Translating Between
Wavelet Tree and Wavelet Matrix Construction

Patrick Dinklage

Efficient Rank and Select Queries
on binary alphabets

$$B = 10100011000$$

Efficient Rank and Select Queries
on binary alphabets

$$B = \texttt{10100011000}$$

$rank_1(B, i)$:   number of 1s from beginning to i

Efficient Rank and Select Queries
on binary alphabets

$$B = \texttt{10100011000}$$

$rank_1(B, i)$:   number of 1s from beginning to i

$select_1(B, k)$: position of the k-th 1

Efficient Rank and Select Queries
on binary alphabets

$$B = \texttt{10100011000}$$

$rank_1(B, i)$:   number of 1s from beginning to i

$select_1(B, k)$: position of the k-th 1

➢ constant time with $O(n)$ preprocessing time and $o(n)$ bits of space

Efficient Rank and Select Queries
on binary alphabets

$$B = \texttt{10100011000}$$

$rank_1(B, \textbf{\textit{i}})$:   number of 1s from beginning to $\textbf{\textit{i}}$

$select_1(B, \textbf{\textit{k}})$: position of the $\textbf{\textit{k}}$-th 1

$n := |B|$

➢ constant time with $O(n)$ preprocessing time and $o(n)$ bits of space

Efficient Rank and Select Queries
on binary alphabets

$$B = \texttt{10100011000}$$

$rank_1(B, i)$:   number of 1s from beginning to i

$select_1(B, k)$: position of the k-th 1

$n := |B|$

➤ constant time with $O(n)$ preprocessing time and $o(n)$ bits of space

(rank$_0$ and select$_0$ analogously)

Efficient Rank and Select Queries
on integer alphabets

Efficient Rank and Select Queries
on integer alphabets

$$T = \texttt{wavelettree}$$

Efficient Rank and Select Queries
on integer alphabets

$$T = \texttt{wavelettree}$$

➤ $\Sigma = \{\texttt{a,e,l,r,t,v,w}\}$

Efficient Rank and Select Queries
on integer alphabets

$$T = \texttt{wavelettree}$$

> ➢ $\Sigma = \{\texttt{a,e,l,r,t,v,w}\}$

$rank_c(T, i)$:   number of occurrences of c from beginning to i

$select_c(T, k)$: position of the k-th occurrence of c

Efficient Rank and Select Queries
on integer alphabets

$$T = \texttt{wavelettree}$$

➢ $\Sigma = \{\texttt{a,e,l,r,t,v,w}\}$

$rank_c(T, \boldsymbol{i})$:   number of occurrences of $\boldsymbol{c}$ from beginning to $\boldsymbol{i}$

$select_c(T, \boldsymbol{k})$: position of the $\boldsymbol{k}$-th occurrence of $\boldsymbol{c}$

➢ time $O(lg\ \sigma)$ with $O(n\ lg\ \sigma)$ preprocessing time

using a **wavelet tree** or **wavelet matrix**

Efficient Rank and Select Queries
on integer alphabets

$$T = \texttt{wavelettree}$$

➤ $\Sigma = \{\texttt{a,e,l,r,t,v,w}\}$

$rank_c(T, i)$:   number of occurrences of c from beginning to i

$select_c(T, k)$: position of the k-th occurrence of c

$$\sigma := |\Sigma|$$

➤ time $O(lg\ \sigma)$ with $O(n\ lg\ \sigma)$ preprocessing time

using a **wavelet tree** or **wavelet matrix**

The Wavelet Tree

$$\Sigma = \{a,e,l,r,t,v,w\}$$

$$T = \boxed{\texttt{wavelettree}}$$

The Wavelet Tree

$\Sigma = \{a,e,l,r,t,v,w\}$

$T = $ `wavelettree`

The Wavelet Tree

$\Sigma = \{a,e,l,r,t,v,w\}$

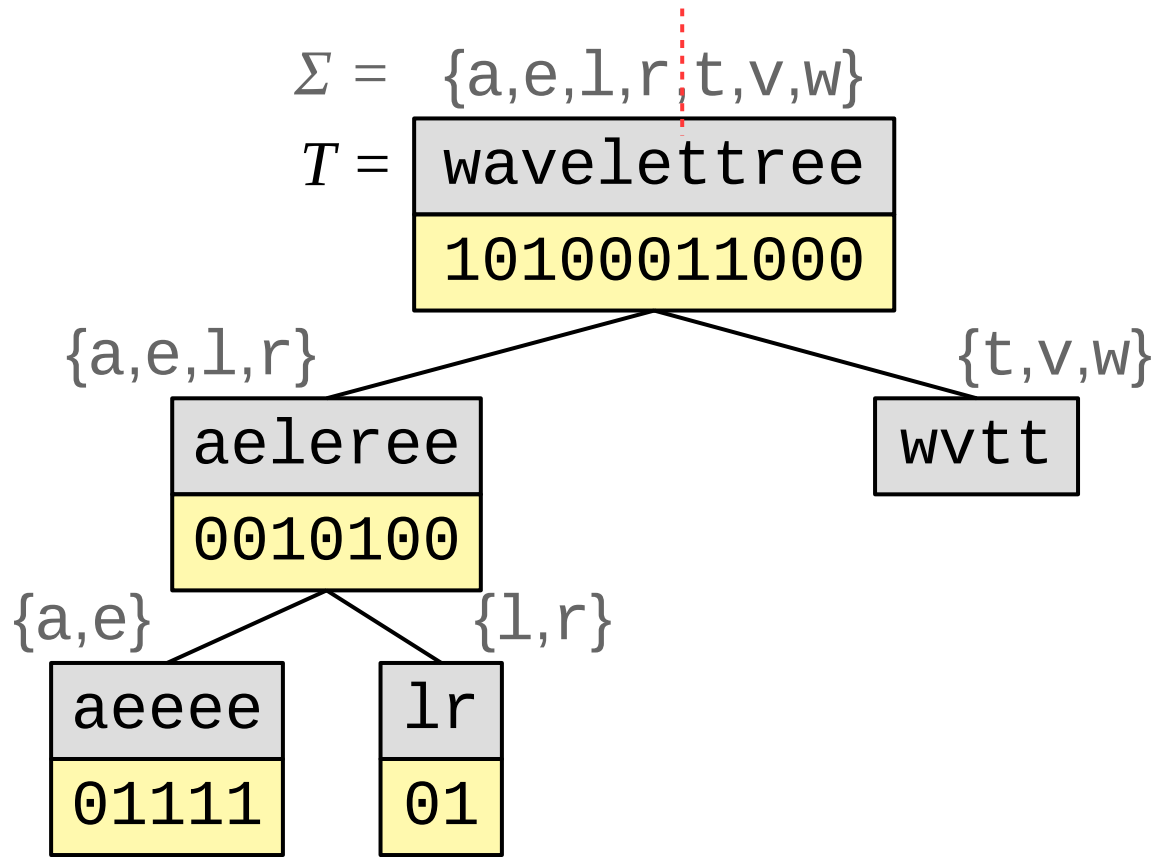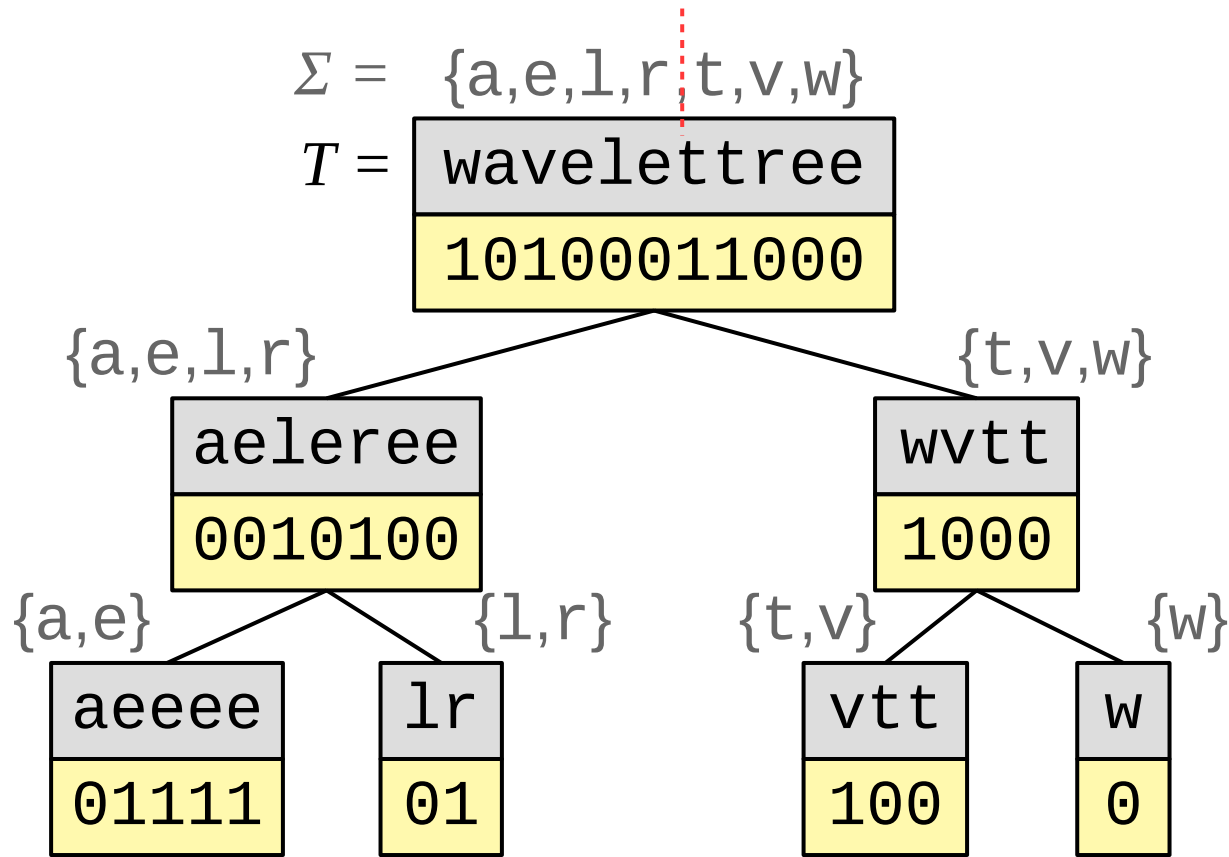$T = $ `wavelettree`

