technische universität
dortmund

Algorithmen auf Sequenzen

Volltext-Indizes - Teil 3

Dominik Kopczynski

Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik
TU Dortmund

Eine weitere Index-Datenstruktur

Überblick über die bisherigen Index-Datenstrukturen:

Datenstruktur	*Erstellung*	*Mustersuche*	*Speicherverbrauch*[1]
Suffix-Tree	$\mathcal{O}(n)$	$\mathcal{O}(m)$	$4\lceil \log n \rceil n + \mathcal{O}(n)$
Suffix-Array	$\mathcal{O}(n)$	$\mathcal{O}(m \log n)$	$2\lceil \log n \rceil n$

Wünschenswert wäre eine Datenstruktur, die loglinear Bits Speicherplatz verbraucht und trotzdem $\mathcal{O}(m)$ für die Mustersuche benötigt.

[1] Angabe in Bits, zzgl. Text

Die Burrows-Wheeler Transformation

- Als Grundlage für die Datenstruktur liegt die Burrows-Wheeler Transformation zu Grunde.
- Die BWT wurde ursprünglich von Michael Burrows und David Wheeler zur verlustfreien Kompression entwickelt.
- Gegeben sei ein Text $T = \Sigma^* \circ \$$, dann ist die $bwt(T)$ eine Permutation des ursprünglichen Textes.

Die Burrows-Wheeler Transformation

Definition (Konstruktion der BWT)

Gegeben sei Text T mit $n := |T|$, forme eine $n \times n$ Matrix M, dessen Einträge Zeichen des Alphabets und Zeilen zyklische Shifts des Textes sind. Sortiere die Zeilen lexikographisch. Sei die BWT die letzte Spalte in M.

Die Burrows-Wheeler Transformation

Definition (Konstruktion der BWT)

Gegeben sei Text T mit $n := |T|$, forme eine $n \times n$ Matrix M, dessen Einträge Zeichen des Alphabets und Zeilen zyklische Shifts des Textes sind. Sortiere die Zeilen lexikographisch. Sei die BWT die letzte Spalte in M.

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

Die Burrows-Wheeler Transformation

Definition (Konstruktion der BWT)

Gegeben sei Text T mit $n := |T|$, forme eine $n \times n$ Matrix M, dessen Einträge Zeichen des Alphabets und Zeilen zyklische Shifts des Textes sind. Sortiere die Zeilen lexikographisch. Sei die BWT die letzte Spalte in M.

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

Die Burrows-Wheeler Transformation

Da diese ursprüngliche Definition sehr konzeptionell ist, wird im Allgemeinen eine andere Definition verwendet:

Definition (Konstruktion der BWT)

Sei Text T mit $n := |T|$, pos das Suffix-Array von T, dann ist die bwt die Abbildung $r \rightarrow B_r := T[pos[r] - 1]$.

Die Burrows-Wheeler Transformation

Da diese ursprüngliche Definition sehr konzeptionell ist, wird im Allgemeinen eine andere Definition verwendet:

Definition (Konstruktion der BWT)

Sei Text T mit $n := |T|$, pos das Suffix-Array von T, dann ist die bwt die Abbildung $r \rightarrow B_r := T[pos[r] - 1]$.

i	pos[i]	bwt[i]	$T[pos[i]:]$
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$

Darstellungen der BWT

Abhängig, ob man einen Sentinal verwendet oder nicht, ergeben sich unterschiedliche BWTs:

i	pos[i]	bwt[i]	$T[pos[i]:]$
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$

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

Darstellungen der BWT

Abhängig, ob man einen Sentinal verwendet oder nicht, ergeben sich unterschiedliche BWTs:

i	pos[i]	bwt[i]	$T[pos[i]:]$
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$

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

Im Folgenden wird der Einfachheit halber immer ein Sentinal verwendet.

Eigenschaften der BWT

- Da das Suffix-Array in Linearzeit konstruiert werden kann, beträgt die Laufzeit für die Konstruktion des BWT auch $\mathcal{O}(n)$.
- Bei natürlichsprachlichen Texten beinhaltet die BWT kompressionsfreudige Sequenzstrukturen.
- Das k-te Auftreten eines Zeichens in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffix-Array.
- Die BWT lässt sich ohne zusätzliche Informationen wieder in ihren ursprünglichen Text rücktransformieren.

Eigenschaften der BWT

- Da das Suffix-Array in Linearzeit konstruiert werden kann, beträgt die Laufzeit für die Konstruktion des BWT auch $\mathcal{O}(n)$.
- Bei natürlichsprachlichen Texten beinhaltet die BWT kompressionsfreudige Sequenzstrukturen.
- Das k-te Auftreten eines Zeichens in der BWT entspricht dem k-ten Auftreten des Zeichens im Suffix-Array.
- Die BWT lässt sich ohne zusätzliche Informationen wieder in ihren ursprünglichen Text rücktransformieren.

```
1 def build_bwt(T):
2     n = len(T)
3     pos = SA_IS(T, 128)
4     return [T[pos[i] - 1] for i in range(n)]
```

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 'sch', 'tz' oder 'ie' häufig vorkommen.
- Dementsprechend beinhaltet eine BWT oft lange Blöcke von gleichen Zeichen, z.B.
  ...aaaaaaaarrraaaaaarrrrbbbbbbbaaaba...
- Offensichtlich lässt sich hierauf ein *run-length-encoding* (RLE) sinnvoll anwenden oder diese diese Buchstabenblöcke mit weniger Bits darzustellen.

Die bzip2 Kompression

Die bzip2 Kompression macht sich die BWT zu nutze. Als Eingabe bekommt bzip2 einen Text und eine Blockgröße und arbeitet wie folgt:

1. Für jeden Block berechne die BWT.
2. Wende auf jede BWT eine Move-to-front Transformation an. Die so entstandenen Sequenzen beinhalten sehr viele Nullen.
3. Wende bei 0-Blöcken ein Run-length Encoding an. Die binären Ziffern $\{0, 1\}$ der Längen stellen Sonderzeichen in der folgenden Huffman-Codierung dar.
4. Auf das Resultat wird die Huffman-Codierung angewendet. Häufige Zeichen werden durch kurze Bitfolgen kodiert, seltene Zeichen durch lange Bitfolgen.

Invertierbarkeit

Lemma

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

Invertierbarkeit

Lemma

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

i	pos[i]	bwt[i]	$T[pos[i]:]$
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

Beweis

Man Betrachte zwei beliebige gleiche Zeichen c_1, c_2 in der BWT, wobei c_1 vor c_2 steht. Da die Suffixe von c_1 und c_2 lexikogriphisch sortiert sind, ist die Reihenfolge von c_1 und c_2 in der BWT folglich auch die gleiche wie im Suffix-Array. □

Invertierbarkeit

Bemerkenswert ist, dass es für alle vier Schritte in der bzip2 Kompression auch verlustfreie invertierbare Methoden gibt. Somit kann der ursprüngliche Text wieder korrekt hergestellt werden.

Invertierbarkeit

Bemerkenswert ist, dass es für alle vier Schritte in der bzip2 Kompression auch verlustfreie invertierbare Methoden gibt. Somit kann der ursprüngliche Text wieder korrekt hergestellt werden.

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n

Durch eine *stabile* Sortierung der Buchstaben stehen wieder die korrekten Nachfolger unterhalb der BWT Zeichen.

Invertierbarkeit

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	4	0	5	6	3	1	2

1) Es wird mit dem Sentinal begonnen, dieser Stand ursprünglich an Position 4. Sein Nachfolger ist das b. Der Originaltext beginnt mit diesem Zeichen.

$$T = \texttt{b}$$

Invertierbarkeit

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	4	0	5	6	3	1	2

2) Das b stand an Position 3. Der Nachfolger von b ist ein a. Das Zeichen wird zum Text hinzugefügt.

$$T = \texttt{ba}$$

Invertierbarkeit

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	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

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	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

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	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

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	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

Um vom nachfolgenden Zeichen zu seinem Nachfolger zu kommen, muss mitgespeichert werden, wo dieses Zeichen ursprünglich in der BWT stand:

i	0	1	2	3	4	5	6
$bwt[i]$	a	n	n	b	$	a	a
sortiert	$	a	a	a	b	n	n
p	4	0	5	6	3	1	2

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

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

Implementierung

```
1  def bwt_bucket_sort(bwt):
2      bwt = bytes(bwt, 'utf-8')
3      counts, p = [0] * 128, [0] * len(bwt)
4      for c in bwt: counts[c] += 1
5      bkt = getBuckets(counts, True)
6      for i, c in enumerate(bwt):
7          p[bkt[c]] = i; bkt[c] += 1
8      return p
9
10 def inverse_bwt(bwt):
11     p, n = bwt_bucket_sort(bwt), len(bwt)
12     T, r = [''] * n, p[0]
13     for i in range(n):
14         r = p[r]; T[i] = bwt[r]
15     return ''.join(T)
```

Laufzeitanalyse

- Das Berechnen der Bucket-Startpositionen dauert $\mathcal{O}(n)$.
- Die Berechnung von Array p dauert $\mathcal{O}(n)$.
- Das Ablaufen und Anhängen der Nachfolger dauert $\mathcal{O}(n)$.

