Karlsruher Institut für Technologie

Algorithmen 2

Kapitel 11: Stringology Teil 3 – Text-Kompression und Wavelet-Trees

Florian Kurpicz

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ `abababbbbaba$`

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba\$

- $f_1 =$ a

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba\$

- $f_1 =$ a
- $f_2 =$ b

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T =$ abababbbaba\$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ aba

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{abab}$
- $f_4 = \texttt{bbb}$
- $f_5 = \texttt{aba}$
- $f_6 = \texttt{\$}$

Lempel-Ziv 77 [ZL77]

Definition: LZ77-Faktorisierung

Sei T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ77-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$ und für alle $i \in [1, z]$
- f_i ist ein einzelnes Zeichen, welches nicht in $f_1 \ldots f_{i-1}$ vorkommt oder
- der längste Substring, der mindestens zweimal in $f_1 \ldots f_i$ vorkommt

$T = \texttt{ababababbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{abab}$
- $f_4 = \texttt{bbb}$
- $f_5 = \texttt{aba}$
- $f_6 = \texttt{\$}$

$T = \underbrace{\texttt{aaa}\ldots\texttt{aa}}_{n-1 \text{ mal}}\texttt{\$}$

- $f_1 = \texttt{a}$
- $f_2 = \underbrace{\texttt{aaa}\ldots\texttt{aa}}_{n-2 \text{ mal}}$
- $f_3 = \texttt{\$}$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T =$ abababbbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ abab
- $f_4 =$ bbb
- $f_5 =$ aba
- $f_6 =$ $

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a} = (0, \texttt{a})$
- $f_2 = \texttt{b} = (0, \texttt{b})$
- $f_3 = \texttt{abab} = (4, 1)$
- $f_4 = \texttt{bbb} = (3, 6)$
- $f_5 = \texttt{aba} = (3, 1) = (3, 3)$
- $f_6 = \texttt{\$} = (0, \texttt{\$})$

Repräsentation der Faktoren

- Faktoren werden als Tupel repräsentiert

$$(\ell_i, p_i)$$

- $\ell_i = 0$
  - Faktor kodiert einzelnes Zeichen
  - Zeichen kann in p_i kodiert werden
- $\ell_i > 0$
  - Faktor kodiert Substring der Länge ℓ_i
  - $f_i = T[p_i..p_i + \ell_i)$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a} = (0, \texttt{a})$
- $f_2 = \texttt{b} = (0, \texttt{b})$
- $f_3 = \texttt{abab} = (4, 1)$
- $f_4 = \texttt{bbb} = (3, 6)$
- $f_5 = \texttt{aba} = (3, 1) = (3, 3)$
- $f_6 = \texttt{\$} = (0, \texttt{\$})$

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	\$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

Mit Bezug auf das *SA*

- in der Nähe im *SA*
- lexikographisch ähnlich
- längstes möglichstes Präfix
- vor dem Suffix in Textreihenfolge

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

Previous- und Next-Smaller-Values (1/2)

Definition: Previous- und Next-Smaller-Value-Arrays

Sei $A[1..n]$ ein Array aus Zahlen, dann ist

- $PSV[i] = \max\{j \in [1, i) \colon A[j] < A[i]\}$
- $NSV[i] = \min\{j \in (i, n] \colon A[j] < A[i]\}$

Mit Bezug auf das *SA*

- in der Nähe im *SA*
- lexikographisch ähnlich
- längstes möglichstes Präfix
- vor dem Suffix in Textreihenfolge

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
PSV	0	0	0	3	3	3	6	3	8	8	8	11	11
NSV	2	3	∞	5	6	8	8	∞	10	11	∞	13	∞
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

-  **PINGO** Welche Laufzeit hat die NSV/PSV-Vorberechnung?

Previous- und Next-Smaller-Values (2/2)

- beide Arrays können in Linearzeit konstruiert werden
- betrachte PSV-Array
  - NSV funktioniert analog
- füge $-\infty$ als Index 0 an

Function ComputePSV(*SA mit* $-\infty$)**:**

```
1  for i = 1, ..., n do
2      j = i − 1
3      while j ≥ 1 und SA[i] < SA[j] do
4          j = PSV[j]
5      PSV[i] = j
6  return PSV
```

Previous- und Next-Smaller-Values (2/2)

- beide Arrays können in Linearzeit konstruiert werden
- betrachte PSV-Array
  NSV funktioniert analog
- füge $-\infty$ als Index 0 an

Function ComputePSV(*SA mit* $-\infty$)**:**

```
1  for i = 1, ..., n do
2      j = i − 1
3      while j ≥ 1 und SA[i] < SA[j] do
4          j = PSV[j]
5      PSV[i] = j
6  return PSV
```

- folge schon berechneten Werten
- dazwischen kann nicht *PSV* sein
- vergleiche jedes Element maximal zweimal
- berechne *PSV* und *NSV* in $O(n)$ Zeit
- Beispiel an der Tafel

RMQs und das LCP-Array

Definition: Range-Minimum-Query

Sei $A[1..m)$ ein Array. Eine **Range-Minimum-Query** für den Bereich $\ell \leq r \in [1, n)$ ergibt

$$RMQ_A(\ell, r) = \arg\min\{A[k] \colon k \in [\ell, r]\}$$

- $lcp(i, j) = \max\{k \colon T[i..i + k) = T[j..j + k)\} = LCP[RMQ_{LCP}(i + 1, j)]$
- RMQs können in $O(1)$ Zeit beantwortet werden und
- benötigen $O(n)$ Wörter Platz
- werden in Advanced Data Structures behandelt

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3

$
a$
abaabcabcabba$
abbba$
abcabba$
abcabcabba$
ba$
babcabcabba$
bba$
bcabba$
bcabcabba$
cabba$
cabcabba$

LZ77-Faktorisierung mit *SA*, *ISA*, *LCP*, *NSV*, *PSV* und *RMQ*s

Function $LZ77(T, SA, ISA, LCP, RMQ, PSV, NSV)$**:**

1 $pos = 1$
2 **while** $pos \leq n$ **do**
3 $psv = SA[PSV[ISA[pos]]]$
4 $nsv = SA[NSV[ISA[pos]]]$
5 **if** $lcp(pos, psv + 1) > lcp(pos + 1, nsv)$ **then**
6 $\ell = lcp(pos, psv + 1)$ und $p = psv$
7 **else**
8 $\ell = lcp(pos + 1, nsv)$ und $p = nsv$
9 **if** $\ell = 0$ **then** $p = T[pos]$
10 **neuer Faktor** (ℓ, p)
11 $pos = pos + \max\{\ell, 1\}$