`10100011000`

The Wavelet Tree

Σ = {a,e,l,r,t,v,w}

T = `wavelettree`

`10100011000`

{a,e,l,r} `aeleree`

{t,v,w} `wvtt`

The Wavelet Tree

$\Sigma =$ {a,e,l,r,t,v,w}

$T =$ `wavelettree`
`10100011000`

{a,e,l,r}
`aeleree`
`0010100`

{t,v,w}
`wvtt`

{a,e}
`aeeee`
`01111`

{l,r}
`lr`
`01`

The Wavelet Tree

$\Sigma =$ {a,e,l,r,t,v,w}

$T =$ | wavelettree |
10100011000

{a,e,l,r}

| aeleree |
0010100

{t,v,w}

| wvtt |
1000

{a,e}

| aeeee |
01111

{l,r}

| lr |
01

{t,v}

| vtt |
100

{w}

| w |
0

The Wavelet Tree

$\Sigma =$ {a,e,l,r,t,v,w}

$T =$ | wavelettree |
| 10100011000 |

{a,e,l,r}

| aeleree |
| 0010100 |

{t,v,w}

| wvtt |
| 1000 |

height $\lceil lg\ \sigma \rceil$

{a,e}

| aeeee |
| 01111 |

{l,r}

| lr |
| 01 |

{t,v}

| vtt |
| 100 |

{w}

| w |
| 0 |
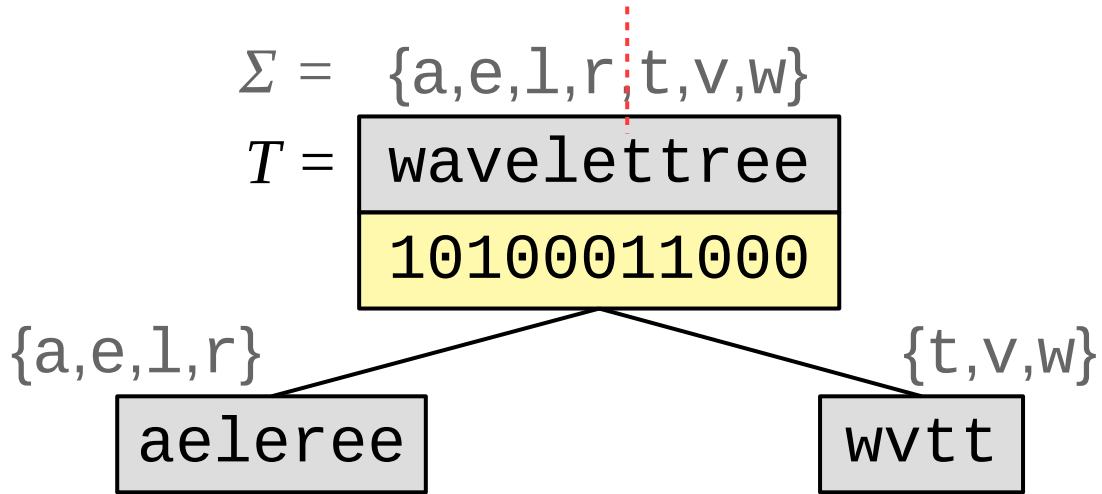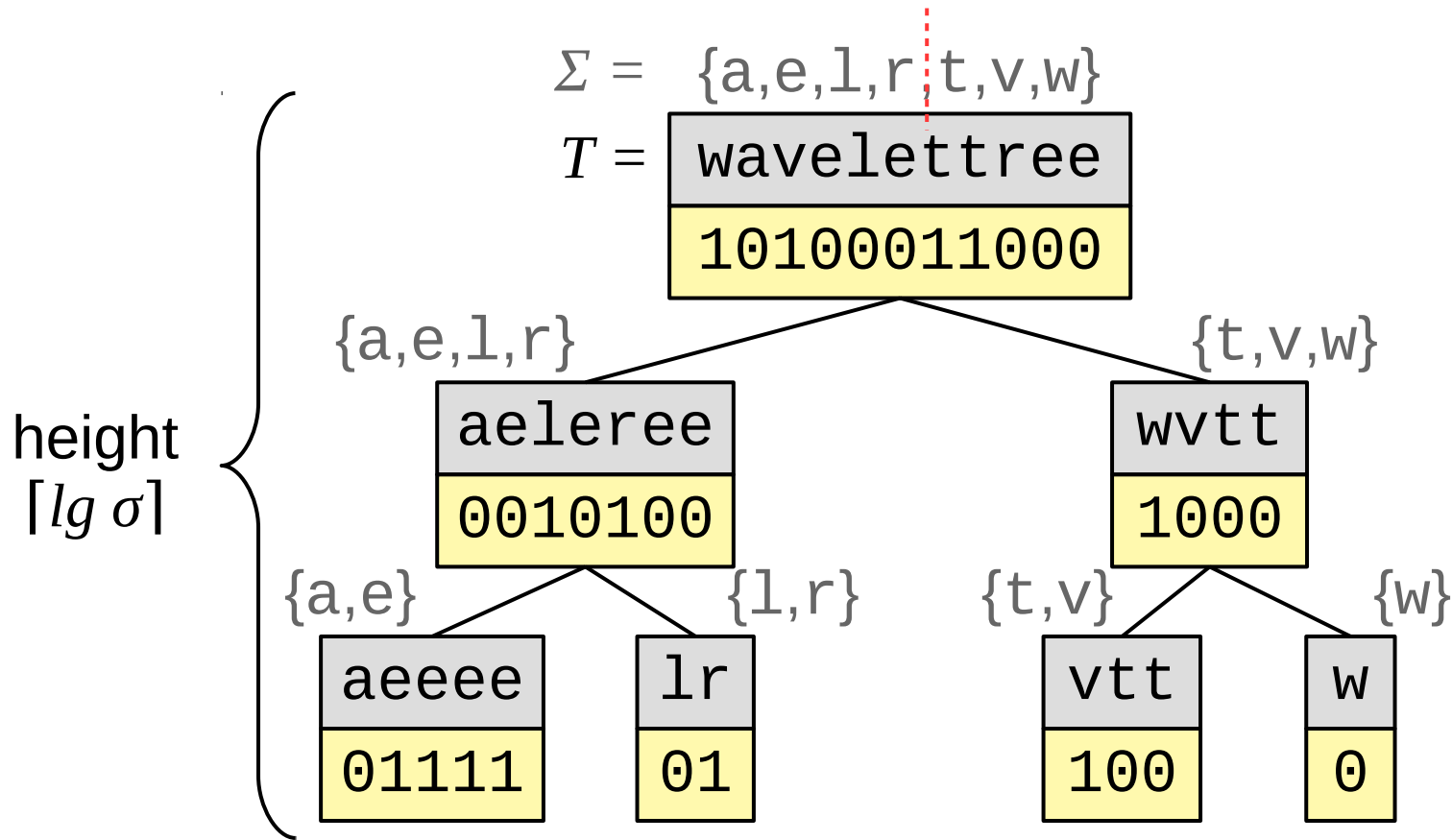
The Wavelet Tree
Storage
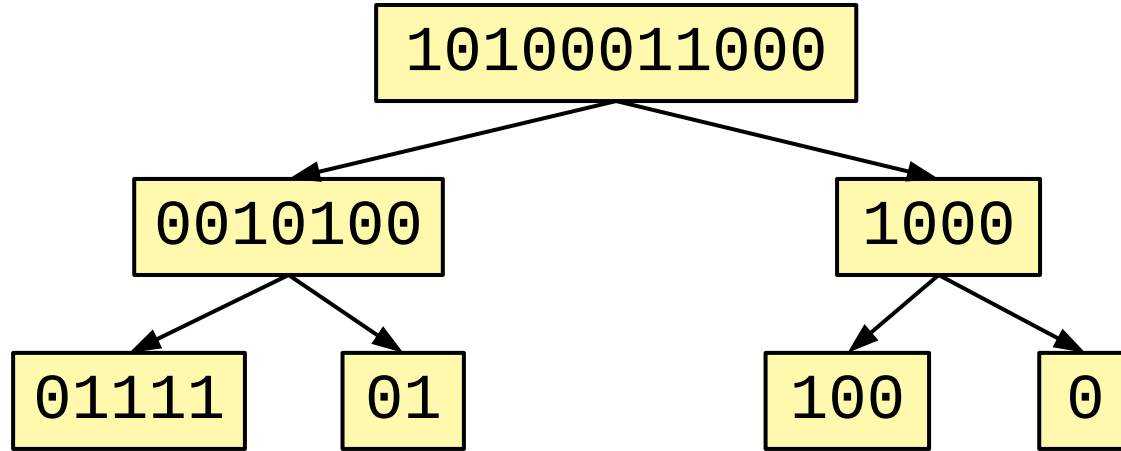
The Wavelet Tree
Storage
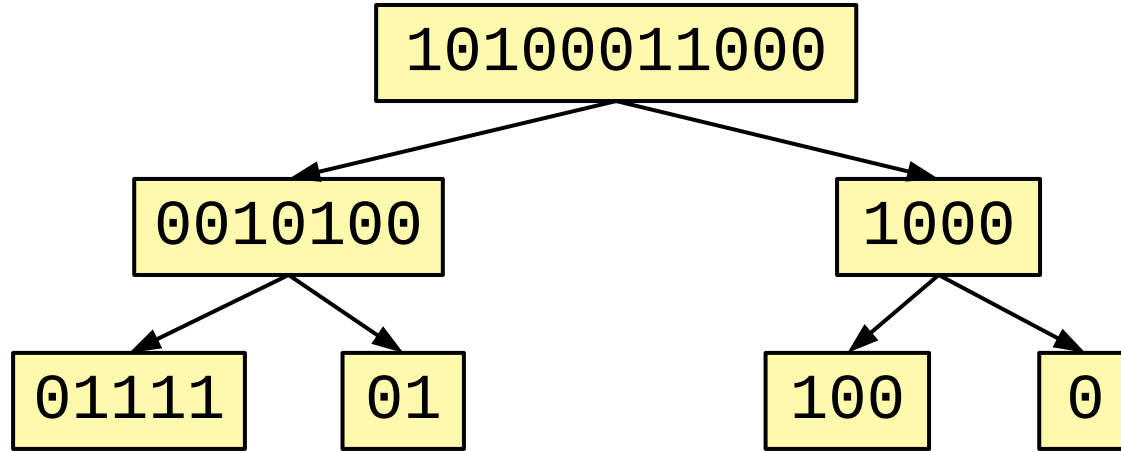
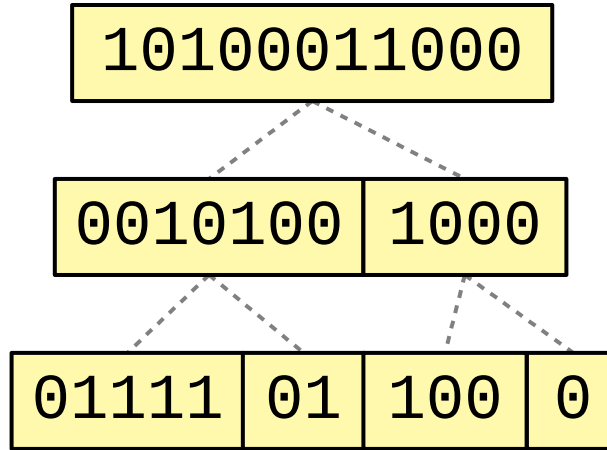➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select

The Wavelet Tree
Storage

➢ *(n+o(n))* ⌈*lg σ*⌉ bits for bit vectors + rank/select

➢ up to *(2σ – 2)* ⌈*lg n*⌉ bits for pointers

The Wavelet Tree
Storage

10100011000

0010100	1000

01111	01	100	0

> ➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select
>
> ➢ up to *(2σ – 2)* $\lceil lg\ n \rceil$ bits for node boundaries
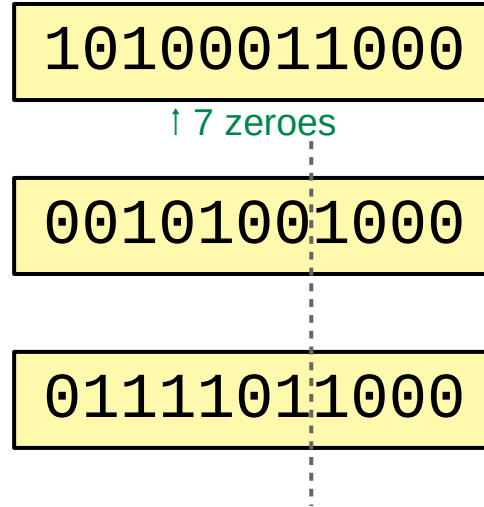
The Wavelet Tree
Storage

10100011000

0010100	1000

01111	01	100	0

> ➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select
>
> ➢ up to *(2σ − 2)* $\lceil lg\ n \rceil$ bits for node boundaries

The Wavelet Tree
Storage

10100011000

00101001000

01111011000

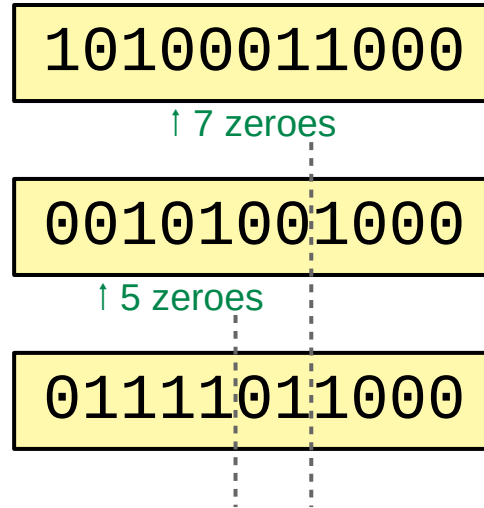➢ *(n+o(n))* ⌈*lg σ*⌉ bits for bit vectors + rank/select

The Wavelet Tree
Storage

$$10100011000$$

↑ 7 zeroes

$$00101001000$$

$$01111011000$$

➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select

The Wavelet Tree
Storage

$$10100011000$$

↑ 7 zeroes

$$00101001000$$

↑ 5 zeroes

$$01111011000$$

➢ *(n+o(n))* ⌈*lg σ*⌉ bits for bit vectors + rank/select

The Wavelet Tree
Storage

```
10100011000
```
↑ 7 zeroes

```
00101001000
```
↑ 5 zeroes   ↑ 3 zeroes

```
01111011000
```

➢ *(n+o(n))* ⌈*lg σ*⌉ bits for bit vectors + rank/select

The Wavelet Tree
Storage

```
10100011000
```
↑ 7 zeroes

```
00101001000
```
↑ 5 zeroes   ↑ 3 zeroes

```
01111011000
```