Laufzeitanalyse

- Das Berechnen der Bucket-Startpositionen dauert $\mathcal{O}(n)$.
- Die Berechnung von Array p dauert $\mathcal{O}(n)$.
- Das Ablaufen und Anhängen der Nachfolger dauert $\mathcal{O}(n)$.
- Insgesamt beträgt die Laufzeit $\mathcal{O}(n)$.

Mustersuche mit der BWT

- Bei der Mustersuche mit der BWT wird versucht im *pos*-Array das Intervall $[L, R]$ das alle Vorkommen des Musters P enthält, zu finden.
- Bei diesem Verfahren handelt es sich um die *Backward Search*.
- Die Backward Search simuliert *nicht* die Suche im Suffix-Tree.
- Das Intervall $[L, R]$ ist wie folgt definiert:

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

für alle $0 \leq i < |T|$.

- Es gibt dann R - L + 1 Muster im Text, *Achtung: R inklusive*.

FM-Index

- Ist $P = \epsilon$, dann gilt: $L = 0, R = n - 1$.
- Zur Suche sind zwei Hilfstabellen nötig:
  - *less*[*c*]: gibt für Zeichen c im Text an, wie viele Zeichen es im Text gibt, die lexikoraphisch kleiner sind als c.
  - *Occ*[c, i]: zählt die Vorkommen von Zeichen c im Präfix $bwt(T)[: i + 1]$.
- Beide Hilfstabellen werden als *FM-Index* bezeichnet.

FM-Index

- Ist $P = \epsilon$, dann gilt: $L = 0, R = n - 1$.
- Zur Suche sind zwei Hilfstabellen nötig:
  - *less*[*c*]: gibt für Zeichen *c* im Text an, wie viele Zeichen es im Text gibt, die lexikoraphisch kleiner sind als *c*.
  - *Occ*[c, i]: zählt die Vorkommen von Zeichen *c* im Präfix $bwt(T)[: i + 1]$.
- Beide Hilfstabellen werden als *FM-Index* bezeichnet.

Beispiel: $T =$ `banana$`, $bwt(T) =$ `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 bekannte Intervall zu P und $[L^+, R^+]$ das gesuchte Intervall zu P^+, dann gilt:

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

Backward Search

Beweis

- *Zu Beginn ist* $P = \epsilon$*, somit ist das Muster im gesamten Intervall vorhanden. Es gilt:* $L = 0, R = n - 1$.
- *Da* P^+ *mit einem Zeichen* c *beginnt, ist klar, dass das Subintervall* $[L_c, R_c]$ *gesucht ist, 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. Die Berechnung stimmt für* $|P^+| = 1$.

Backward Search

Beweis

- *Sei nun $P^+ = cP$, dann gibt $[L, R]$ das Intervall im Suffix-Array an, an dem die Suffixe mit P beginnen.*
- *Innerhalb dieses Intervalls muss überprüft werden, wie viele Vorgänger ein c sind.*
- *Da bekannt ist, dass das k-te Zeichen c in der BWT auch dem k-ten Zeichen c im Suffix-Array entspricht, muss an Stelle L gezählt werden, wie oft c im Präfix $bwt[: L]$ vorkam, also $Occ[c, L-1]$.*

Backward Search

Beweis

- *Diese Anzahl an cs muss dann im* $[L_c, R_c]$ *Intervall für* L_c *übersprungen werden, also* $L^+ = L_c + Occ[c, L-1]$.
- *Um die neue rechte Grenze zu erhalten, muss bekannt sein wie oft Zeichen c bis zur Position R vorkam, also* $Occ[c, R]$.
- *Da die rechte Grenze immer einschließlich ist, gilt:* $R^+ = less[c] + Occ[c, R] - 1$. □

Backward Search

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

bwt(T)=   nsnm$ssssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        0
R=        32
```

Backward Search

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

$bwt(T)$= nsnm$ssssgenlneeearneleienessseae

$T[pos]$= $aaeeeeeeeeeeegillmnnnnnnrsssssssss

L= 0 $less[s] + Occ[s, -1]$

R= 32 $less[s] + Occ[s, 32] - 1$

Backward Search

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

$bwt(T)$= nsnm$ssssgenlneeearneleienessseae

$T[pos]$= $aaeeeeeeeeeeegillmnnnnnnrsssssssss

L= 0 $\quad\quad 25 + Occ[\mathrm{s}, -1]$

R= 32 $\quad\quad 25 + Occ[\mathrm{s}, 32] - 1$

Backward Search

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

bwt(T)=   nsnm$sssgenlneeearneleienesssseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        0                 25 + 0
R=        32                25 + Occ[s, 32] − 1
```

Backward Search

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

bwt(T)=   nsnm$ssssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        25
R=        32
```

Backward Search

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

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

Backward Search

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

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

Backward Search

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

bwt(T)=   nsnm$sssgenlneeearneleienesssseae
T[pos]=   $aaeeeeeeeeeegillmnnnnnnrsssssssss
L=        25          25 + 5
R=        32          25 + Occ[s, 32] − 1
```

Backward Search

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

bwt(T)=   nsnm$ssssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        30
R=        32
```

Backward Search

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

bwt(T)=   nsnm$sssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeegillmnnnnnnrsssssssss
L=  30                less[e] + Occ[e, 29]
R=  32                less[e] + Occ[e, 32] − 1
```

Backward Search

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

$bwt(T)$= nsnm$ssssgenlneeearneleienessseae

$T[pos]$= $aaeeeeeeeeeeegillmnnnnnnrsssssssss

L= 30 $\quad 3 + Occ[\mathrm{e}, 29]$

R= 32 $\quad 3 + Occ[\mathrm{e}, 32] - 1$

Backward Search

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

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

Backward Search

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

bwt(T)=   nsnm$ssssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        11
R=        12
```

Backward Search

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

$bwt(T)$= nsnm$sssgenlneeearneleienessseae

$T[pos]$= $aaeeeeeeeeeeegillmnnnnnnrsssssssss

L= 11 $less[\mathtt{l}] + Occ[\mathtt{l}, 10]$

R= 12 $less[\mathtt{l}] + Occ[\mathtt{l}, 12] - 1$

Backward Search

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

bwt(T)=   nsnm$ssssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
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(T)=   nsnm$sssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        11              15 + 0
R=        12              15 + Occ[l, 12] − 1
```