- Beispiel für Kompression und Dekompression an der Tafel

LZ77: Laufzeit

Lemma: LZ77 Laufzeit

Die LZ77-Faktorisierung von einem Text der Länge n kann in $O(n)$ Zeit berechnet werden

Proof (Sketch)

- SA, LCP, PSV, NSV, RMQ_{LCP} können in $O(n)$ Zeit berechnet werden
- für jedes Zeichen maximal $O(1)$ Zeit

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

$T =$ `abababbbbaba$`

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

$T =$ abababbbbaba$

- $f_1 =$ a

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

$T =$ abababbbbaba\$

- $f_1 =$ a
- $f_2 =$ b

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} : f_k = f_i \alpha$$

$T =$ abababbbbaba\$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} : f_k = f_i \alpha$$

$T =$ abababbbaba\$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab
- $f_4 =$ abb

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

$T =$ abababbbbaba$

- $f_1 =$ a
- $f_2 =$ b
- $f_3 =$ ab
- $f_4 =$ abb
- $f_5 =$ bb

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} \colon f_k = f_i \alpha$$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{ab}$
- $f_4 = \texttt{abb}$
- $f_5 = \texttt{bb}$
- $f_6 = \texttt{aba}$

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} : f_k = f_i \alpha$$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{ab}$
- $f_4 = \texttt{abb}$
- $f_5 = \texttt{bb}$
- $f_6 = \texttt{aba}$
- $f_7 = \texttt{\$}$

Lempel-Ziv 78 [ZL78]

Definition: LZ78-Faktorisierung

Sei text T ein Text der Länge n über dem Alphabet Σ, dann ist die **LZ78-Faktorisierung**

- eine Menge aus z Faktoren $f_1, f_2, \ldots, f_z \in \Sigma^+$, so dass
- $T = f_1 f_2 \ldots f_z$, $f_0 = \epsilon$ und für alle $i \in [1, z]$ gilt, dass
- wenn $f_1 \ldots f_{i-1} = T[1..j-1]$, dann ist f_i das längste Präfix $T[j..n]$, so dass

$$\exists k \in [0, i), \alpha \in \Sigma \cup \{\$\} : f_k = f_i \alpha$$

$T = \texttt{abababbbbaba\$}$

- $f_1 = \texttt{a}$
- $f_2 = \texttt{b}$
- $f_3 = \texttt{ab}$
- $f_4 = \texttt{abb}$
- $f_5 = \texttt{bb}$
- $f_6 = \texttt{aba}$
- $f_7 = \texttt{\$}$

- $T = \texttt{abababbbbaba\$}$

LZ78-Faktorisierung in Linearzeit

Lemma:

Die LZ78-Faktorisierung für einen Text der Länge n kann in $O(n)$ Zeit berechnet werden

LZ78-Faktorisierung in Linearzeit

Lemma:

Die LZ78-Faktorisierung für einen Text der Länge n kann in $O(n)$ Zeit berechnet werden

Proof (Sketch)

- jedes Zeichen wird im Trie maximal einmal gesucht
- jedes Zeichen wird im Trie maximal einmal eingefügt

Burrows-Wheeler-Transformation [BurrowsW1994BWT]

Definition: Burrows-Wheeler-Transformation

Sei T ein Text der Länge n und SA das Suffix-Array von T. Für $i \in [1, n]$ ist die **Burrows-Wheeler-Transformation** definiert als

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$

Burrows-Wheeler-Transformation [BurrowsW1994BWT]

Definition: Burrows-Wheeler-Transformation

Sei T ein Text der Länge n und SA das Suffix-Array von T. Für $i \in [1, n]$ ist die **Burrows-Wheeler-Transformation** definiert als

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	\$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
BWT	a	b	\$	c	c	b	b	a	a	a	a	b	b

Burrows-Wheeler-Transformation [BurrowsW1994BWT]

Definition: Burrows-Wheeler-Transformation

Sei T ein Text der Länge n und SA das Suffix-Array von T. Für $i \in [1, n]$ ist die **Burrows-Wheeler-Transformation** definiert als

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$

- Zeichen vor dem Suffix in SA-Reihenfolge
- wähle Zeichen zyklisch ℹ $ for first suffix

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

Burrows-Wheeler-Transformation [BurrowsW1994BWT]

Definition: Burrows-Wheeler-Transformation

Sei T ein Text der Länge n und SA das Suffix-Array von T. Für $i \in [1, n]$ ist die **Burrows-Wheeler-Transformation** definiert als

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$

- Zeichen vor dem Suffix in SA-Reihenfolge
- wähle Zeichen zyklisch ℹ $ for first suffix

- BWT kann in $O(n)$ Zeit berechnet werden

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

Burrows-Wheeler-Transformation [BurrowsW1994BWT]

Definition: Burrows-Wheeler-Transformation

Sei T ein Text der Länge n und SA das Suffix-Array von T. Für $i \in [1, n]$ ist die **Burrows-Wheeler-Transformation** definiert als

$$BWT[i] = \begin{cases} T[SA[i]-1] & SA[i] > 1 \\ \$ & SA[i] = 1 \end{cases}$$

- Zeichen vor dem Suffix in SA-Reihenfolge
- wähle Zeichen zyklisch ℹ $ for first suffix

- BWT kann in $O(n)$ Zeit berechnet werden

	1	2	3	4	5	6	7	8	9	10	11	12	13
T	a	b	a	b	c	a	b	c	a	b	b	a	$
SA	13	12	1	9	6	3	11	2	10	7	4	8	5
LCP	0	0	1	2	2	5	0	2	1	1	4	0	3
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- aus der BWT kann der Text zurückgewonnen werden 🌶

Kompression der BWT: Run-Length-Kompression

Definition: Run-Length Encoding

Sei T ein Text. Repräsentiere jeden maximalen Run $T[i..i+\ell)$ als Tupel

$$(T[i], \ell)$$

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $(\mathtt{a}, 1)$
- $(\mathtt{b}, 1)$
- $(\mathtt{\$}, 1)$
- $(\mathtt{c}, 2)$
- $(\mathtt{b}, 2)$
- $(\mathtt{a}, 4)$
- $(\mathtt{b}, 2)$

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c
- $MTF = 2\,3\,3\,4$ und $X =$ c, $, b, a

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c
- $MTF = 2\,3\,3\,4$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1$ und $X =$ c, $, b, a

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c
- $MTF = 2\,3\,3\,4$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1\,1$ und $X =$ c, $, b, a

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c
- $MTF = 2\,3\,3\,4$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1\,1$ und $X =$ c, $, b, a
- ...

