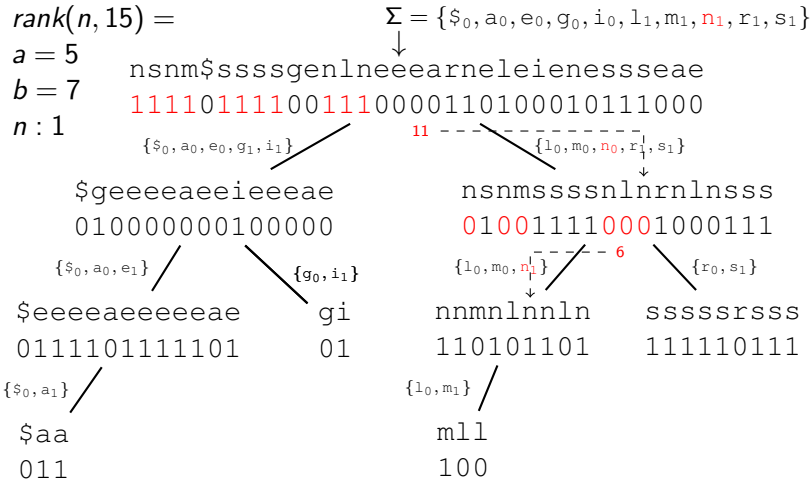
Waveletbaum Rank-Anfrage



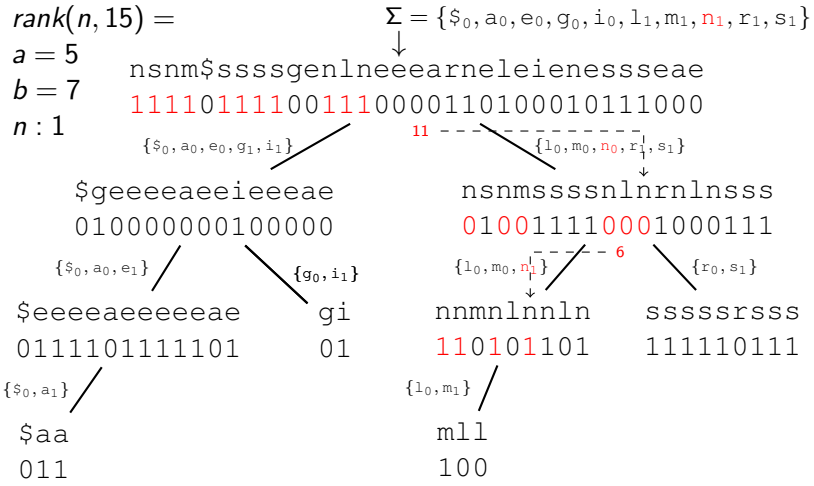
Waveletbaum Rank-Anfrage



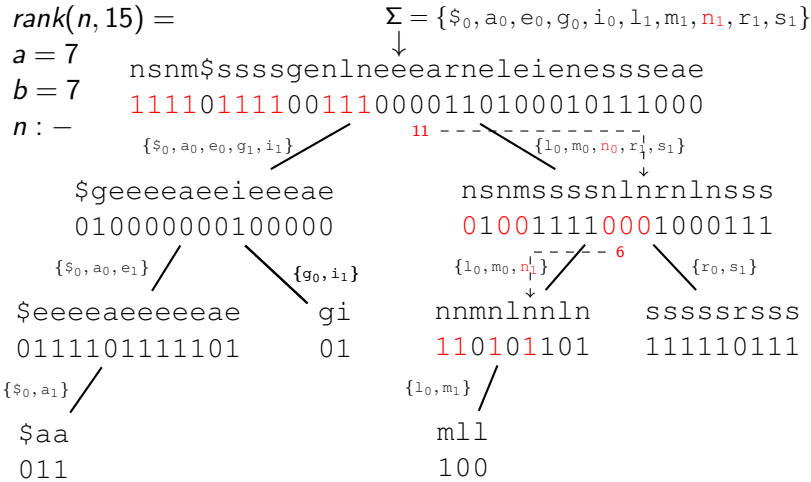
Waveletbaum Rank-Anfrage



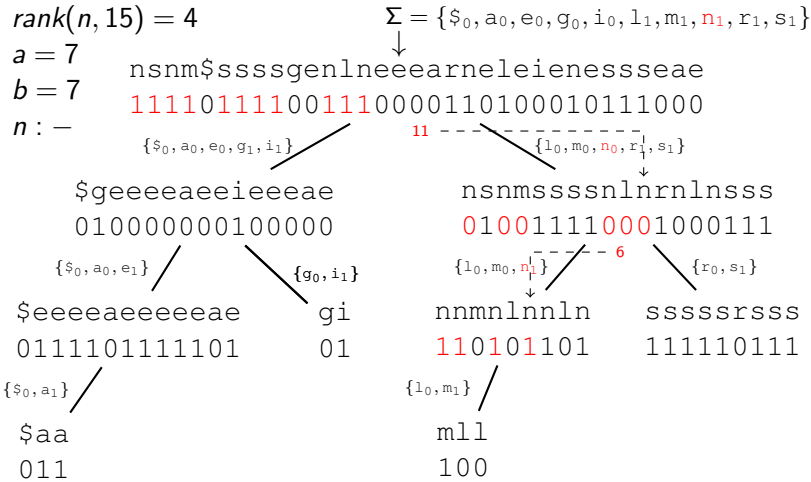
Waveletbaum Rank-Anfrage



Waveletbaum Rank-Anfrage



Waveletbaum Rank-Anfrage



Zusammenfassung: BWT, FM-Index

- Die BWT ist eine invertierbare Permutation des Ausgangstextes.
- Sie "verwandelt" wiederholte Teilstrings in Läufe desselben Buchstaben.
- Die BWT ist invertierbar.
- Das Erstellen oder Invertieren der BWT ist in $\mathcal{O}(n)$ Zeit möglich.
- Mit Hilfe des FM-Indexes (Hilfstabellen *less, Occ* zur BWT) lässt sich die Mustersuche in $\mathcal{O}(m)$ Zeit durchführen (Backward search).
- Der Waveletbaum der BWT reduziert den Speicherverbrauch der *Occ* Tabelle; insgesamt werden $\mathcal{O}(n \log |\Sigma|) + o(n \log |\Sigma|)$ Bits benötigt.