Backward Search

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

bwt(T)=   nsnm$sssgenlneeearneleienessseae
T[pos]=   $aaeeeeeeeeeeegillmnnnnnnrsssssssss
L=        15
R=        15
```

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

Backward Search

- Pro Zeichen des Patterns werden pro Grenze zwei Lookups durchgeführt, also $\mathcal{O}(1)$.
- Insgesamt erfolgt die Patternsuche in $\mathcal{O}(m)$.
- Der Speicherplatz sieht wie folgt aus:
  - *less*: $\mathcal{O}(|\Sigma|\lceil\log(n)\rceil)$
  - *Occ*: $\mathcal{O}(|\Sigma|n\lceil\log(n)\rceil)$

Backward Search

- Pro Zeichen des Patterns werden pro Grenze zwei Lookups durchgeführt, also $\mathcal{O}(1)$.
- Insgesamt erfolgt die Patternsuche in $\mathcal{O}(m)$.
- Der Speicherplatz sieht wie folgt aus:
  - *less*: $\mathcal{O}(|\Sigma|\lceil\log(n)\rceil)$
  - *Occ*: $\mathcal{O}(|\Sigma|n\lceil\log(n)\rceil)$
- *Occ* verbraucht ziemlich viel Speicherplatz, ist das nötig?

Backward Search

Rank-Datenstruktur für *Occ* verwenden.
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Occ	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

Backward Search

Rank-Datenstruktur für *Occ* verwenden.
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Occ	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 eine eigene Rank-Datenstruktur.