Kompression der BWT: Move-to-Front

Definition: Move-To-Front Encoding

Sei T ein Text über dem Alphabet $\Sigma = [1, \sigma]$. Das **MTF-Encoding** $MTF(T)$ wird wie folgt berechnet:

- starte mit der Liste $X = \Sigma[1], \Sigma[2], \ldots, \Sigma[\sigma]$
- scanne den Text von links nach rechts und für jedes Zeichen $T[i]$
  - füge Position von $T[i]$ in X an $MTF(T)$ an
  - verschiebe $T[i]$ an den Anfang von X

- MTF-Encoding einfach zu dekodiert
- besteht aus vielen kleinen Zahlen
- Runs bleiben erhalten
- kann mit Huffman weiter komprimiert werden

$T =$ ababcabcabba$

	1	2	3	4	5	6	7	8	9	0	11	12	13
BWT	a	b	$	c	c	b	b	a	a	a	a	b	b

- $X =$ $, a, b, c
- $MTF = 2$ und $X =$ a, $, b, c
- $MTF = 2\,3$ und $X =$ b, a, $, c
- $MTF = 2\,3\,3$ und $X =$ $, b, a, c
- $MTF = 2\,3\,3\,4$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1$ und $X =$ c, $, b, a
- $MTF = 2\,3\,3\,4\,1\,1$ und $X =$ c, $, b, a
- ...
- $MTF = 2\,3\,3\,4\,1\,1\,3\,1\,4\,1\,1\,1\,2\,1$

Rank- und Select-Anfragen

- zum Abschluss noch einen Index
- Anwendung als Textindex und
- für Bereichsanfragen

Definition:

Sei T ein Text der Länge n über einem Alphabet $\Sigma = [1, \sigma]$ und $\alpha \in \Sigma$, dann ist

- $rank_\alpha(i) = |\{j \in [1, i) \colon T[j] = \alpha\}|$
- $select_\alpha(i) = \min\{j \in [2, n+1] \colon rank_\alpha(j) = i\} - 1$

- rank: Wie oft kommt ein Zeichen vor mir vor?
- select: Position des j-ten Vorkommens.

Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

Rank-Anfragen in Bit-Vektoren (1/2)

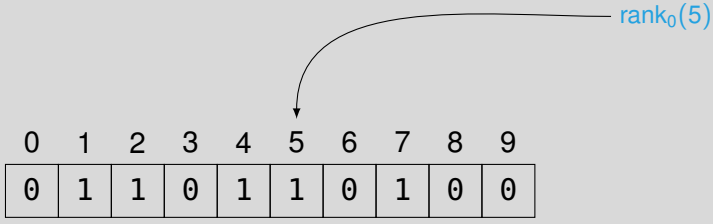
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α

$\text{rank}_0(5)$

0	1	2	3	4	5	6	7	8	9
0	1	1	0	1	1	0	1	0	0

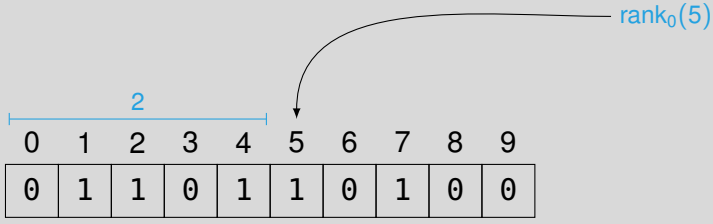
Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



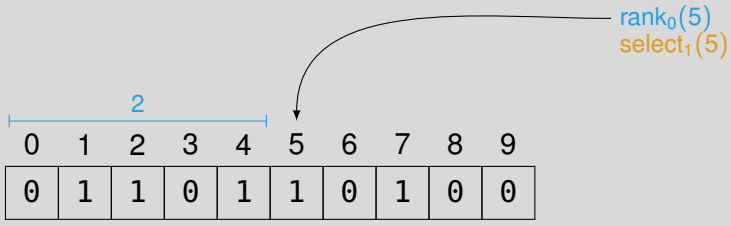
Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

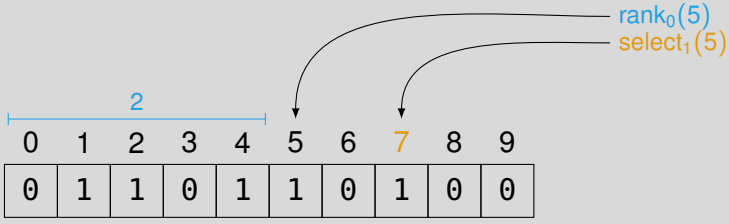
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

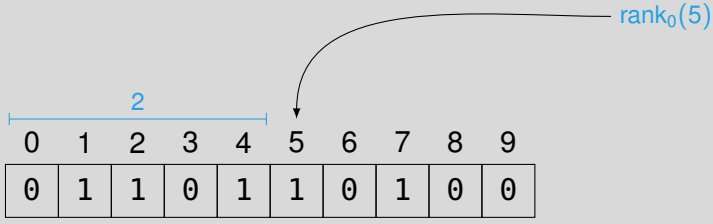
$\text{rank}_\alpha(i)$ # an αs vor Position i

$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

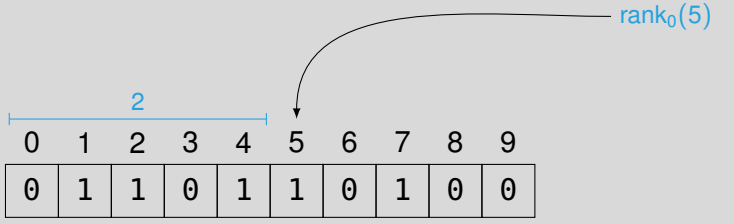
$\text{rank}_\alpha(i)$ # an αs vor Position i
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

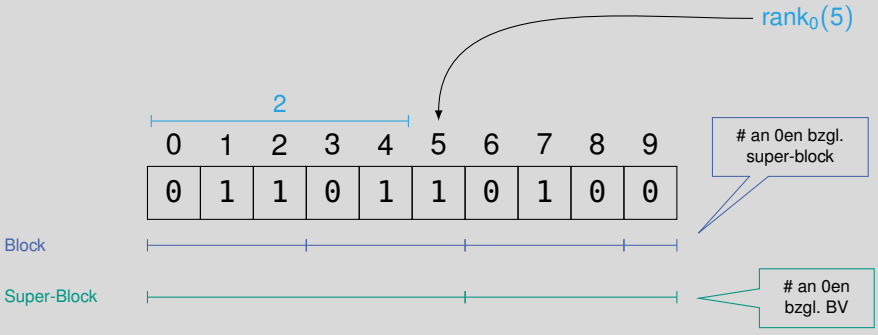
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