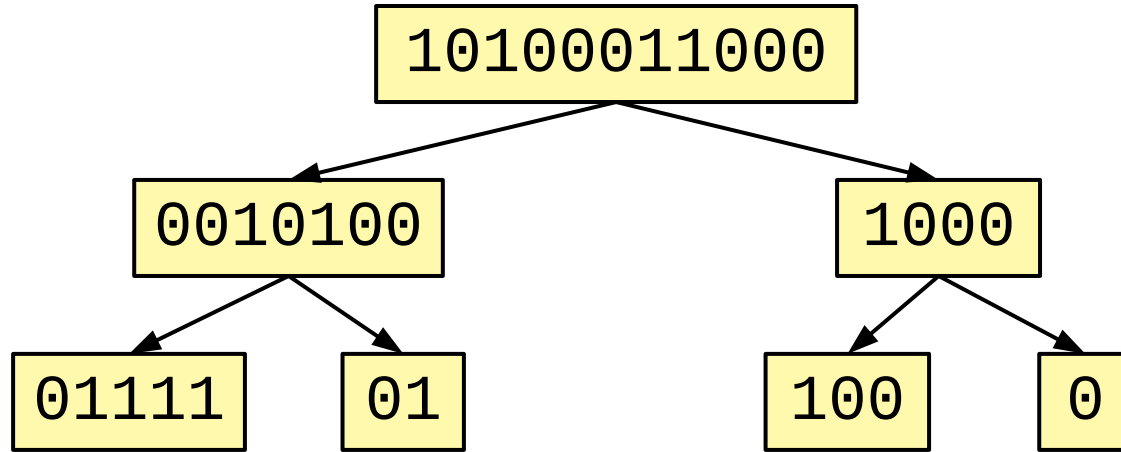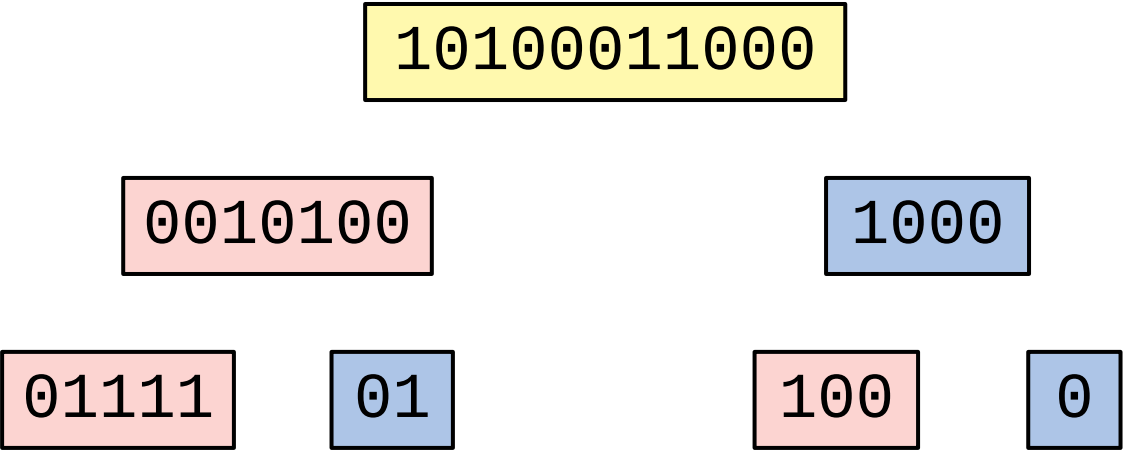➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select

➢ implicit node boundaries!

The Wavelet Matrix

The Wavelet Matrix

10100011000

0010100

1000

01111

01

100

0

The Wavelet Matrix

```
10100011000
```

```
0010100
```
```
1000
```

```
01111
```
```
01
```
```
100
```
```
0
```
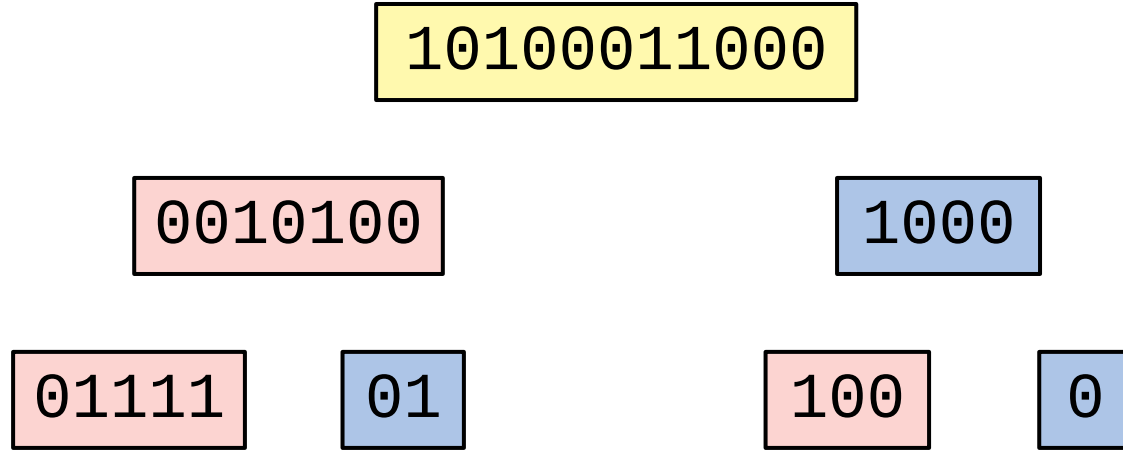
group left children at the left, right children at the right

The Wavelet Matrix

10100011000

0010100 1000

01111 100 010

The Wavelet Matrix
Storage

```
10100011000
```

```
0010100 1000
```

```
01111100 010
```

The Wavelet Matrix
Storage

```
10100011000
```
→ 7 zeroes

```
0010100 1000
```
→ 8 zeroes

```
01111100 010
```

The Wavelet Matrix
Storage

```
10100011000
```
→ 7 zeroes

```
0010100 1000
```
→ 8 zeroes

```
01111100 010
```

➢ *(n+o(n))* ⌈*lg σ*⌉ bits for bit vectors + rank/select

The Wavelet Matrix
Storage

$$10100011000 \rightarrow \text{7 zeroes}$$

$$0010100 \mid 1000 \rightarrow \text{8 zeroes}$$
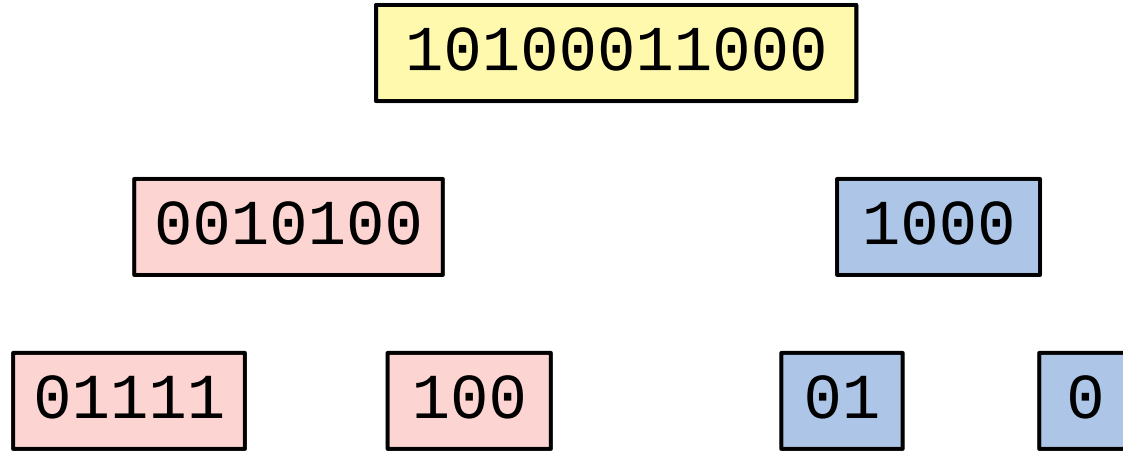
$$01111100 \mid 010$$

➢ *(n+o(n))* $\lceil lg\ \sigma \rceil$ bits for bit vectors + rank/select

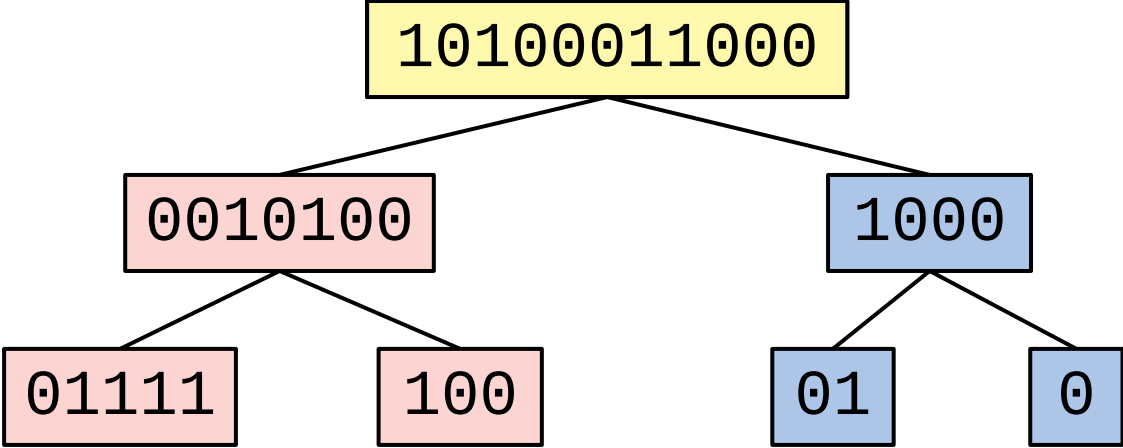➢ affordable $\lceil lg\ n \rceil \lceil lg\ \sigma \rceil$ bits for zeroes (boundaries)
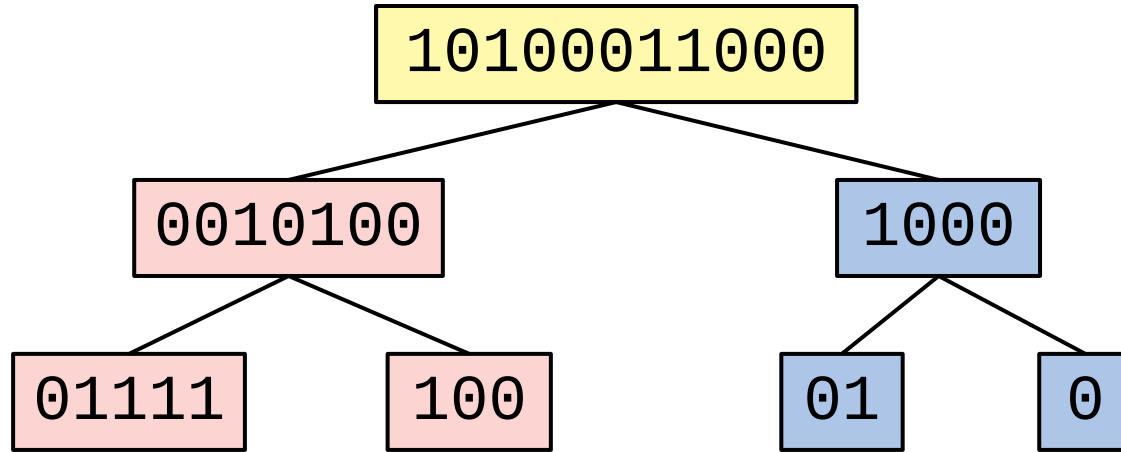
The Wavelet Matrix
… as a tree