Speicherverbrauch: $\mathcal{O}\big(|\Sigma|(n + \lceil n/W \rceil \lceil \log(n) \rceil)\big)$

Backward Search

Rank-Datenstruktur für *Occ* verwenden.
Beispiel: $T = \texttt{banana\$}$, $bwt = \texttt{annb\$aa}$

Occ	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 eine eigene Rank-Datenstruktur.
Speicherverbrauch: $\mathcal{O}\big(|\Sigma|(n + \lceil n/W \rceil \lceil \log(n) \rceil)\big)$

Geht es noch besser?

Wavelet-Tree

Definition

Gegeben sei Text T, $n := |T|$ und Alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{s-1}\}$, dann ist ein Wavelet-Tree ein balancierter Binärbaum mit n Bits pro Ebene und bis zu n Bits in der untersten Ebene. Es gibt $|\Sigma|$ Blattknoten. Ein innerer Knoten beschreibt eine Subsequenz von T über dem Teilalphabet $\{a, \ldots, b\}$.

Wavelet-Tree

Definition

Gegeben sei Text T, $n := |T|$ und Alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{s-1}\}$, dann ist ein Wavelet-Tree ein balancierter Binärbaum mit n Bits pro Ebene und bis zu n Bits in der untersten Ebene. Es gibt $|\Sigma|$ Blattknoten. Ein innerer Knoten beschreibt eine Subsequenz von T über dem Teilalphabet $\{a, \ldots, b\}$. Beginnend von der Wurzel besitzt das linke Kind das Teilalphabet $\{a, \ldots, \lfloor (a+b)/2 \rfloor\}$ und das rechte Kind das Teilalphabet $\{\lfloor (a+b)/2 \rfloor + 1, \ldots, b\}$.