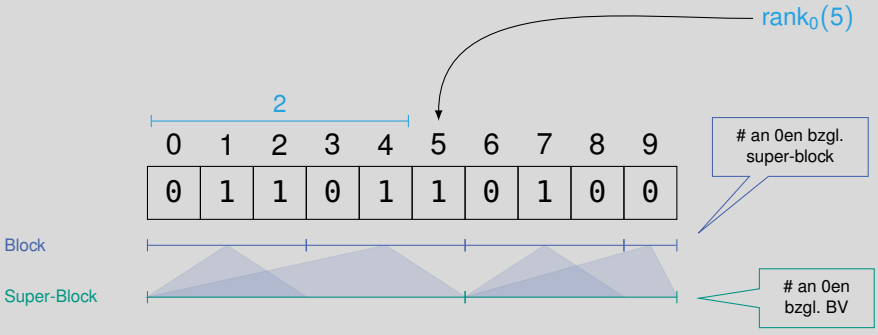
$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (1/2)

$\text{rank}_\alpha(i)$ # an αs vor Position i

$\text{select}_\alpha(j)$ Position des j-ten α



Rank-Anfragen in Bit-Vektoren (2/2)

-  **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

- für alle Bitvektoren der Länge s, speichere für jede Position i die Anzahl der 0en bis zu der Position
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ Bits Platz

Rank-Anfragen in Bit-Vektoren (2/2)

- **PINGO** Wie viel Platz benötigt eine Rank-Datenstruktur mit $O(1)$ Anfragezeit mindestens?

- für einen Bitvektor der Länge n betrachte
- Blöcke der Länge $s = \lfloor \frac{\lg n}{2} \rfloor$
- Super-Blöcke der Länge $s' = s^2 = \Theta(\lg^2 n)$

- für alle $\lfloor \frac{n}{s'} \rfloor$ Super-Blöcke, speichere Anzahl an 0en von Anfang des Bitvektors bis zum Ende des Super-Blocks
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ Bits Platz

- für alle $\lfloor \frac{n}{s} \rfloor$ Blöcke, speichere Anzahl an 0en von Anfang des Super-Blocks bis zum Ende des Blocks
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ Bits Platz

- für alle Bitvektoren der Länge s, speichere für jede Position i die Anzahl der 0en bis zu der Position
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ Bits Platz

- $rank_0(i) = i - rank_1(i)$

Select-Anfragen in Bitvektoren

- Select-Anfragen auch in $O(1)$ Zeit und
- $o(n)$ Bits Platz

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

MSB	0	0	1	1	0	1	1	0	1	0
	0	0	1	1	0	0	0	1	1	1
LSB	0	1	0	1	1	1	0	0	0	1

Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$

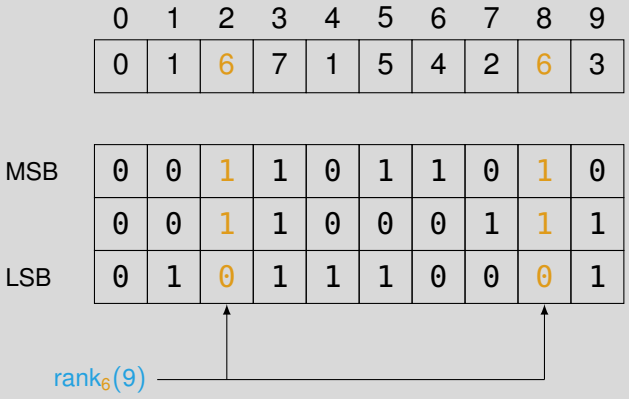
0	1	2	3	4	5	6	7	8	9
0	1	6	7	1	5	4	2	6	3

MSB	0	0	1	1	0	1	1	0	1	0
	0	0	1	1	0	0	0	1	1	1
LSB	0	1	0	1	1	1	0	0	0	1

$\text{rank}_6(9)$

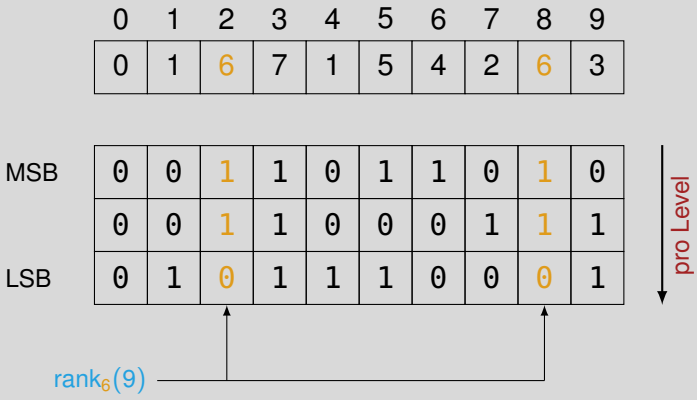
Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Wie für Größere Alphabete?

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$



Wavelet-Trees [GGV03] (1/2)

Definition: Wavelet-Tree

Für einen Text T der Länge n über dem Alphabet $\Sigma = [1, \sigma]$ ist ein **Wavelet-Tree** ein Binärbaum, bei dem

- jeder Knoten Zeichen in $[\ell, r] \subseteq [1, \sigma]$ repräsentiert
- das linke und rechte Kind eines Knotens der Zeichen in $[\ell, r]$ repräsentiert
- repräsentieren Zeichen in $[\ell, (\ell + r)/2)$ und $[(\ell + r)/2, r]$
- ein Knoten ist ein Blatt, wenn $\ell + 2 \geq r$
- Zeichen werden durch ein Bit in einem Bitvektor repräsentieren
- Eintrag ist 1, wenn Zeichen im rechten Kind repräsentiert wird, sonst 0

Wavelet-Trees (2/2)

[0, 7]

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

→

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

Wavelet-Trees (2/2)

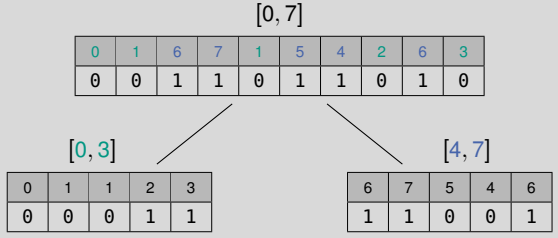
[0, 7]

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

→

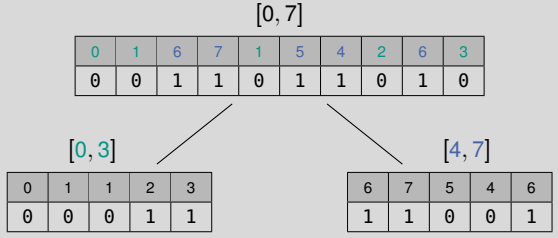
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)