10100011000

0010100

1000

01111

100

01

0

The Wavelet Matrix
… as a tree

The Wavelet Matrix
… as a tree


```
              10100011000

     0010100                  1000

  01111      100         01        0
```
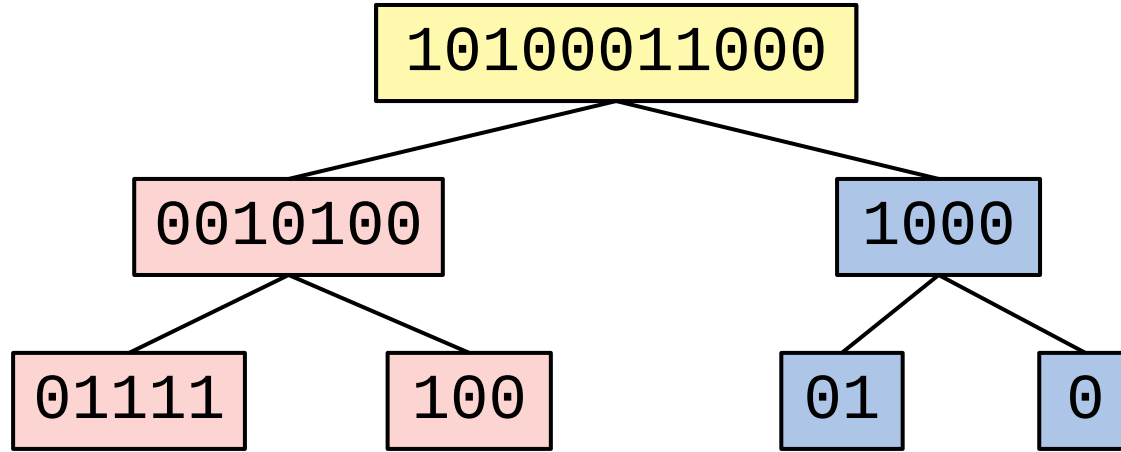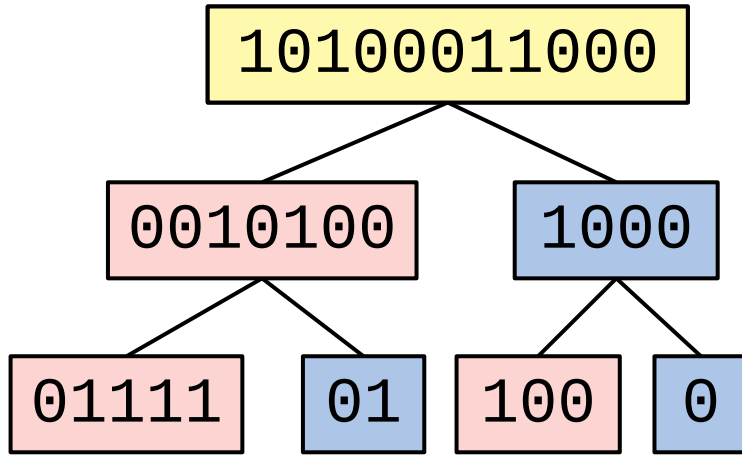
➤ will be useful for our cause

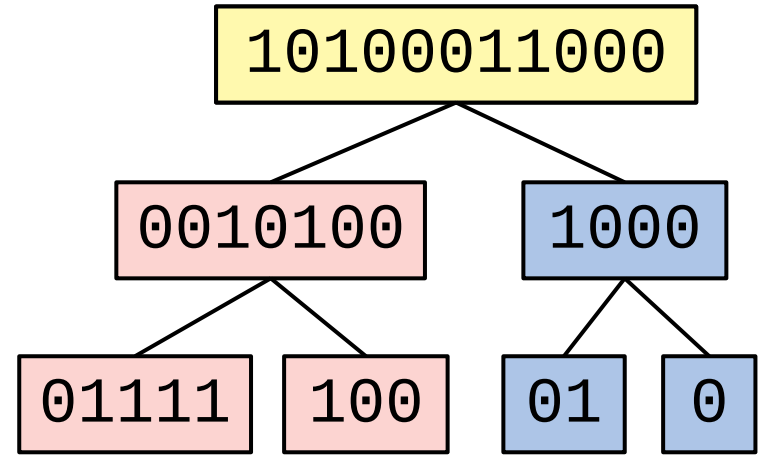# The Wavelet Matrix
… as a tree

➢ will be useful for our cause

(but is not of practical relevance)

Wavelet Tree vs Wavelet Matrix
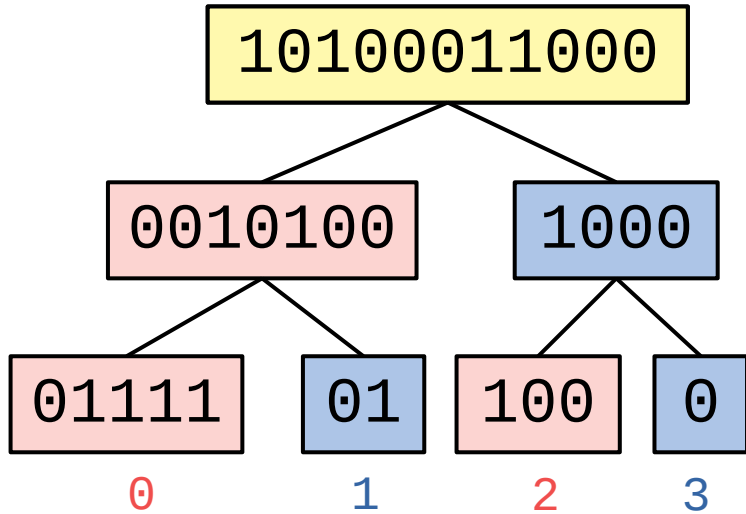
Tree

Matrix

Wavelet Tree vs Wavelet Matrix

Tree

10100011000

0010100   1000

01111   01   100   0

0   1   2   3

Matrix

10100011000

0010100   1000
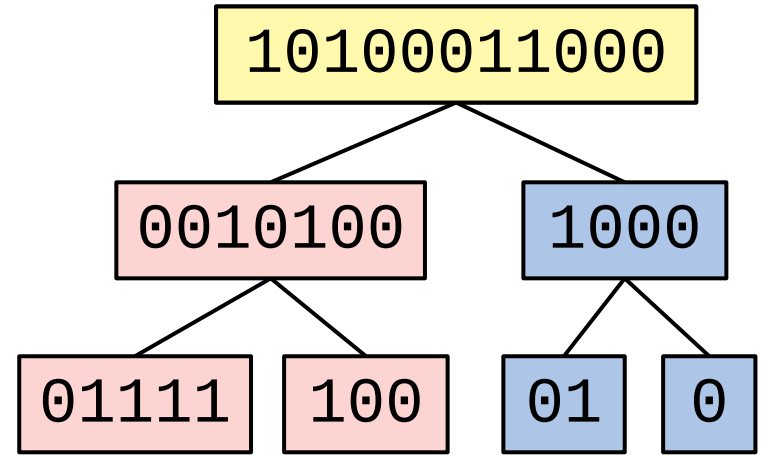
01111   100   01   0

Wavelet Tree vs Wavelet Matrix

Tree

Matrix

Wavelet Tree vs Wavelet Matrix

Tree

10100011000

0010100 — 1000

01111 — 01 — 100 — 0

0   1   2   3

00_b 01_b 10_b 11_b

Matrix

10100011000

0010100 — 1000

01111 — 100 — 01 — 0

0   2   1   3

00_b 10_b 01_b 11_b

46

Wavelet Tree vs Wavelet Matrix

Tree

Matrix

10100011000

0010100   1000

01111 01 100 0

0   1   2   3

00_b 01_b 10_b 11_b

10100011000

0010100   1000

01111 100 01 0

0   2   1   3

00_b 10_b 01_b 11_b
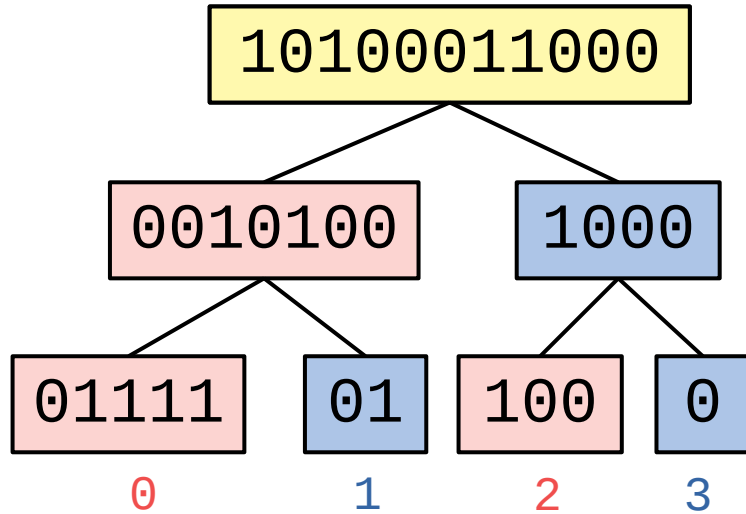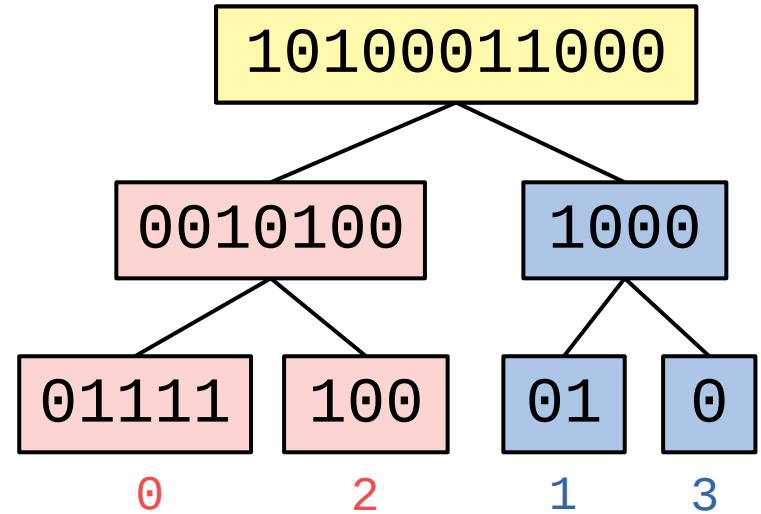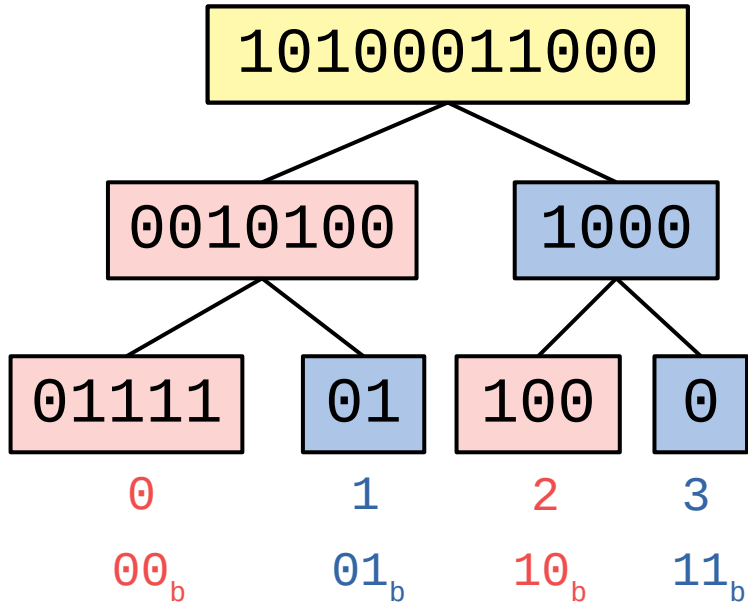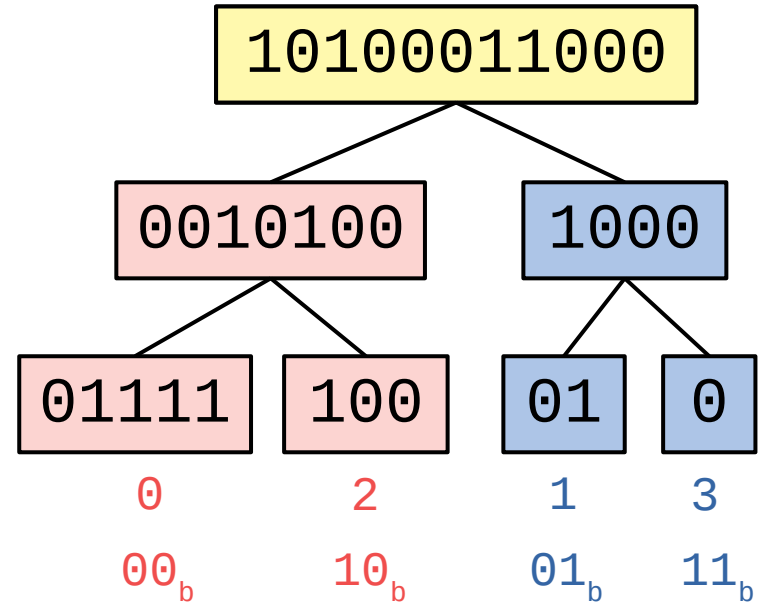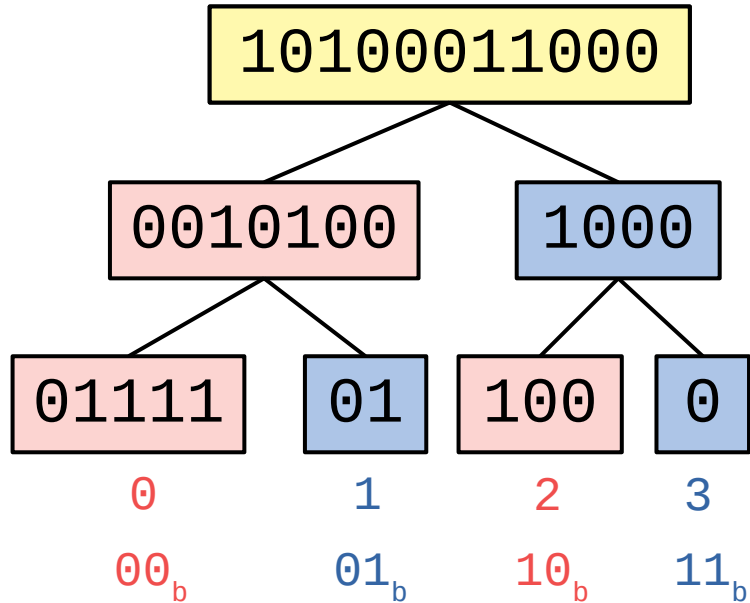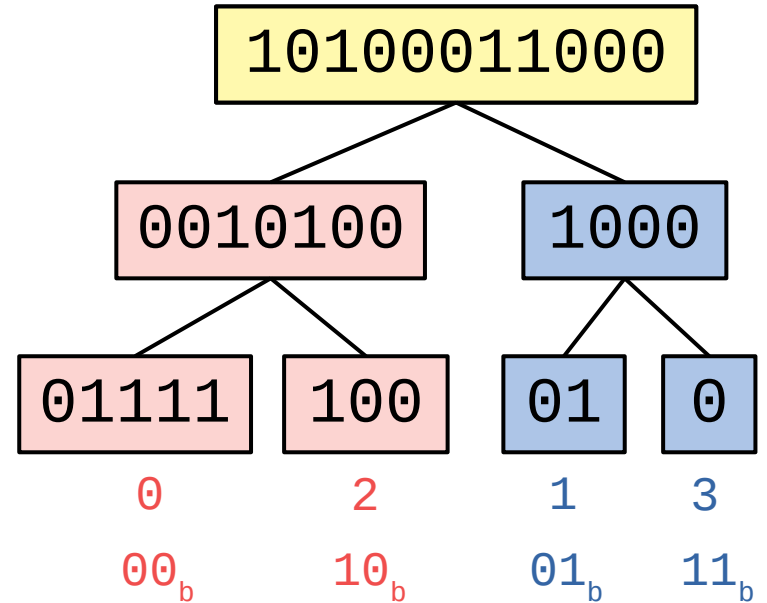
➤ bit-reversal permutation!