Wavelet-Tree

Definition

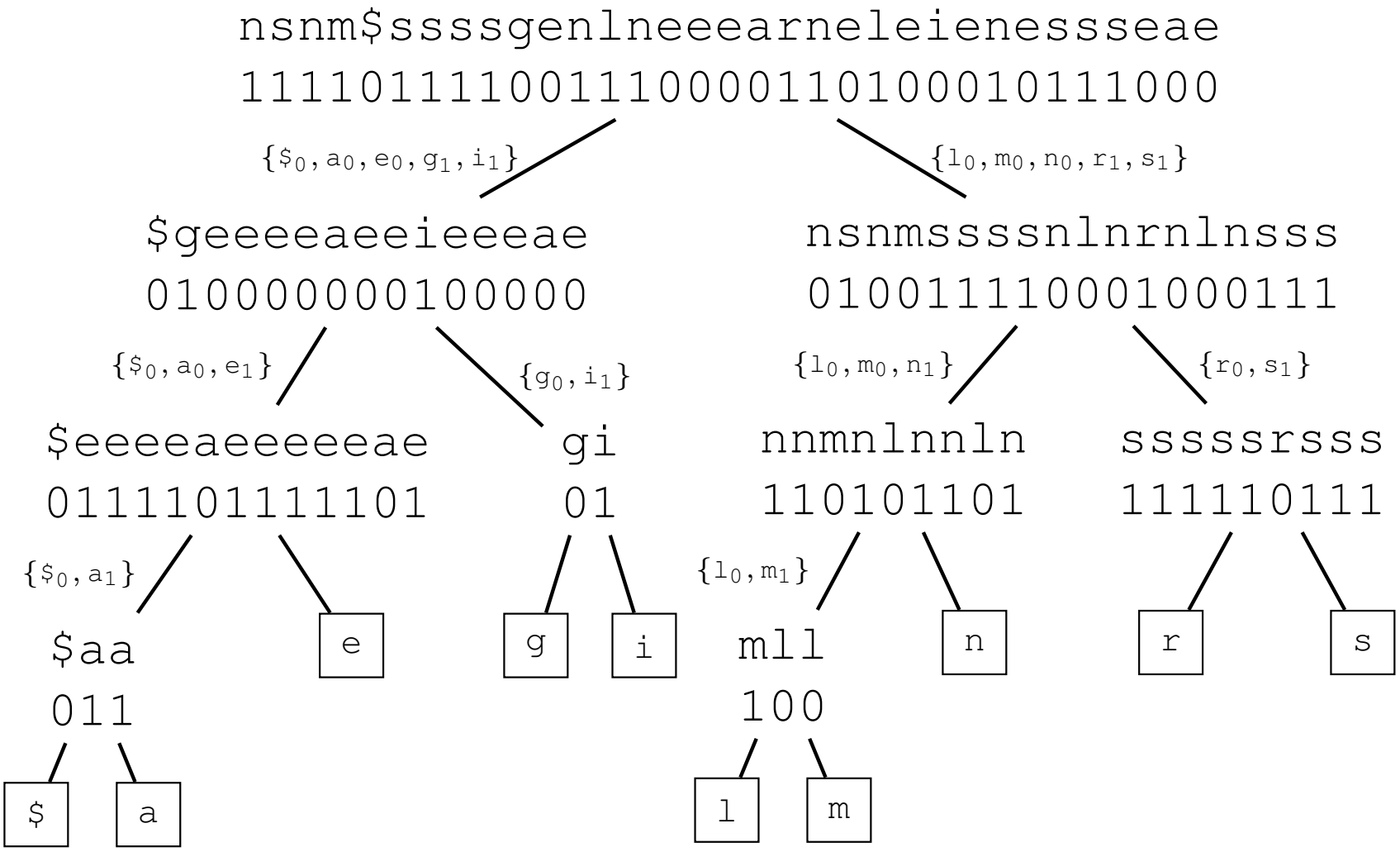
Gegeben sei Text T, $n := |T|$ und Alphabet $\Sigma = \{\sigma_0, \ldots, \sigma_{s-1}\}$, dann ist ein Wavelet-Tree ein balancierter Binärbaum mit n Bits pro Ebene und bis zu n Bits in der untersten Ebene. Es gibt $|\Sigma|$ Blattknoten. Ein innerer Knoten beschreibt eine Subsequenz von T über dem Teilalphabet $\{a, \ldots, b\}$. Beginnend von der Wurzel besitzt das linke Kind das Teilalphabet $\{a, \ldots, \lfloor (a+b)/2 \rfloor\}$ und das rechte Kind das Teilalphabet $\{\lfloor (a+b)/2 \rfloor + 1, \ldots, b\}$. Für Knoten k sei die Bitsequenz B mit Teilalphabet $\{a, \ldots, b\}$ und $a < b$:

$$B_k[i] = \begin{cases} 0, & \text{falls } T[\mathcal{I}_i] \leq \sigma_{\lfloor (a+b)/2 \rfloor}, \\ 1, & \text{sonst} \end{cases}$$

für alle $0 \leq i < |\mathcal{I}|, \mathcal{I} = \{j \mid T[j] \in \{a, \ldots, b\}\}$.

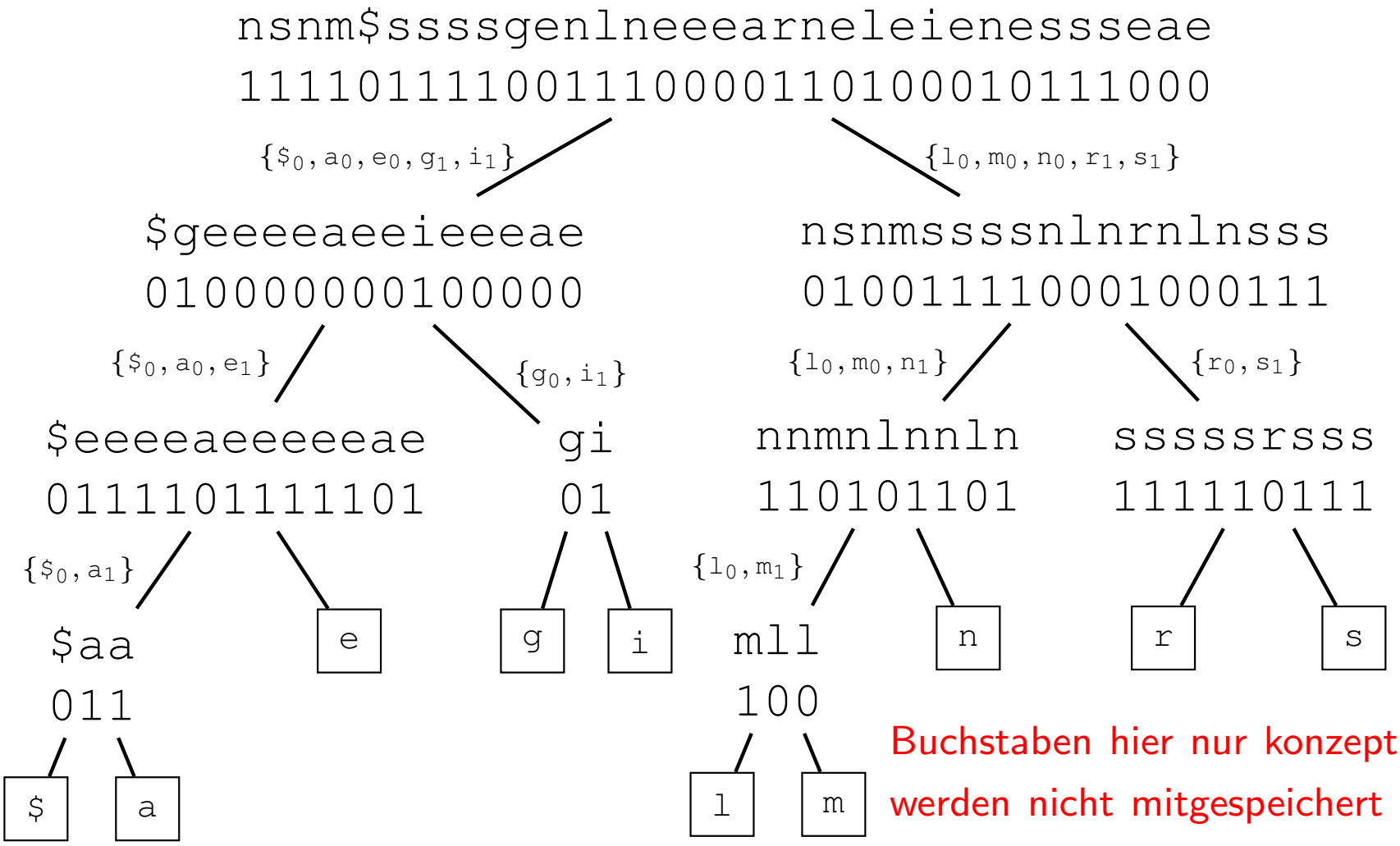
Wavelet-Tree

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



Wavelet-Tree

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