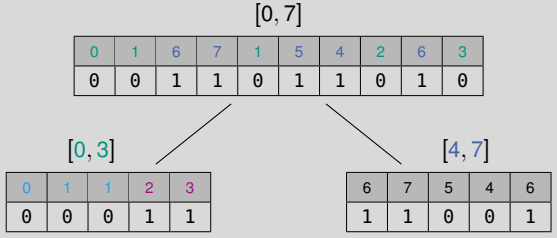
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)



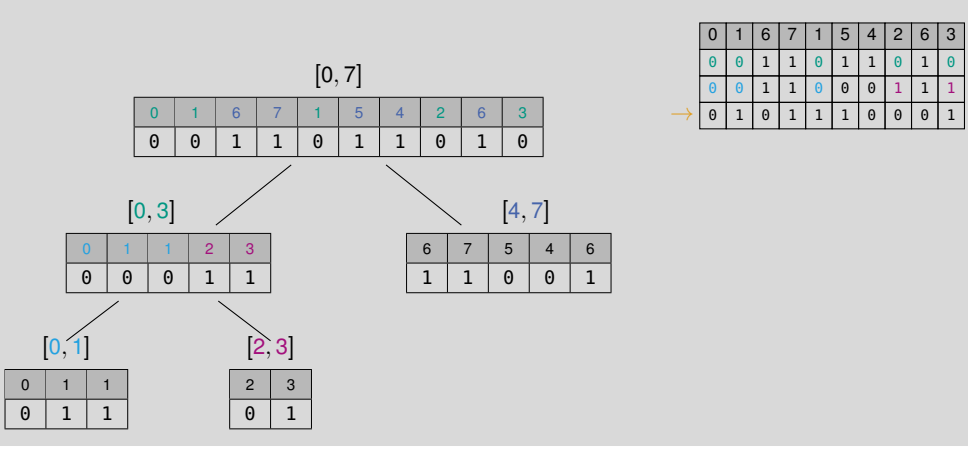
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees (2/2)

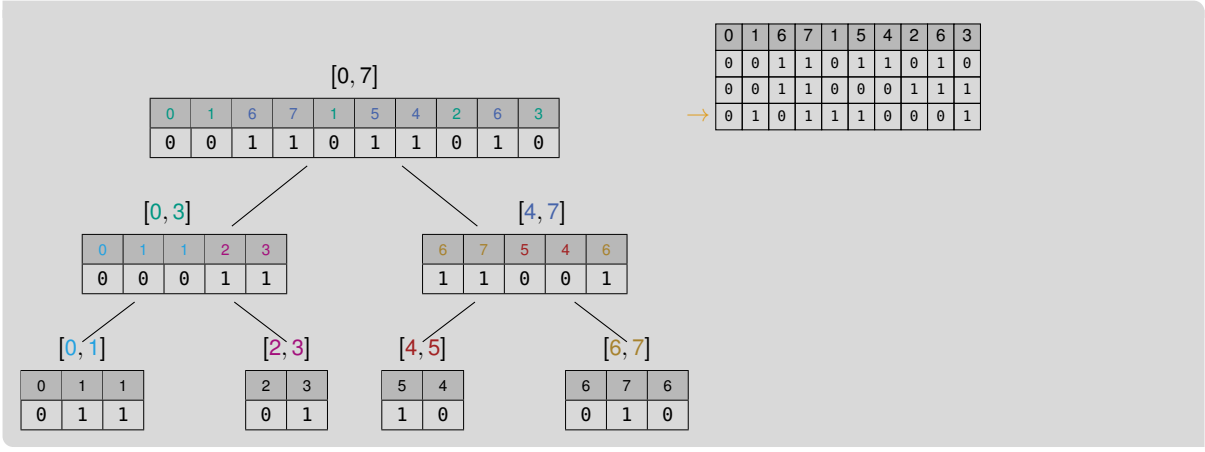


0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

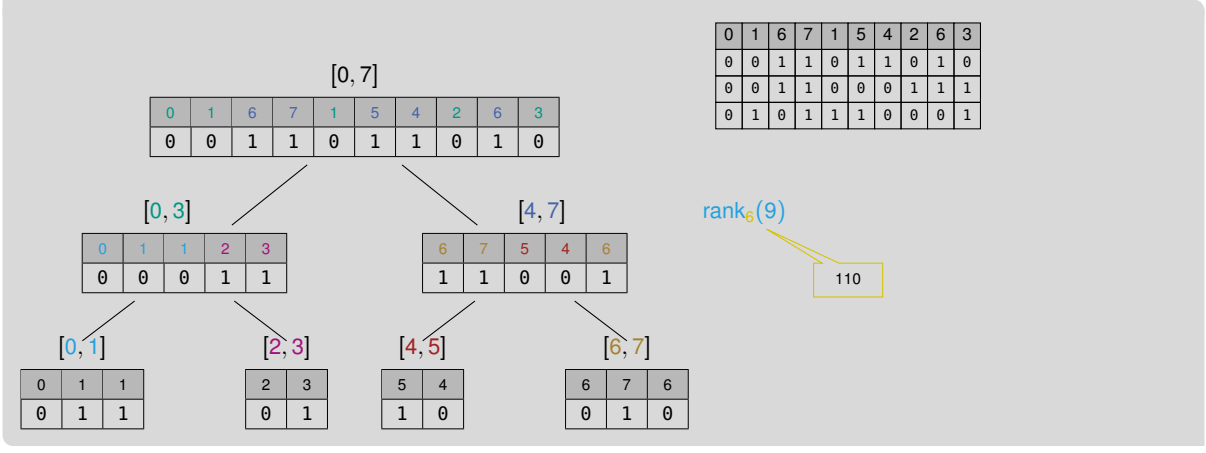
Wavelet-Trees (2/2)



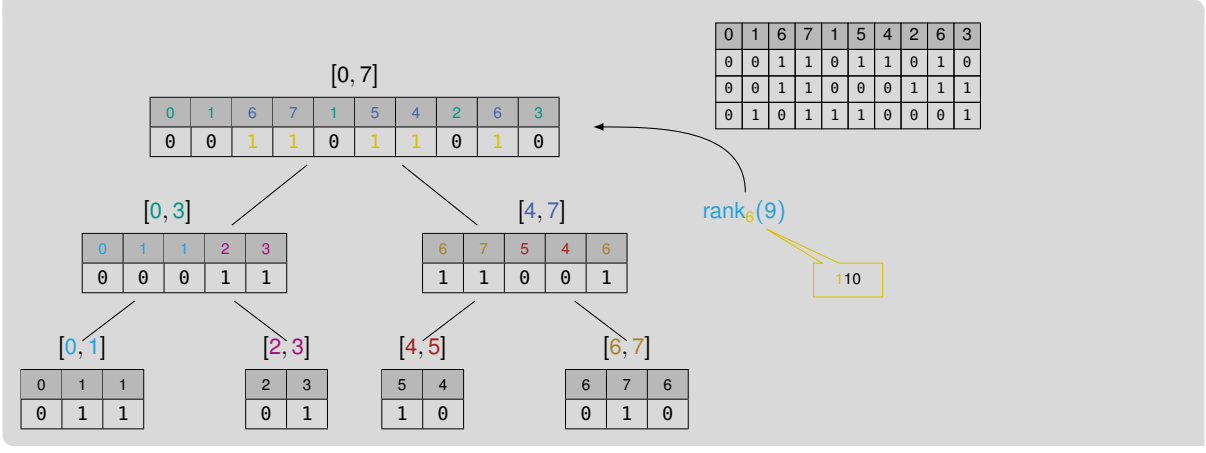
Wavelet-Trees (2/2)