Translation Problem

Given: WTCA, level ℓ, position $i < n$

Translation Problem

Given: WTCA, level ℓ, position $i < n$

WTCA about to write i-th bit on level ℓ of the wavelet tree.

Translation Problem

Given: WTCA, level ℓ, position $i < n$

WTCA about to write i-th bit on level ℓ of the wavelet tree.

Find (efficiently): Position $j < n$

Translation Problem

Given: WTCA, level ℓ, position $i < n$

WTCA about to write i-th bit on level ℓ of the wavelet tree.

Find (efficiently): Position j < n

WTCA writes bit at position j instead and becomes WMCA.

Translation Problem

Given: WTCA, level ℓ, position $i < n$

WTCA about to write *i*-th bit on level ℓ of the wavelet tree.

Find (efficiently): Position *j* < n

WTCA writes bit at position *j* instead and becomes W**M**CA.

➢ solved by Fischer, Kurpicz & Löbel [ALENEX, 2018]
(DS of size $(n+\sigma)(1+o(1)) + (\sigma+2)\lceil lg\ n \rceil$ bits)

Translation Problem
The Inverse

Given: WMCA, level ℓ, position $j < n$

Translation Problem
The Inverse

Given: WMCA, level ℓ, position $j < n$

WMCA about to write j-th bit on level ℓ of the wavelet matrix.

Find (efficiently): Position $i < n$

WMCA writes bit at position i instead and becomes W**T**CA.

Translation Problem
The Inverse

Given: WMCA, level ℓ, position $j < n$

WMCA about to write *j*-th bit on level ℓ of the wavelet matrix.

Find (efficiently): Position i < n

WMCA writes bit at position *i* instead and becomes W*T*CA.

➢ **this work**

Strategy
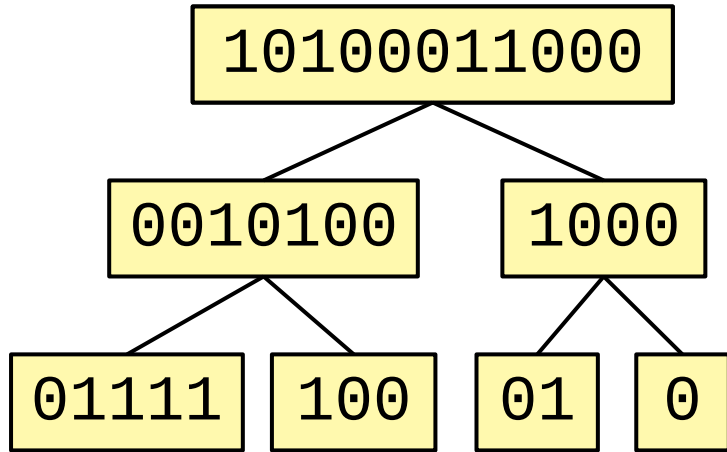
Strategy
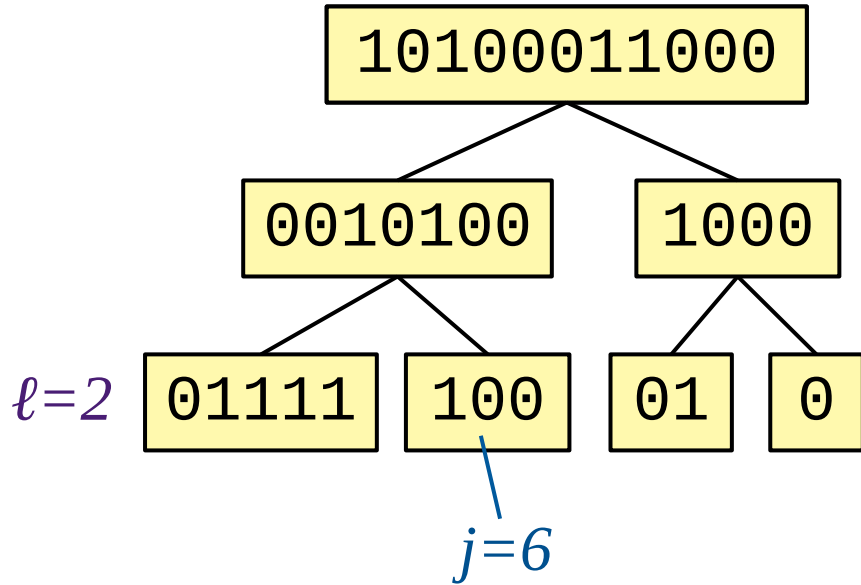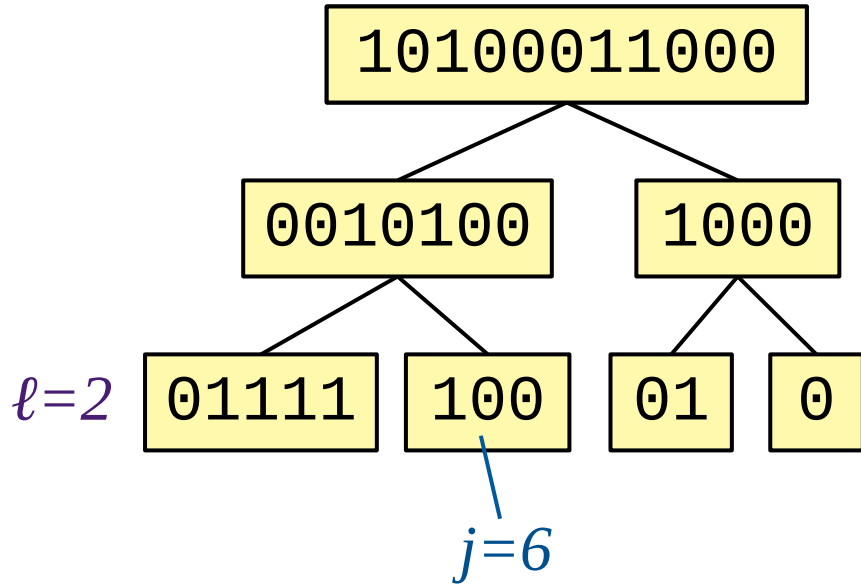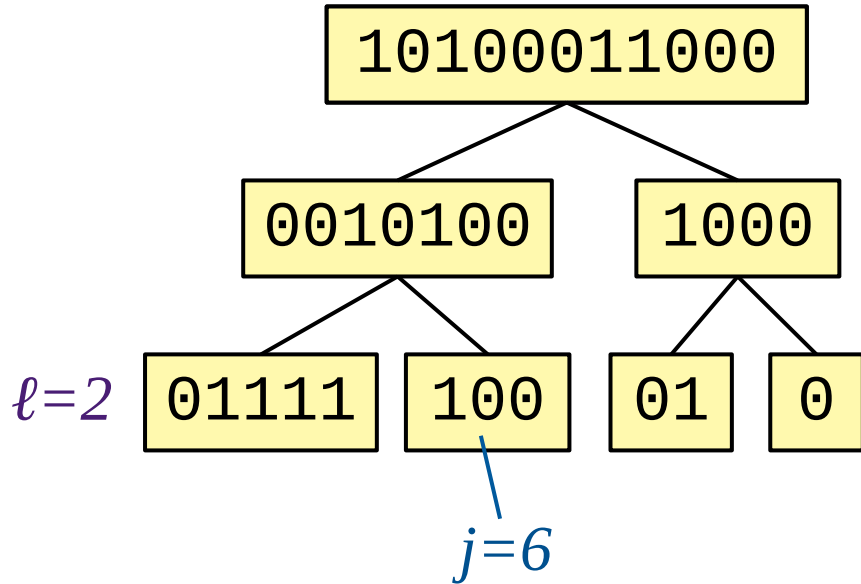
```
        ┌─────────────────┐
        │  10100011000    │
        └─────────────────┘
           ╱           ╲
  ┌─────────────┐   ┌──────────┐
  │  0010100    │   │  1000    │
  └─────────────┘   └──────────┘
     ╱       ╲         ╱     ╲
┌─────────┐ ┌───────┐ ┌─────┐ ┌─────┐
│ 01111   │ │ 100   │ │ 01  │ │ 0   │
└─────────┘ └───────┘ └─────┘ └─────┘
```

ℓ=2

j=6

Strategy

Locate:

10100011000

0010100   1000

$\ell=2$ 01111   100   01   0

$j=6$

Strategy

Locate:

- The node that contains pos. i or j

```
              10100011000

        0010100         1000

ℓ=2  01111   100    01    0

                 j=6
```

Strategy

Locate:

- The node that contains pos. *i* or *j*
- The position of its first bit

```
          10100011000
         /           \
    0010100          1000
    /     \         /    \
01111    100      01      0
```

ℓ=2   01111   100

j=6

Strategy

Locate:

- The node that contains pos. *i* or *j*
- The position of its first bit

Then:

10100011000

0010100   1000

ℓ=2   01111   100   01   0

j=6
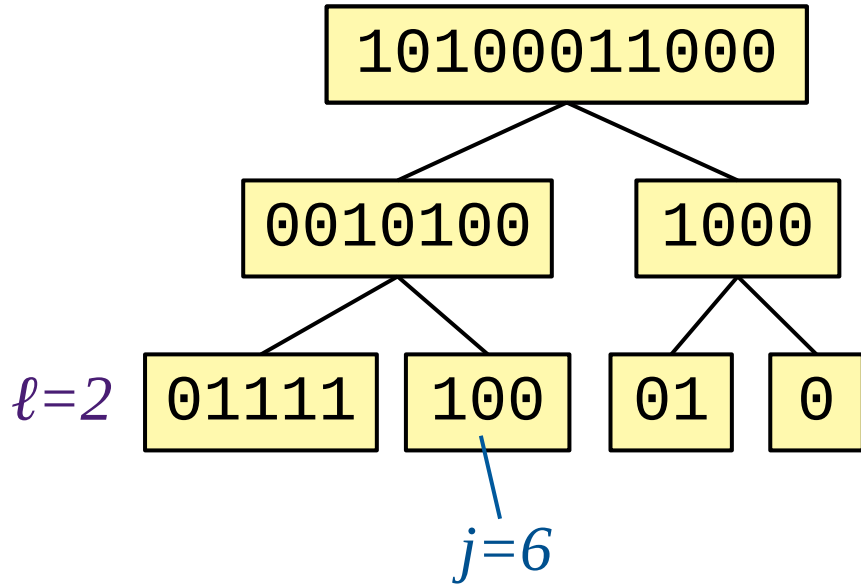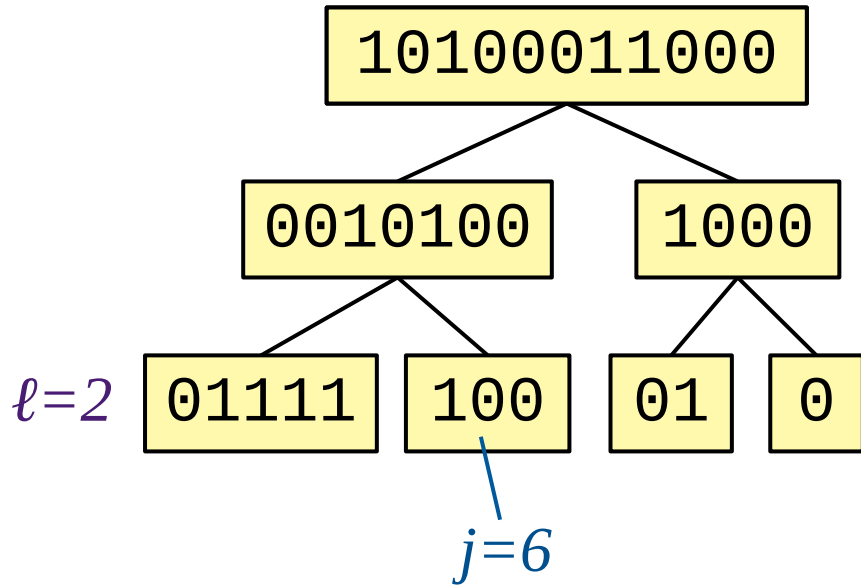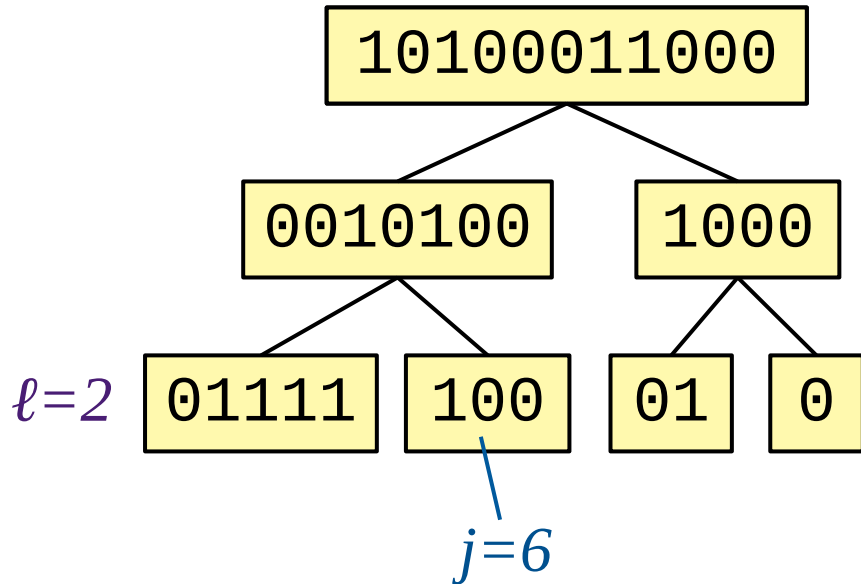
Strategy

Locate:

- The node that contains pos. *i* or *j*
- The position of its first bit

Then:

➢ Use bit reversal to find counterpart node

Strategy

Locate:

- The node that contains pos. *i* or *j*
- The position of its first bit

Then:

➢ Use bit reversal to find counterpart node

➢ Find the node's first bit in counterpart

10100011000

0010100   1000

ℓ=2 01111   100   01   0

j=6

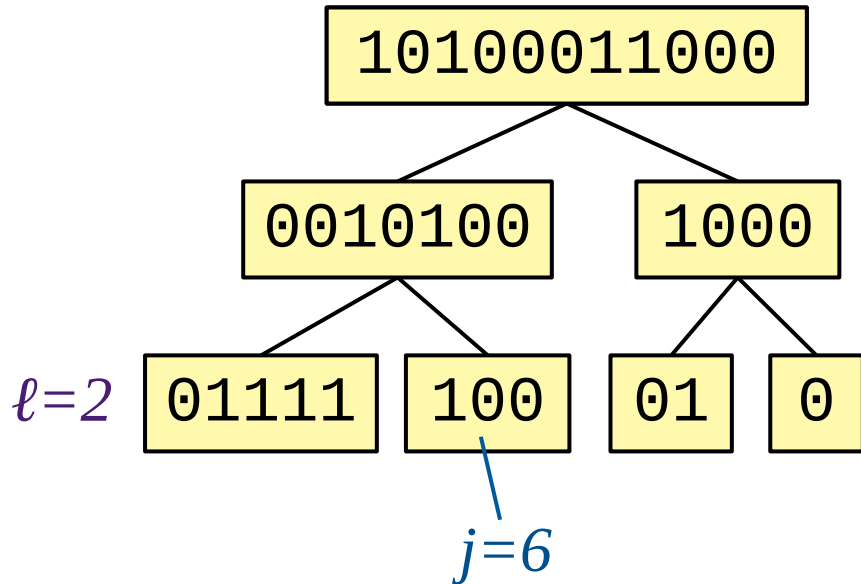# Strategy
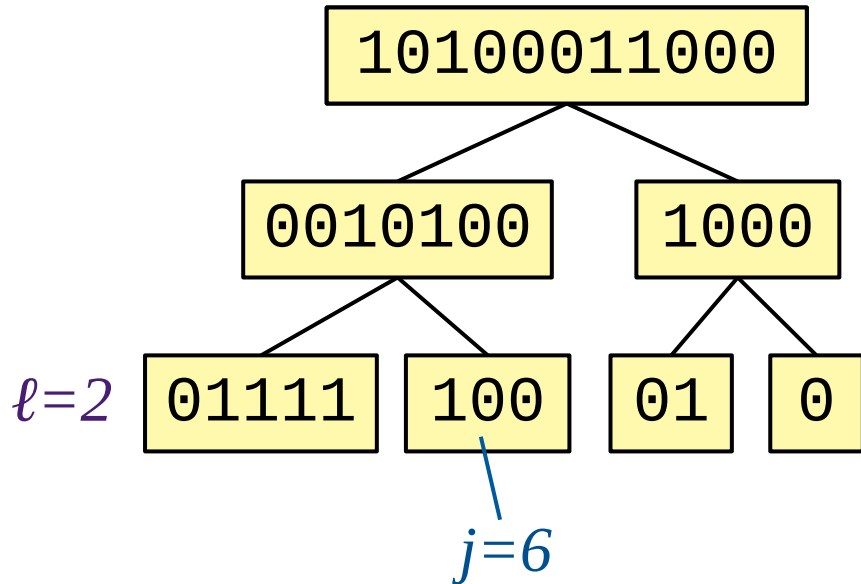
Locate:
- The node that contains pos. *i* or *j*
- The position of its first bit

Then:
➢ Use bit reversal to find counterpart node
➢ Find the node's first bit in counterpart
➢ Add offset

```
          10100011000

   0010100          1000

01111   100      01     0
```

ℓ=2

j=6

Virtual Bottom Level

{a,e,l,r,t,v,w}

wavelettree
10100011000

{a,e,l,r}

aeleree
0010100

{t,v,w}

wvtt
1000

{a,e}

aeeee
01111

{l,r}

lr
01

{t,v}

vtt
100

{w}

w
0

Virtual Bottom Level

{a,e,l,r,t,v,w}

wavelettree
10100011000

{a,e,l,r}

aeleree
0010100

{t,v,w}

wvtt
1000

{a,e}

aeeee
01111

{l,r}

lr
01

{t,v}

vtt
100

{w}

w
0

a	eeee
0	0000

l	r
0	0

tt	v
00	0

w	
0	

Virtual Bottom Level

wavelettree

a eeee l r   tt v w

Virtual Bottom Level

wavelettree

1	4	1	1		2	1	1	0
a	eeee	l	r		tt	v	w	
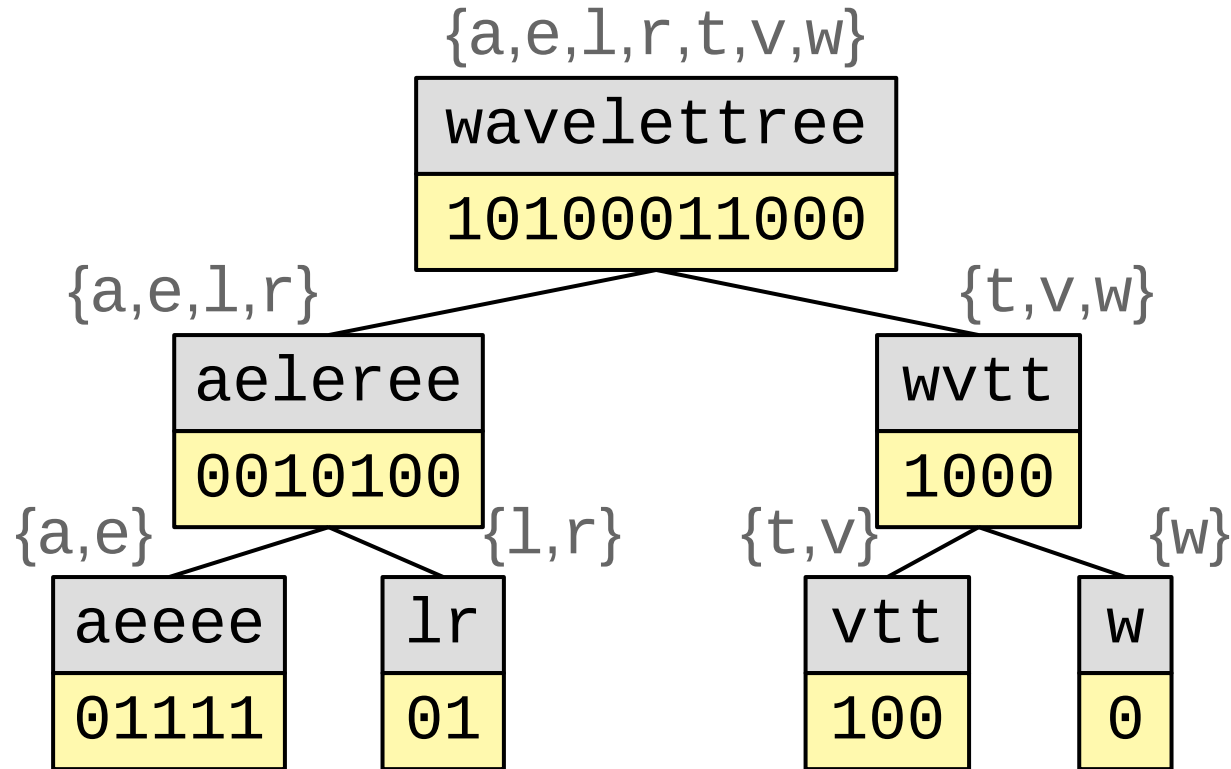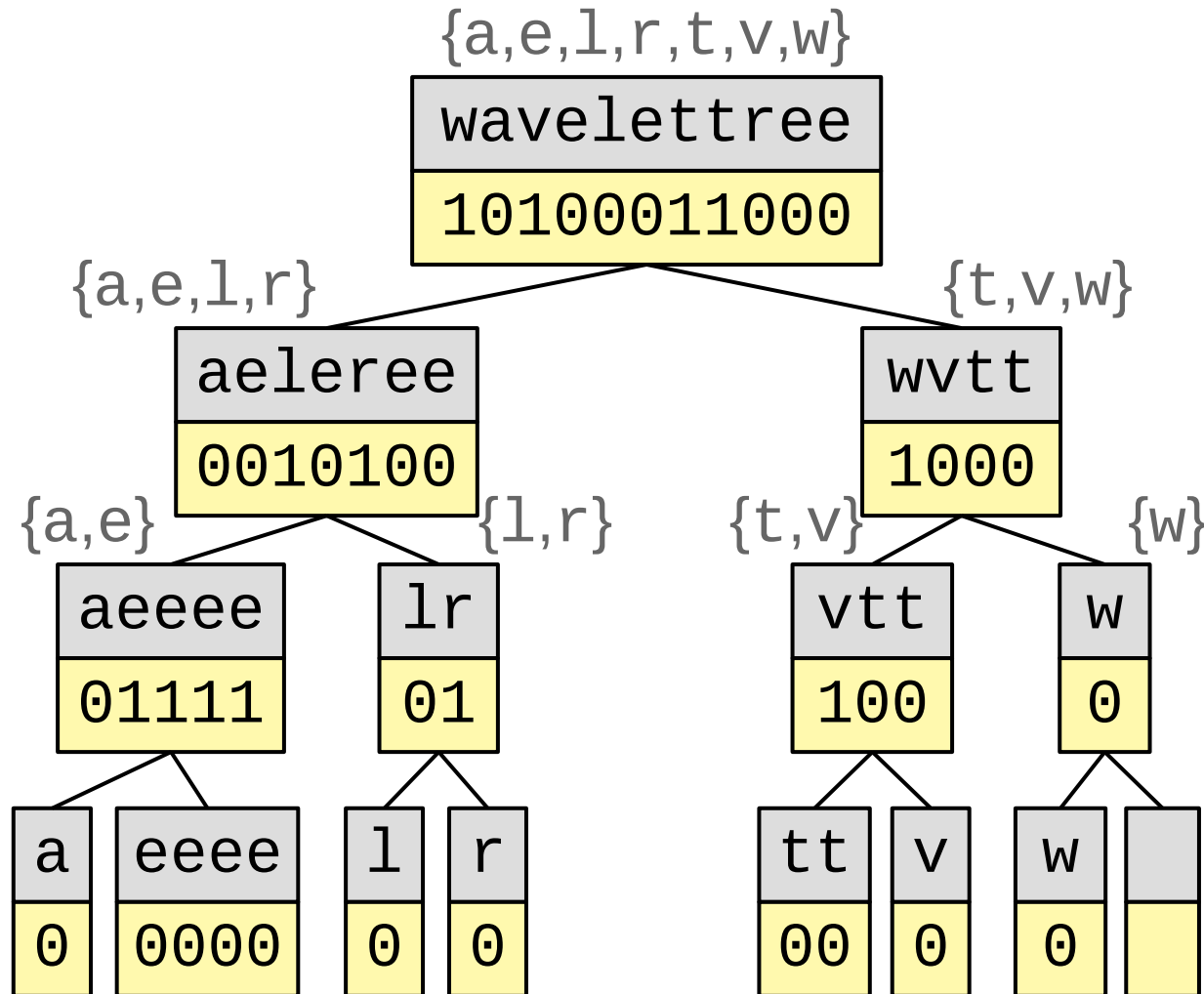
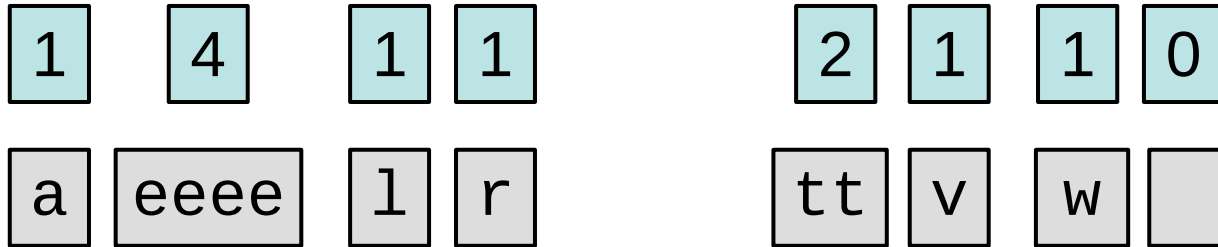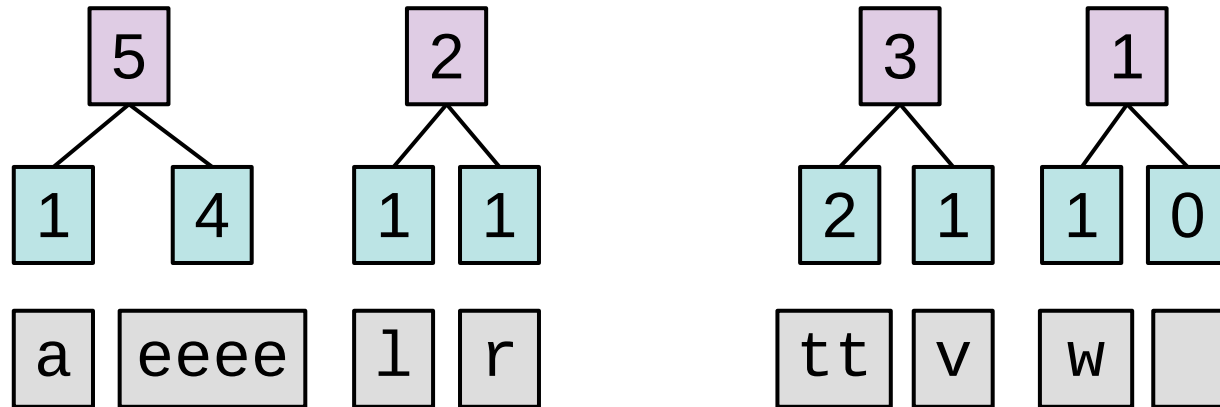# Virtual Bottom Level