Wavelet-Tree

- Es gibt $\log(|\Sigma|)$ Ebenen in einem Wavelet-Tree.
- Ein Wavelet-Tree verbraucht genauso viel Speicher, wie der ursprüngliche Text T, nämlich $n \cdot \lceil \log(|\Sigma|) \rceil$ Bits.
- Für den gesamten Tree müssen noch $\mathcal{O}(|\Sigma|)$ Informationen für die Baumstruktur mitgespeichert werden.

Wavelet-Tree

- Es gibt $\log(|\Sigma|)$ Ebenen in einem Wavelet-Tree.
- Ein Wavelet-Tree verbraucht genauso viel Speicher, wie der ursprüngliche Text T, nämlich $n \cdot \lceil \log(|\Sigma|) \rceil$ Bits.
- Für den gesamten Tree müssen noch $\mathcal{O}(|\Sigma|)$ Informationen für die Baumstruktur mitgespeichert werden.
- Durch die bekannten weiteren Hilfsarrays kann eine Rank-Anfrage in $\mathcal{O}(\log(|\Sigma|))$ durchgeführt werden.
- Pro Knoten müsste ein Hilfsarray für die kummulierten Summen mitgespeichert werden.

Wavelet-Tree

- Es gibt $\log(|\Sigma|)$ Ebenen in einem Wavelet-Tree.
- Ein Wavelet-Tree verbraucht genauso viel Speicher, wie der ursprüngliche Text T, nämlich $n \cdot \lceil \log(|\Sigma|) \rceil$ Bits.
- Für den gesamten Tree müssen noch $\mathcal{O}(|\Sigma|)$ Informationen für die Baumstruktur mitgespeichert werden.
- Durch die bekannten weiteren Hilfsarrays kann eine Rank-Anfrage in $\mathcal{O}(\log(|\Sigma|))$ durchgeführt werden.