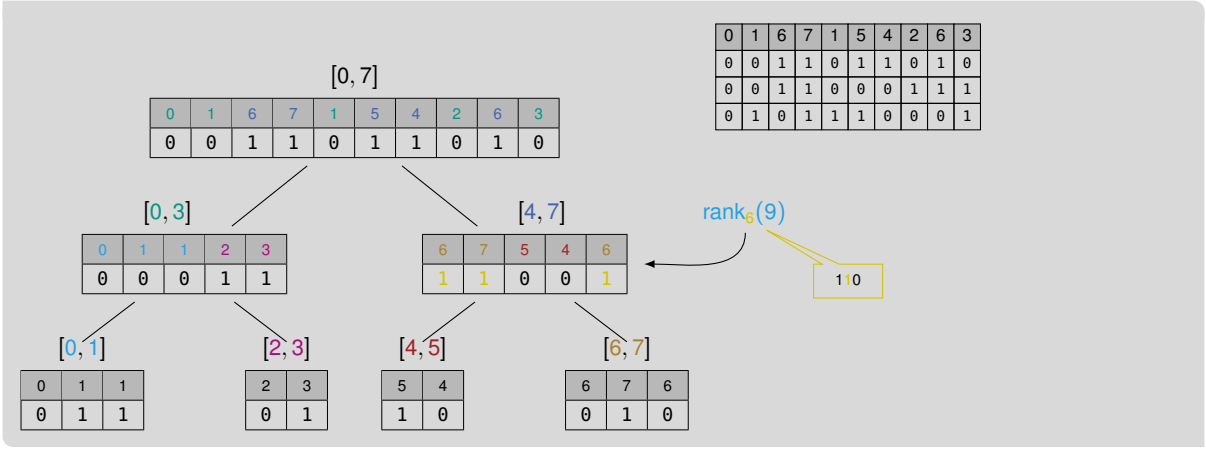
Wavelet-Trees (2/2)



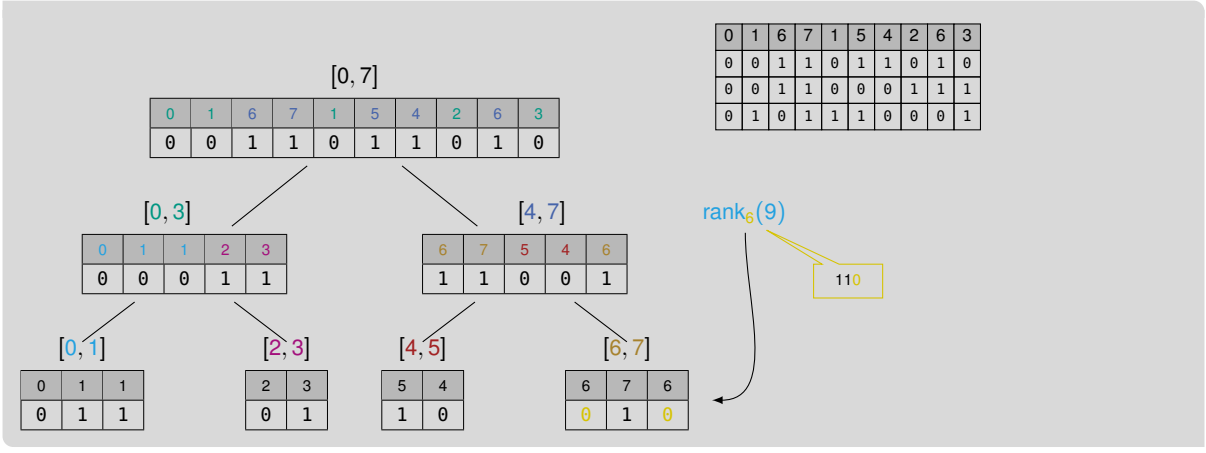
Wavelet-Trees (2/2)



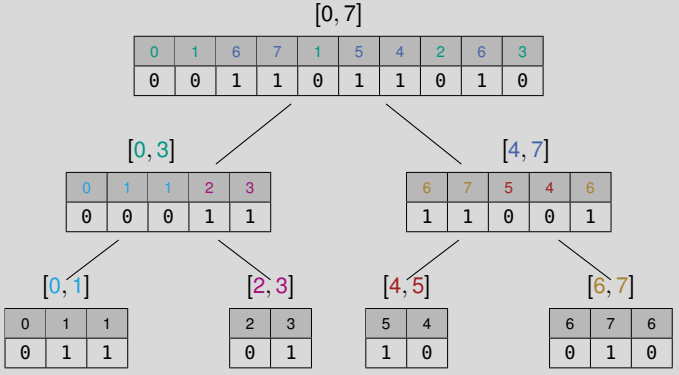
Wavelet-Trees (2/2)



Wavelet-Trees (2/2)

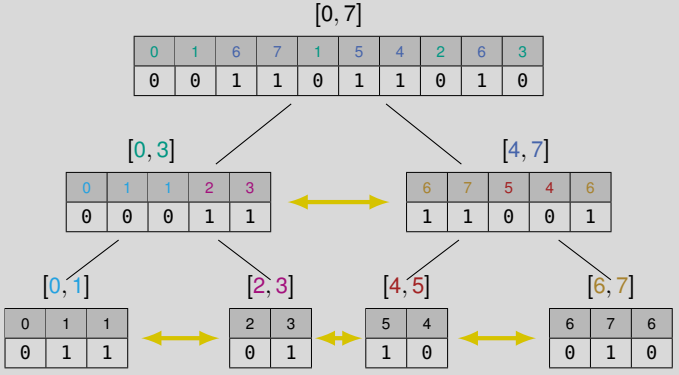


Wavelet-Trees (2/2)



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

Wavelet-Trees (2/2)