wavelettree

Virtual Bottom Level
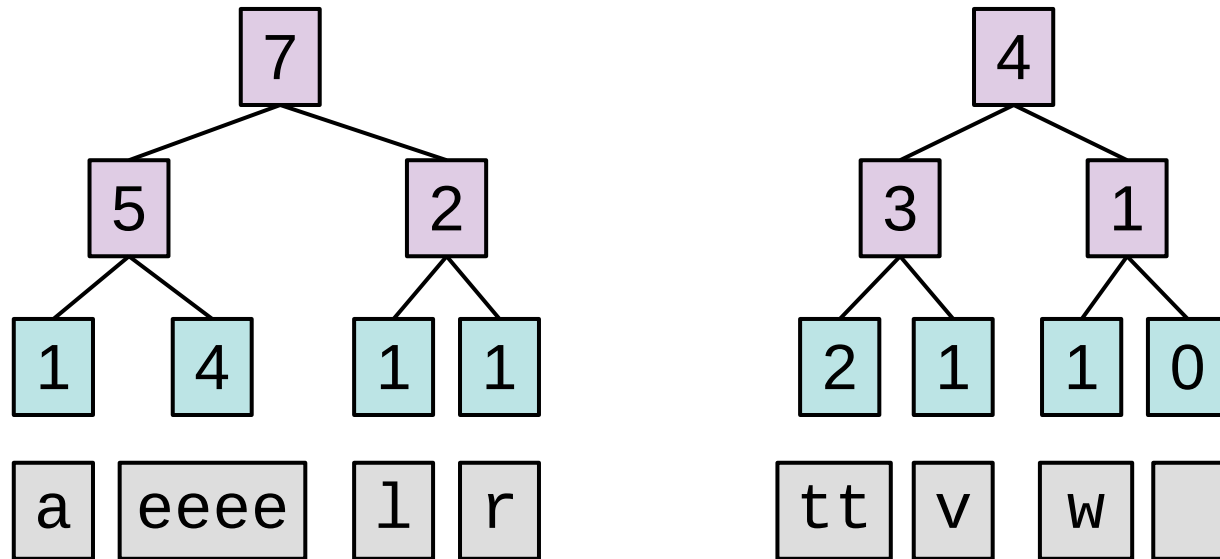
Virtual Bottom Level
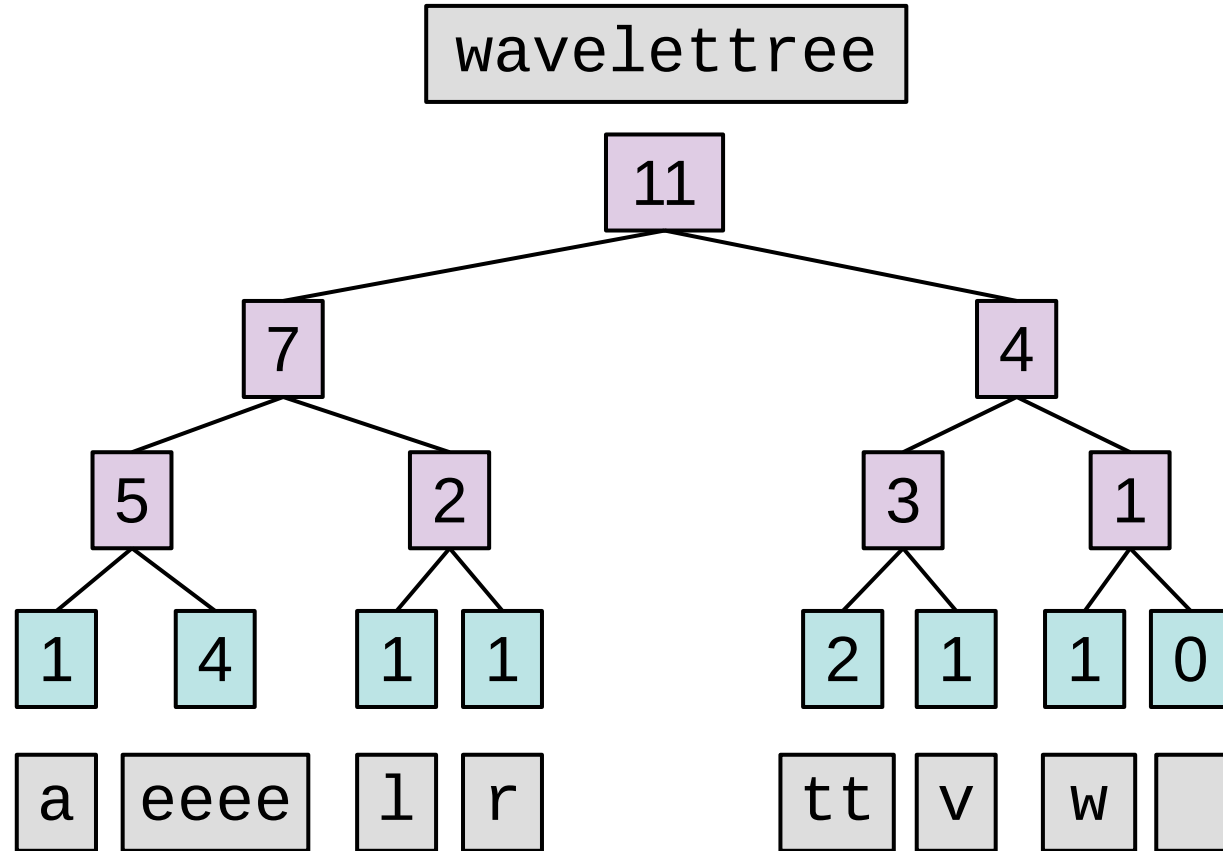
Virtual Bottom Level
encodes all node sizes!
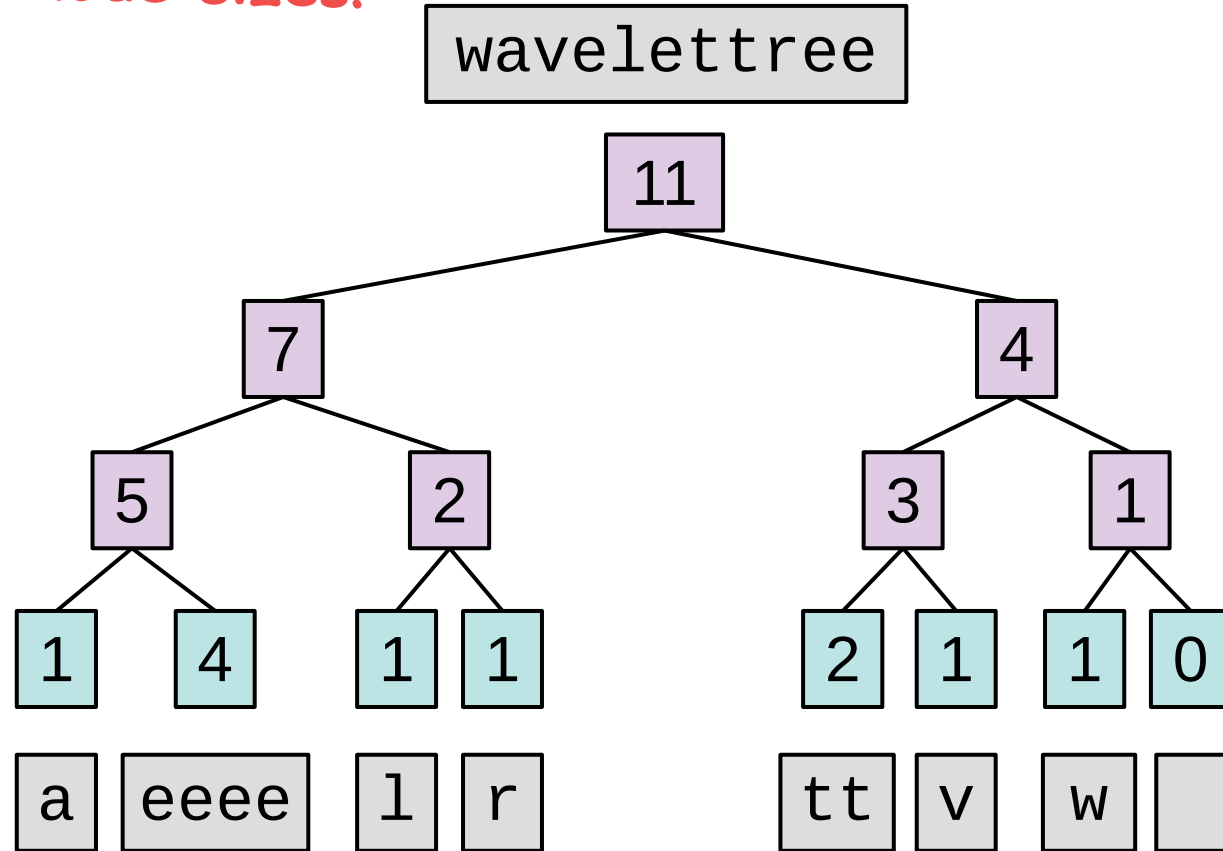
The C Array