- Pro Knoten müsste ein Hilfsarray für die kummulierten Summen mitgespeichert werden.
- Aus einem Wavelet-Tree kann T wieder rekonstriert werden. T muss also nicht mehr mitgespeichert werden.
- Der Gesamtspeicher beträgt also: $\mathcal{O}\big(|\Sigma| + \lceil \log(|\Sigma|) \rceil \cdot (n + \lceil n/W \rceil \ \lceil \log(n) \rceil)\big)$.

Wavelet-Tree 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 ($\texttt{0}$, $\texttt{1}$) in dieser Ebene σ kodiert wird.
  - $\texttt{1}$-Bit: $rank(i)$.
  - $\texttt{0}$-Bit: $i - rank(i) + 1$.

Wavelet-Tree 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 ($\texttt{0}$, $\texttt{1}$) in dieser Ebene σ kodiert wird.
  - $\texttt{1}$-Bit: $rank(i)$.
  - $\texttt{0}$-Bit: $i - rank(i) + 1$.
- Entsprechend der Kodierung ins 0er oder 1er Kind gehen und die Grenzen a, b anpassen. In dieser Ebene *rank* Anfrage mit Ergebnis vom oberen Ergebnis aufrufen.
- So lange den Baum traversieren, bis für die Grenzen $a = b$ gilt.

Wavelet-Tree Rank-Anfrage

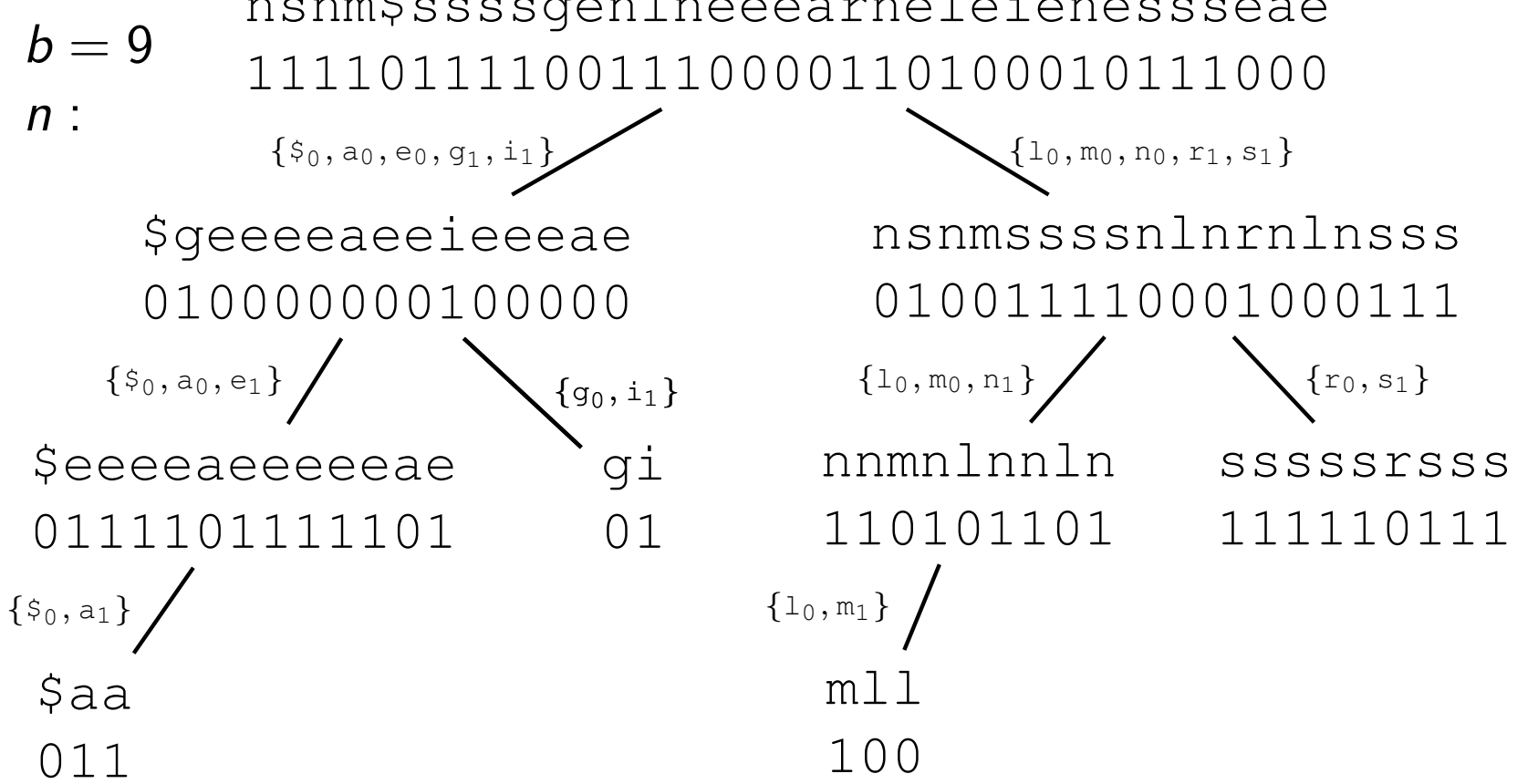
$rank(n, 15) =$

$a = 0$

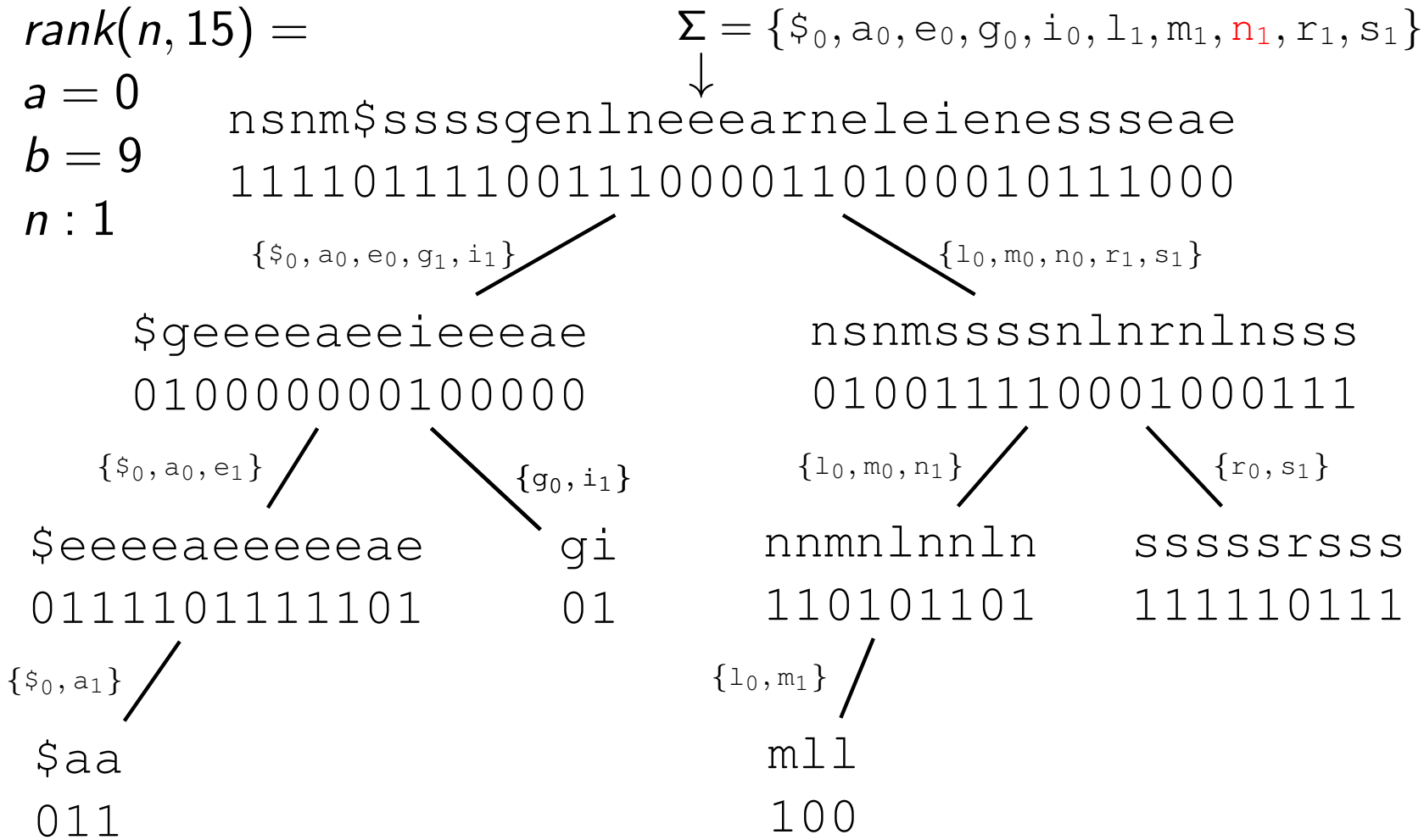
$b = 9$

n :

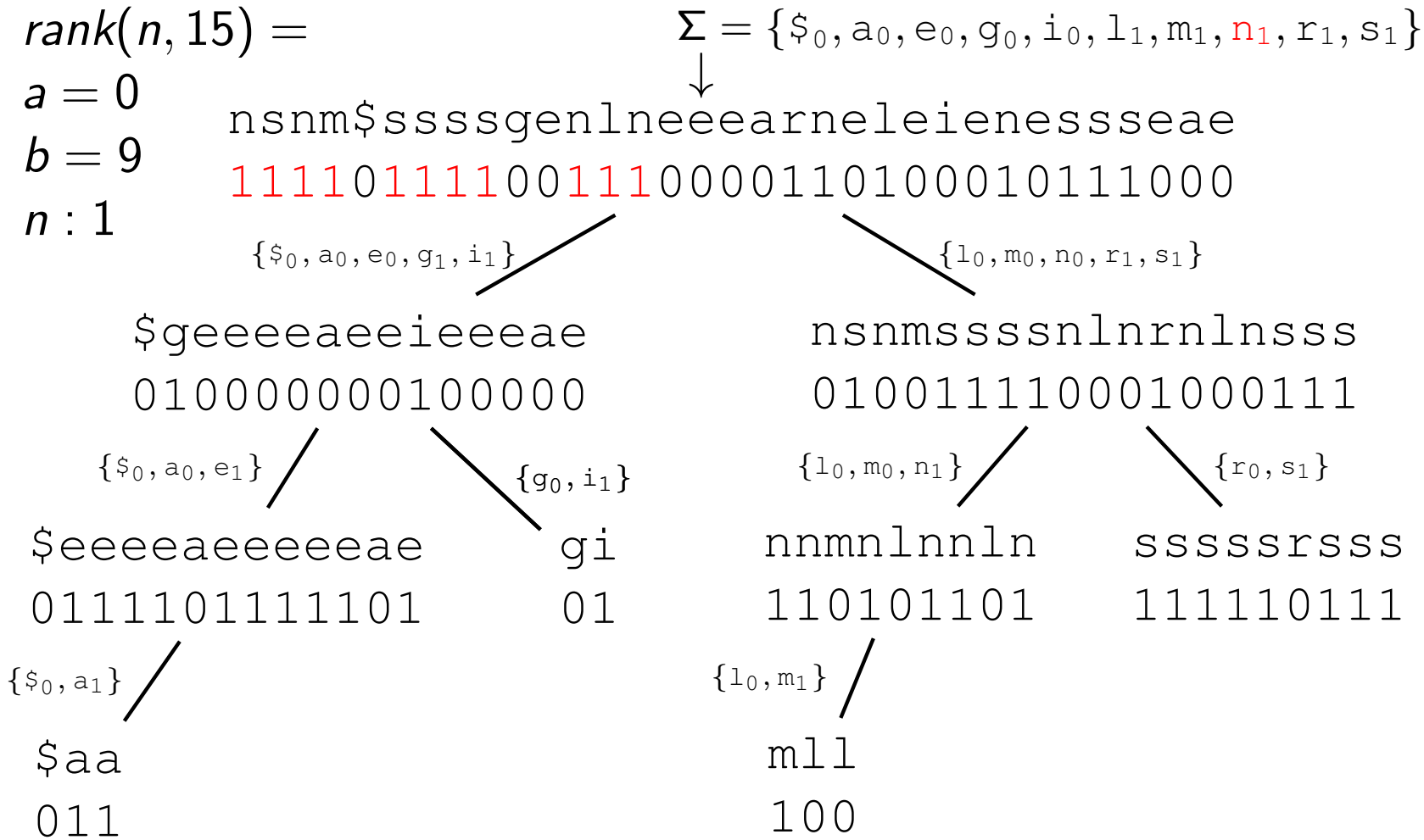