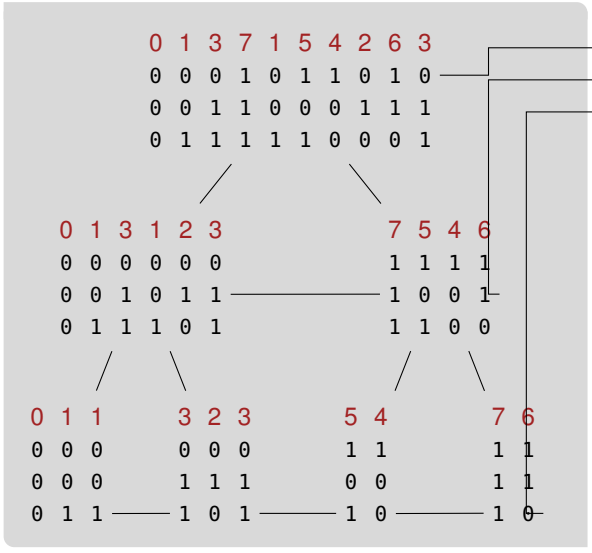
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet-Trees: Konstruktion und Anfragen

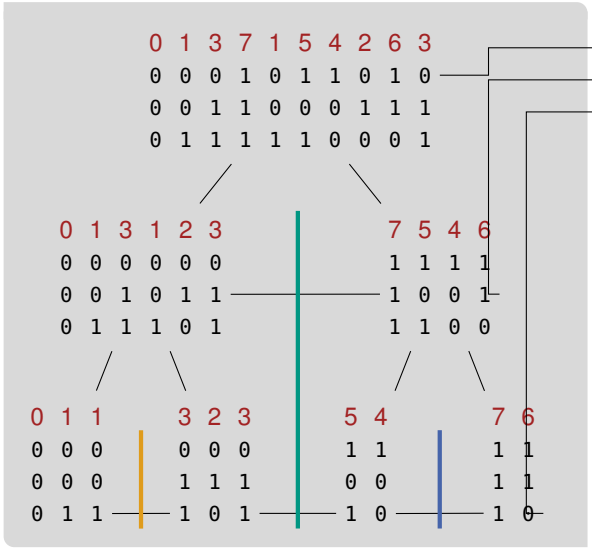
Der Wavelet-Tree

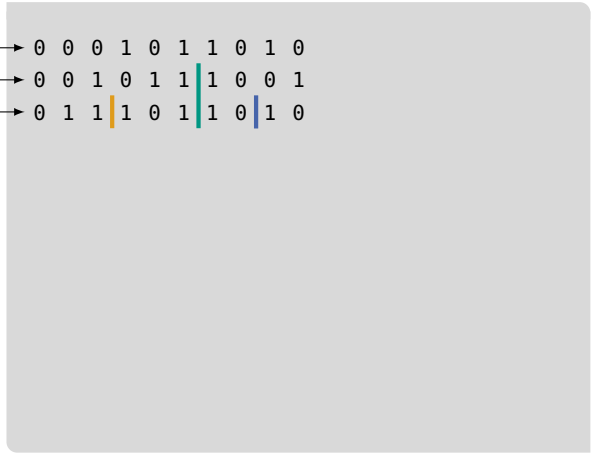
- kann naiv in $O(n \lg \sigma)$ Zeit konstruiert werden
- kann in $O(n \lg \sigma / \sqrt{\lg n})$ Zeit konstruiert werden [Bab+15; MNV16]
- benötigt $n\lceil \lg \sigma \rceil (1 + o(1))$ Bits Platz

- Rank- und Select-Anfragen können in $O(\lg \sigma)$ Zeit beantwortet werden
- Zugriff auf Zeichen in $O(\lg \sigma)$ Zeit möglich

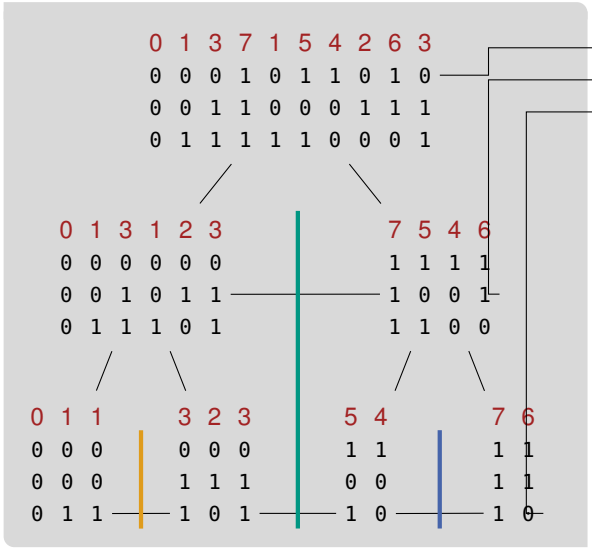


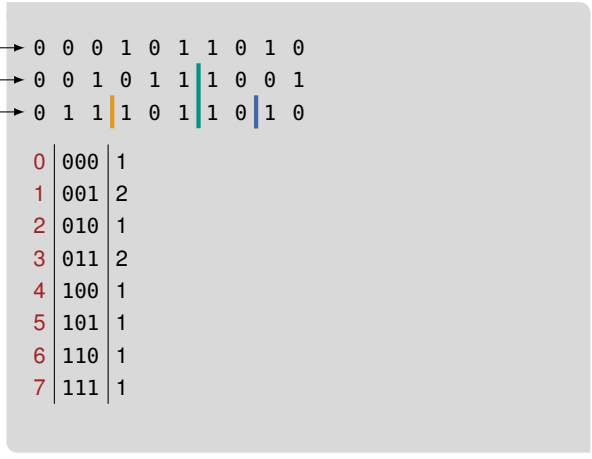
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]



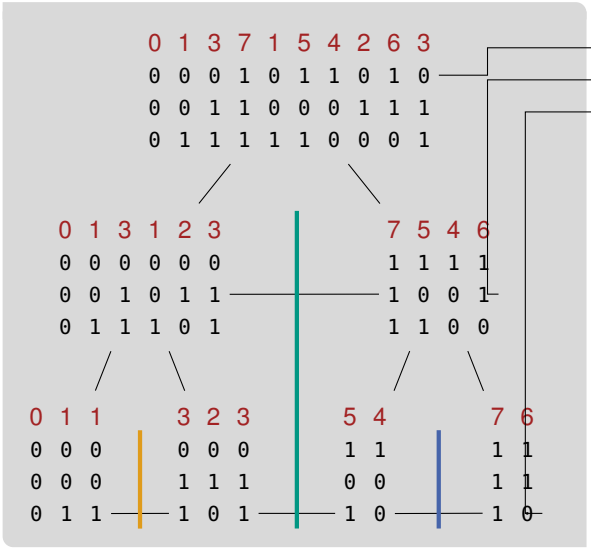


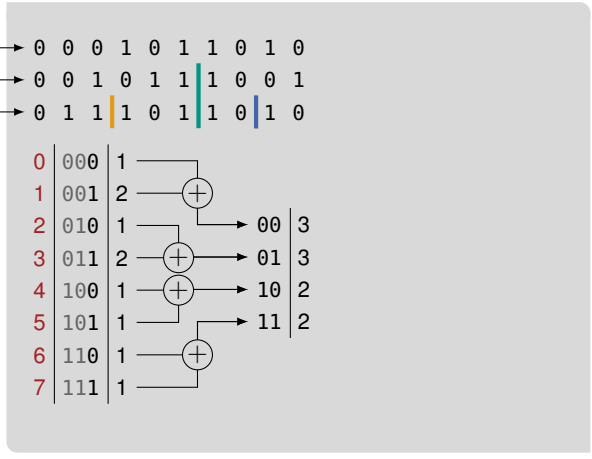
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]



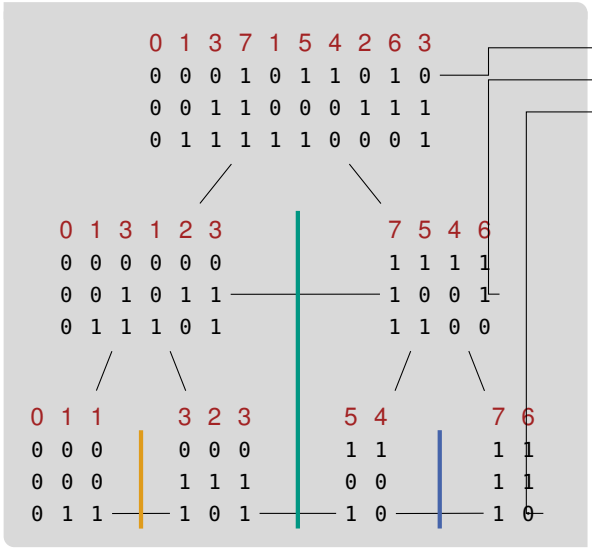


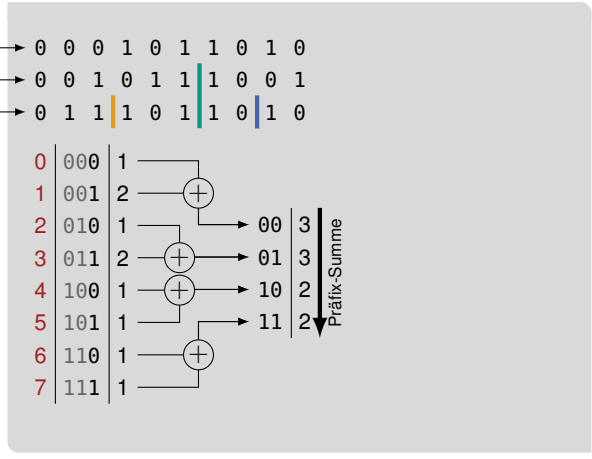
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]



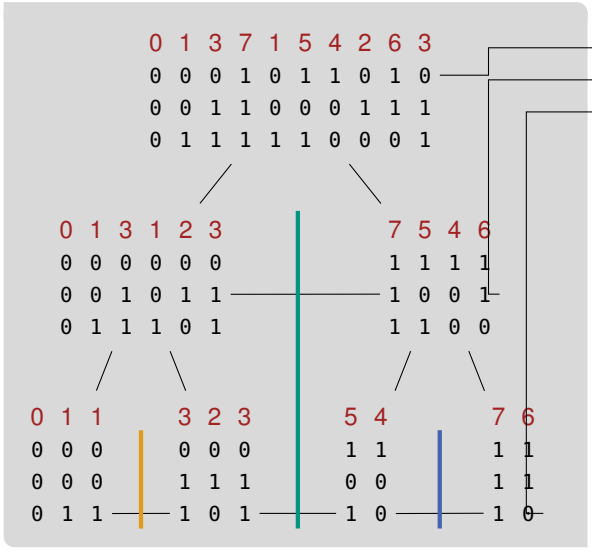


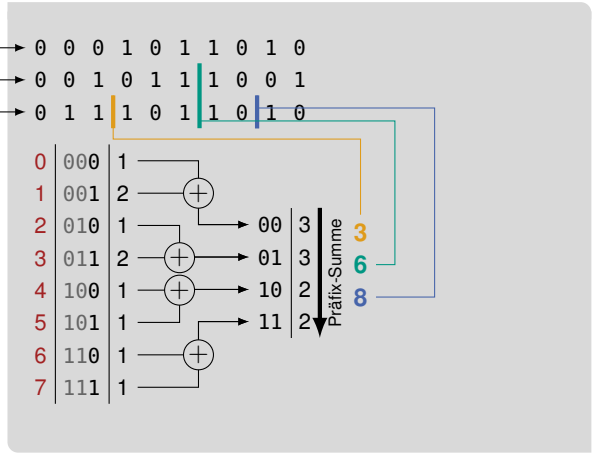
- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]





- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]





- Wavelet-Trees Bottom-Up-Konstruktion [FKL18]

Literatur I

[Bab+15] Maxim A. Babenko, Pawel Gawrychowski, Tomasz Kociumaka und Tatiana Starikovskaya. „Wavelet Trees Meet Suffix Trees“. In: *SODA*. SIAM, 2015, Seiten 572–591. DOI: 10.1137/1.9781611973730.39.

[FKL18] Johannes Fischer, Florian Kurpicz und Marvin Löbel. „Simple, Fast and Lightweight Parallel Wavelet Tree Construction“. In: *ALENEX*. SIAM, 2018, Seiten 9–20. DOI: 10.1137/1.9781611975055.2.

[GGV03] Roberto Grossi, Ankur Gupta und Jeffrey Scott Vitter. „High-Order Entropy-Compressed Text Indexes“. In: *SODA*. ACM/SIAM, 2003, Seiten 841–850.

[MNV16] J. Ian Munro, Yakov Nekrich und Jeffrey Scott Vitter. „Fast construction of wavelet trees“. In: *Theor. Comput. Sci.* 638 (2016), Seiten 91–97. DOI: 10.1016/j.tcs.2015.11.011.

[ZL77] Jacob Ziv und Abraham Lempel. „A Universal Algorithm for Sequential Data Compression“. In: *IEEE Trans. Inf. Theory* 23.3 (1977), Seiten 337–343. DOI: 10.1109/TIT.1977.1055714.

[ZL78] Jacob Ziv und Abraham Lempel. „Compression of Individual Sequences via Variable-Rate Coding“. In: *IEEE Trans. Inf. Theory* 24.5 (1978), Seiten 530–536. DOI: 10.1109/TIT.1978.1055934.