$\Sigma = \{\$_0, a_0, e_0, g_0, i_0, l_1, m_1, n_1, r_1, s_1\}$



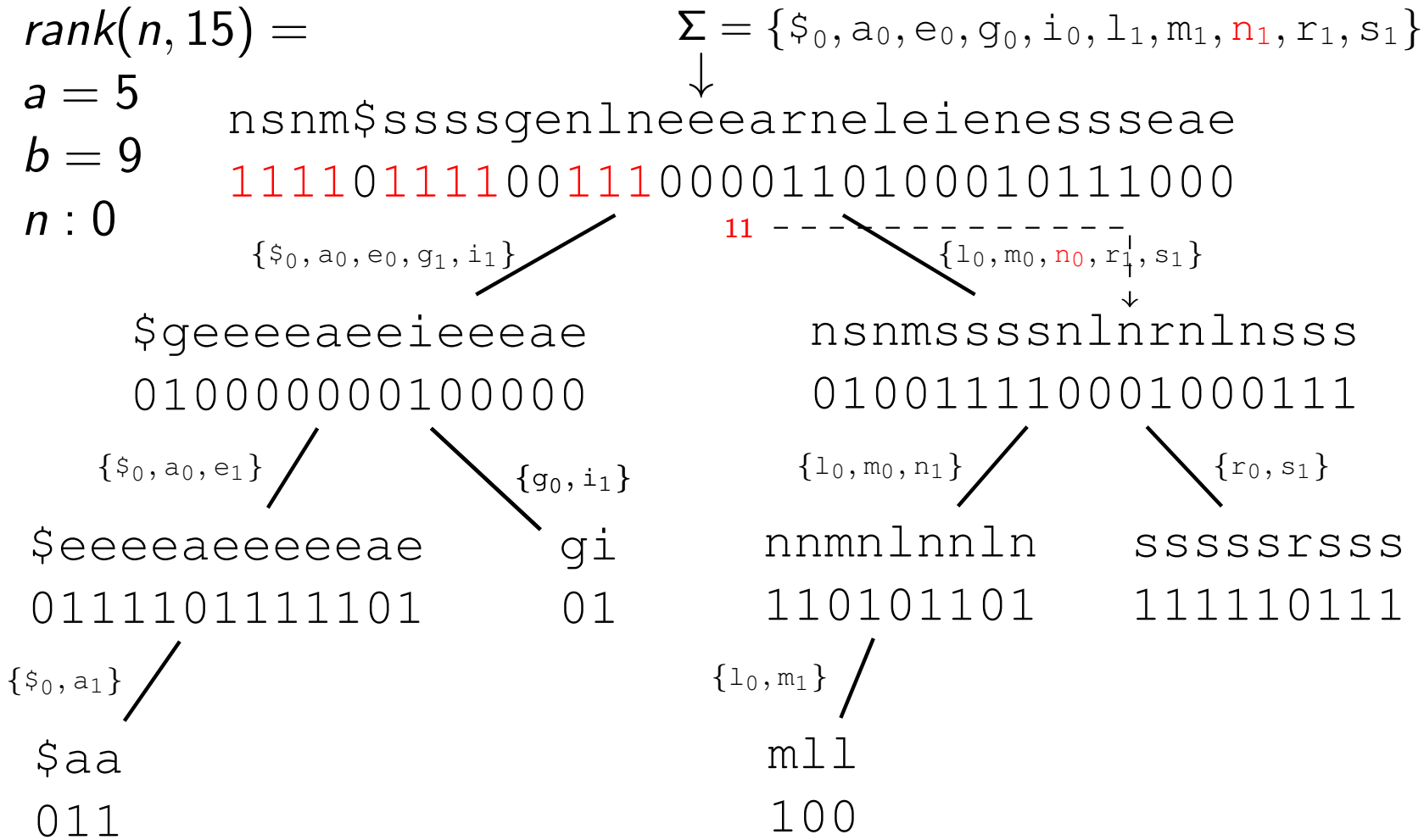
Wavelet-Tree Rank-Anfrage



Wavelet-Tree Rank-Anfrage



Wavelet-Tree Rank-Anfrage



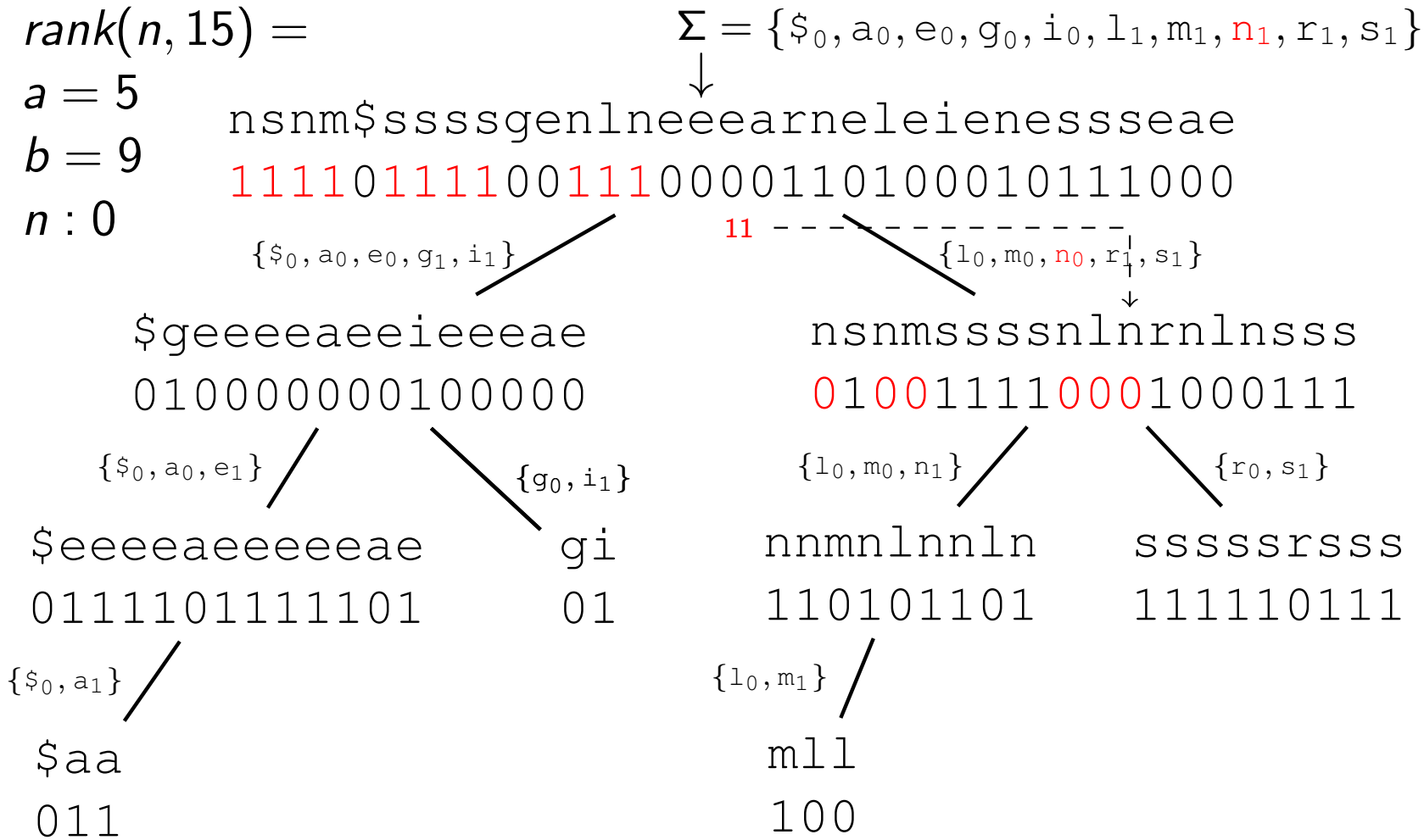
Wavelet-Tree Rank-Anfrage

$rank(n, 15) =$

$a = 5$

$b = 9$

$n : 0$



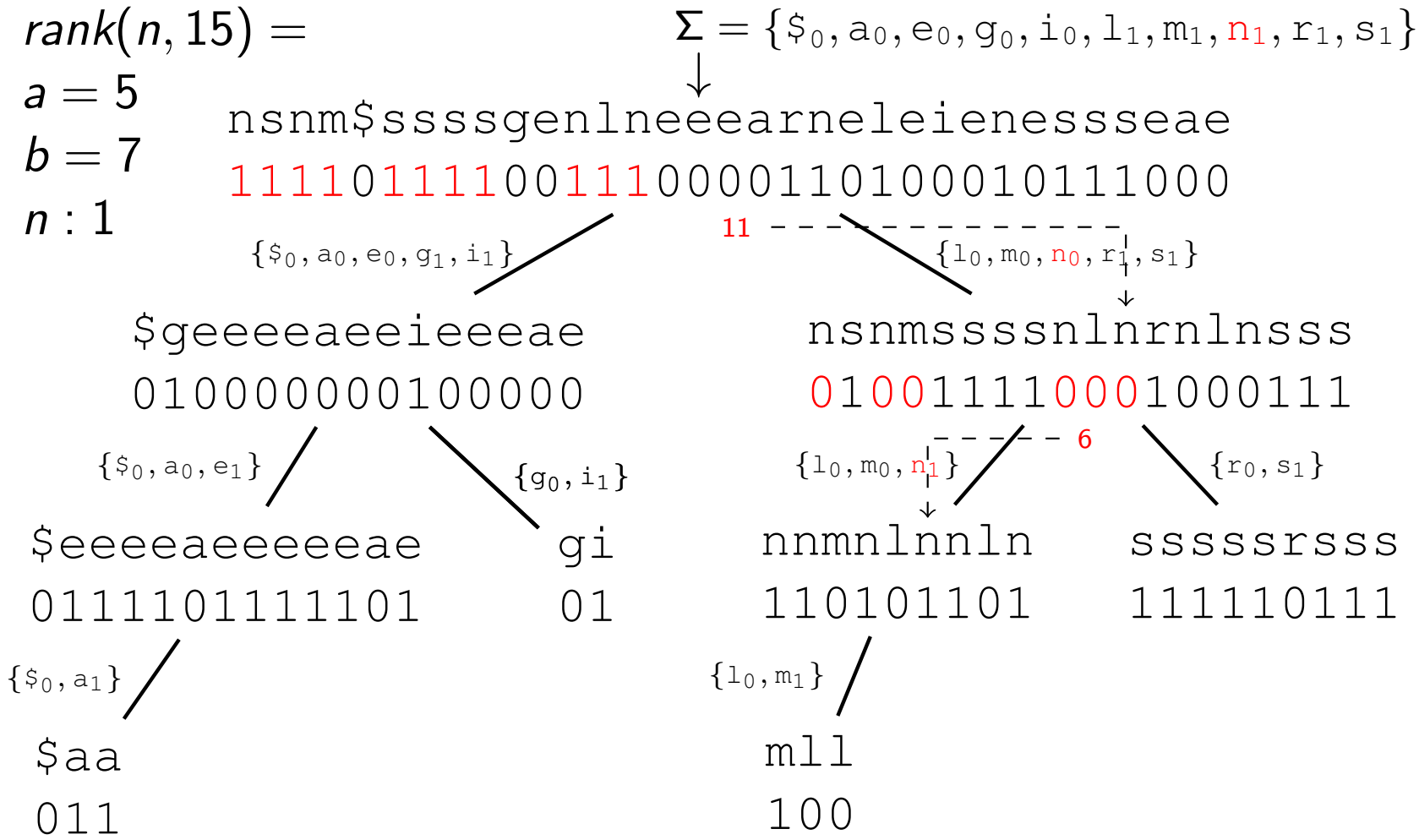
Wavelet-Tree Rank-Anfrage

$rank(n, 15) =$

$a = 5$

$b = 7$

$n : 1$



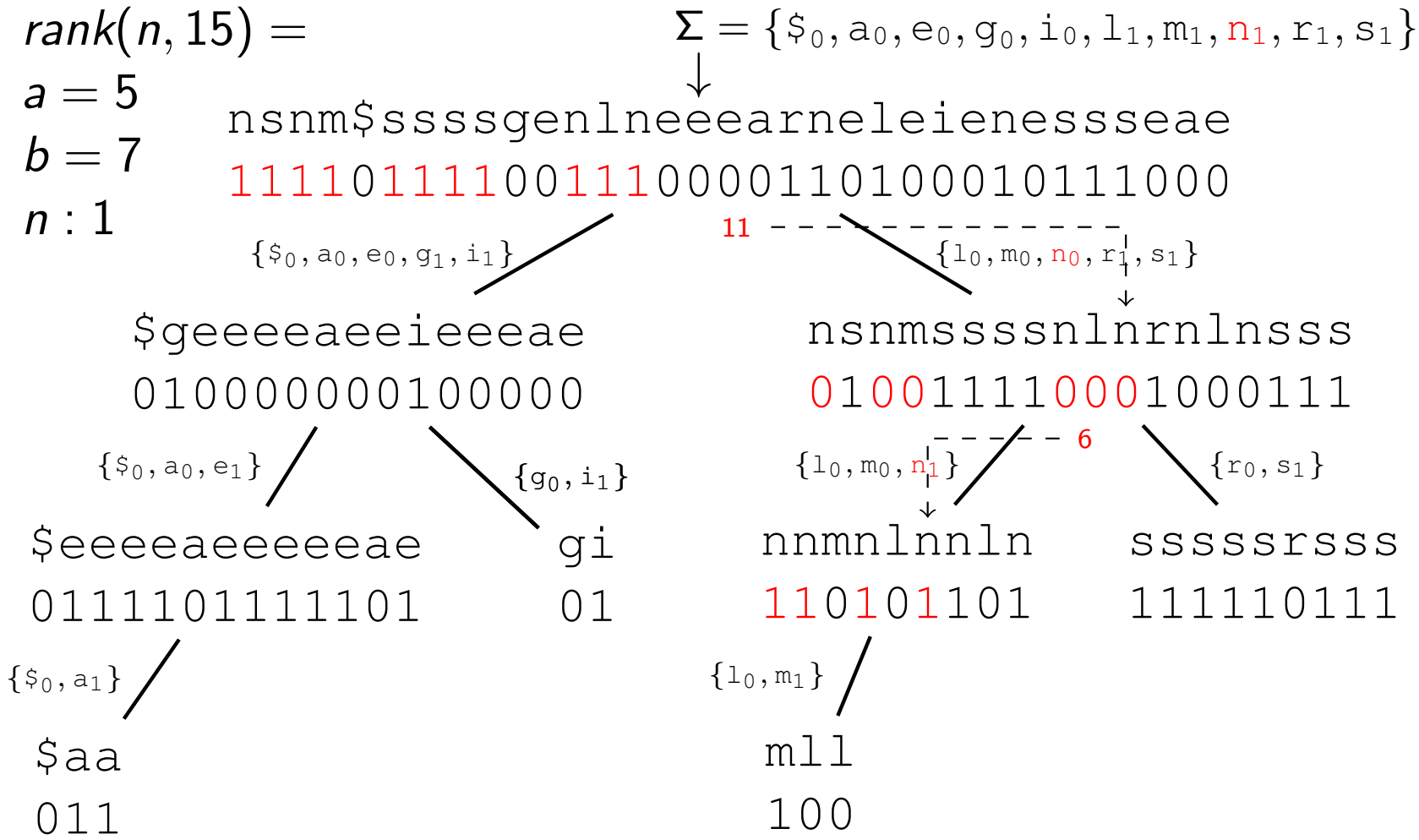
Wavelet-Tree Rank-Anfrage

$rank(n, 15) =$

$a = 5$

$b = 7$

$n : 1$



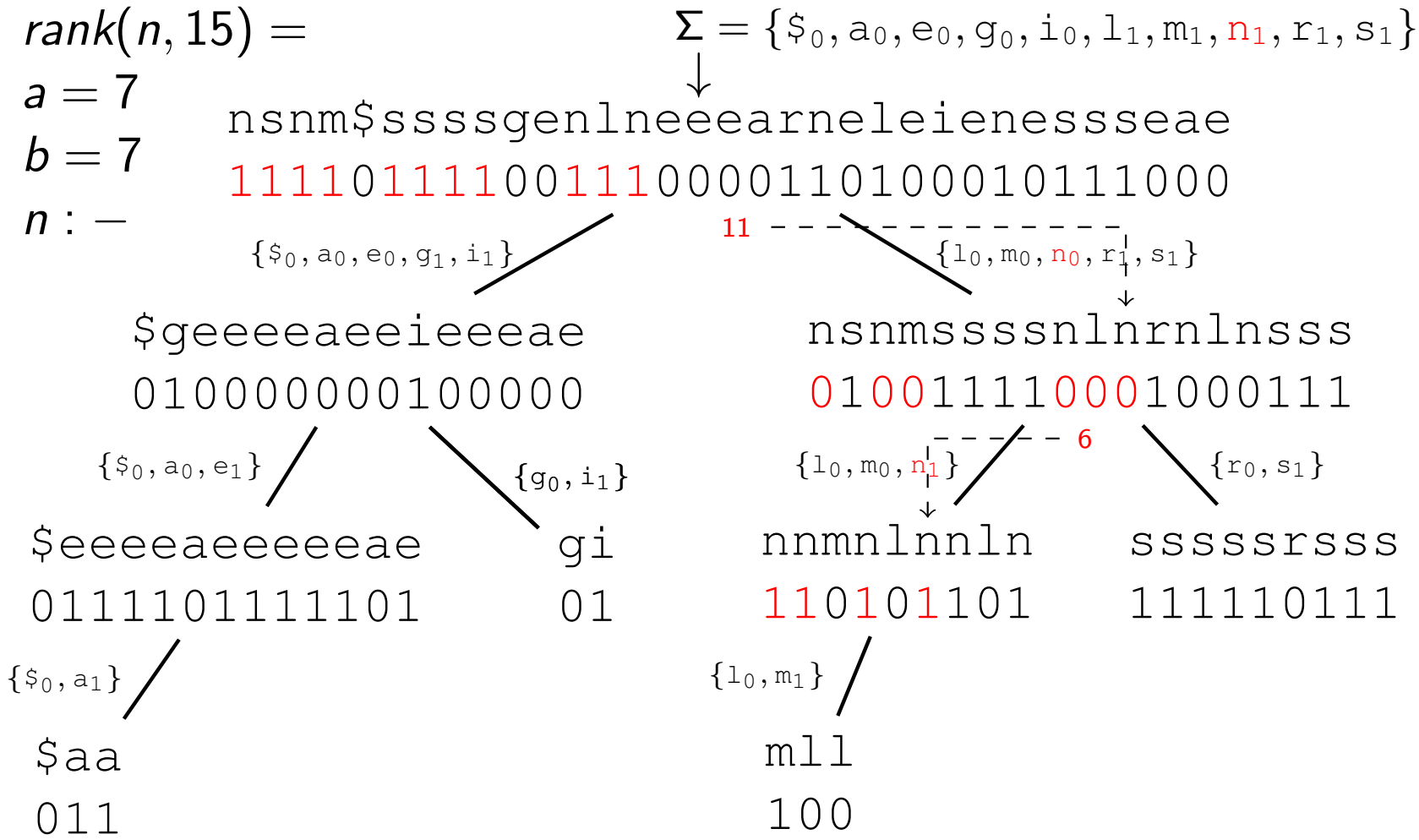
Wavelet-Tree Rank-Anfrage

$rank(n, 15) =$

$a = 7$

$b = 7$

$n : -$



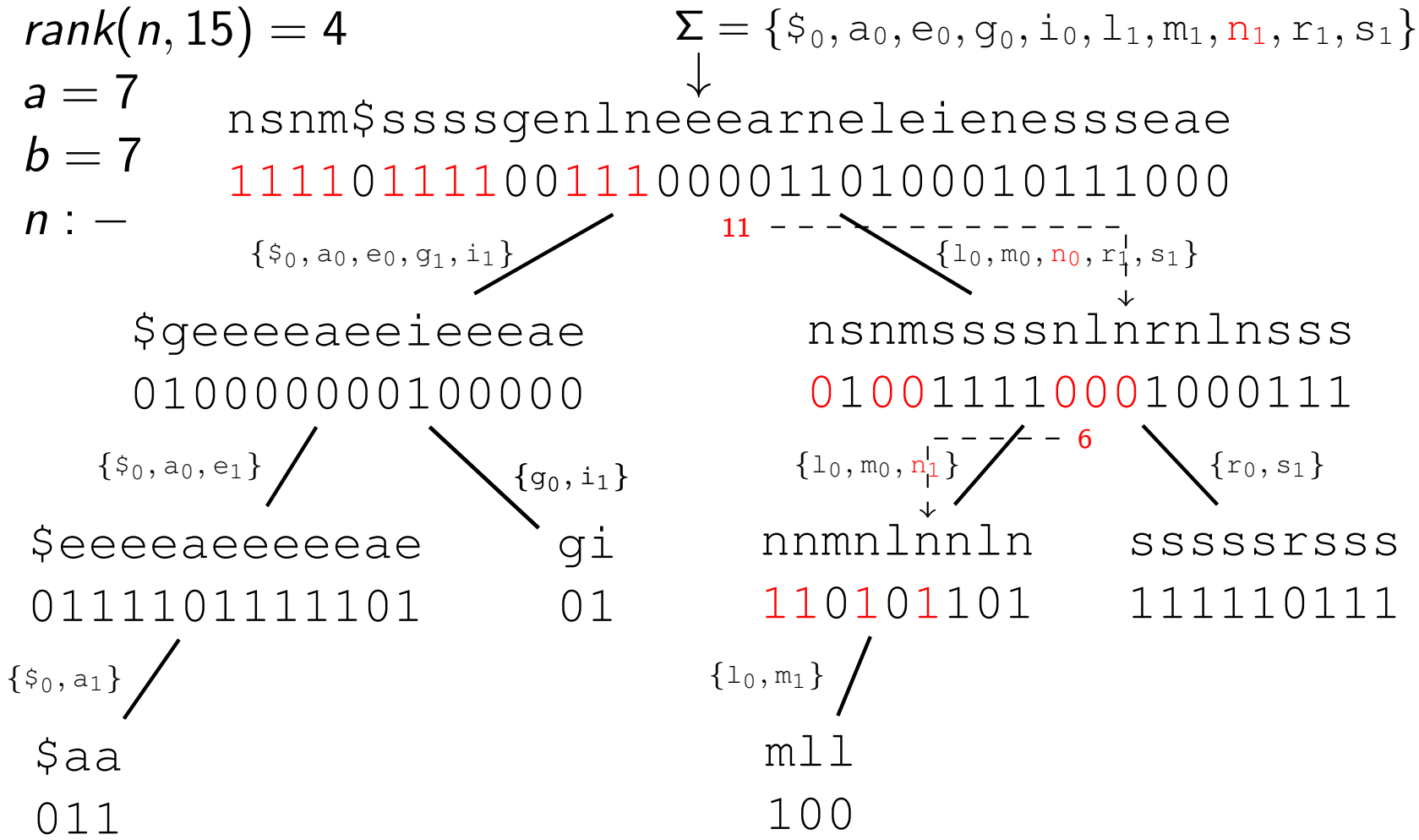
Wavelet-Tree Rank-Anfrage

$rank(n, 15) = 4$

$a = 7$

$b = 7$

$n : -$



Zusammenfassung

- Die BWT ist eine invertierbare Permutation eines Ausgangstextes.
- Das Erstellen der BWT hat eine Laufzeit von $\mathcal{O}(n)$.
- Mit der BWT lässt sich ein FM-Index erstellen.
- Mit Hilfe des FM-Indexes lässt sich die Patternsuche in Laufzeit $\mathcal{O}(m)$ durchführen.
- Datenstrukturen wie Wavelet-Trees reduzieren den Speicherverbrauch der *Occ* Tabelle erheblich auf $\mathcal{O}\big(|\Sigma| + \lceil \log(|\Sigma|) \rceil \cdot (n + \lceil n/W \rceil \ \lceil \log(n) \rceil)\big)$