c	a	e	l	r	t	v	w	
	0	1	2	3	4	5	6	7

The C Array

c	a	e	l	r	t	v	w	
	0	1	2	3	4	5	6	7
$H[c]$	1	4	1	1	2	1	1	

The C Array

c	a	e	l	r	t	v	w	
	0	1	2	3	4	5	6	7
$H[c]$	1	4	1	1	2	1	1	
$C[c]$	0	1	5	6	7	9	10	11

The C Array

c	a 0	e 1	l 2	r 3	t 4	v 5	w 6	7
$H[c]$	1	4	1	1	2	1	1	
$C[c]$	0	1	5	6	7	9	10	11

$$H[c] = C[c+1] - C[c]$$

The C Array

c	a	e	l	r	t	v	w	
	0	1	2	3	4	5	6	7
H[c]	1	4	1	1	2	1	1	
C[c]	0	1	5	6	7	9	10	11

$$H[c] = C[c+1] - C[c]$$

Locating in the WT
on the virtual bottom level
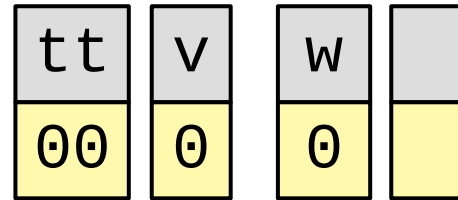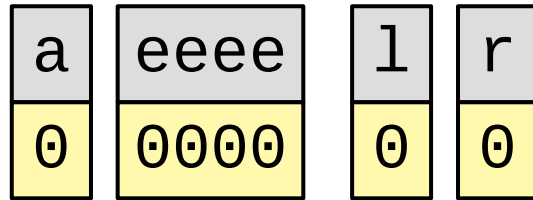
a	eeee	l	r
0	0000	0	0

tt	v	w	
00	0	0	

Locating in the WT
on the virtual bottom level

c 0   1   2   3   4   5   6   7

| a | eeee | l | r | | tt | v | w | |
| 0 | 0000 | 0 | 0 | | 00 | 0 | 0 | |

Locating in the WT
on the virtual bottom level

c	0	1	2	3		4	5	6	7
	a	eeee	l	r		tt	v	w	
	0	0000	0	0		00	0	0	
$C[c]$	0	1	5	6		7	9	10	11	11

Locating in the WT
on the virtual bottom level

c 0   1   2   3   4   5   6   7

a	eeee	l	r
0	0000	0	0

tt	v	w	
00	0	0	

$C[c]$ 0   1   5   6   7   9   10 11   11
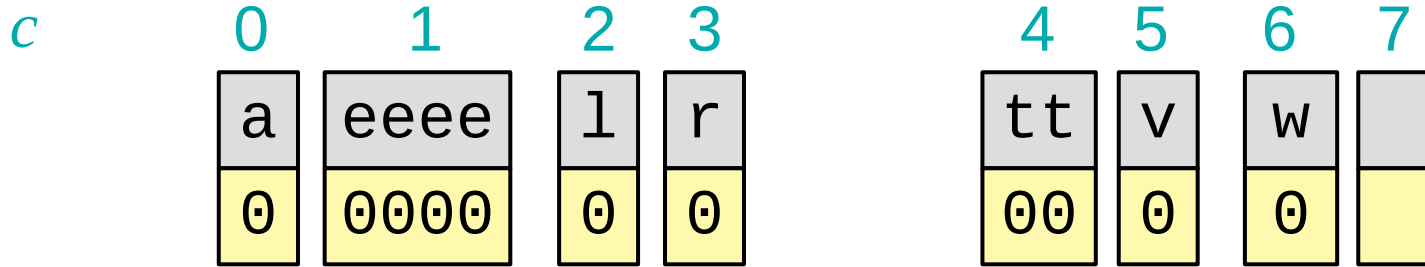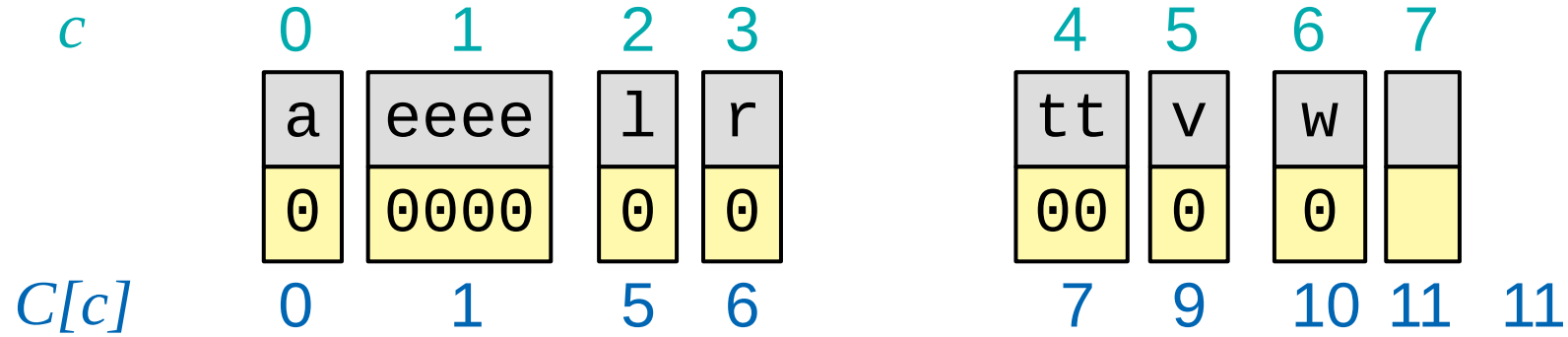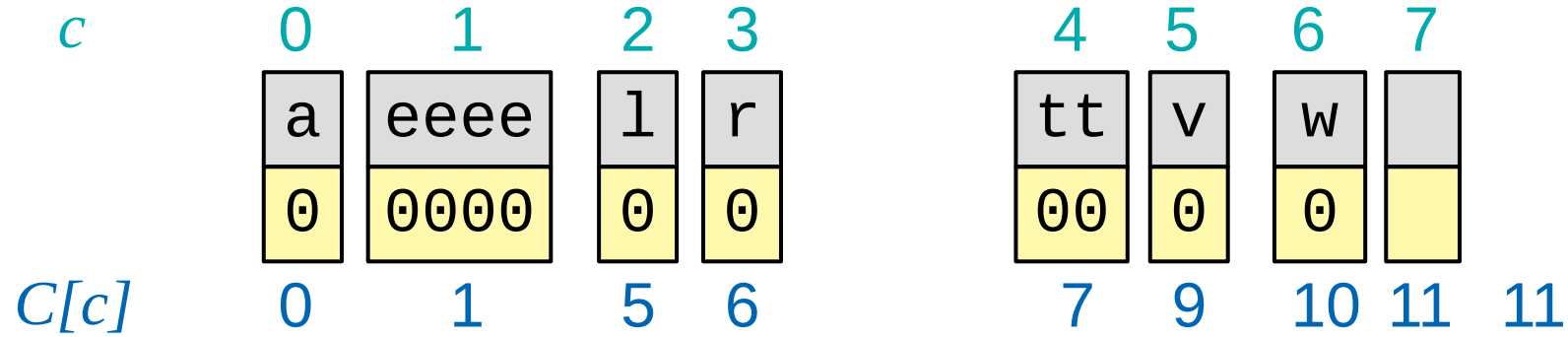
Given position i:

Locating in the WT
on the virtual bottom level

c   0   1   2   3   4   5   6   7

a	eeee	l	r
0	0000	0	0

tt	v	w	
00	0	0	

C[c]   0   1   5   6   7   9   10 11 11

Given position *i*:

- Find $v = min\{c \mid C[c] > i\} - 1$

Locating in the WT
on the virtual bottom level

c	0	1	2	3	4	5	6	7

a	eeee	l	r		tt	v	w	
0	0000	0	0		00	0	0	

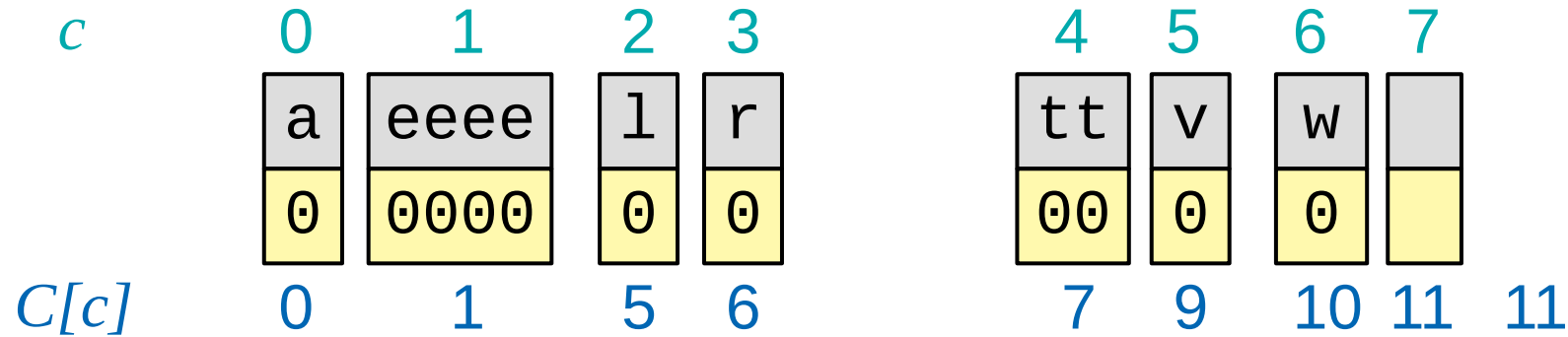|									
C[c]	0	1	5	6	7	9	10	11	11

Given position i:

- Find $v = min\{c \mid C[c] > i\} - 1$

- v's first bit is located at $C[v]$

83

Locating in the WT
on the virtual bottom level

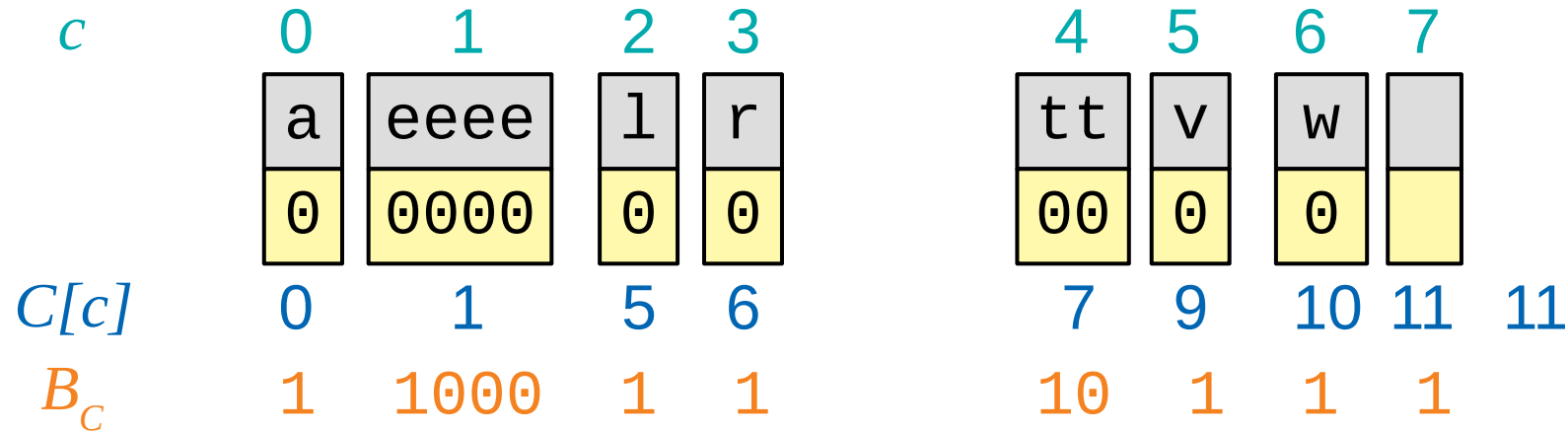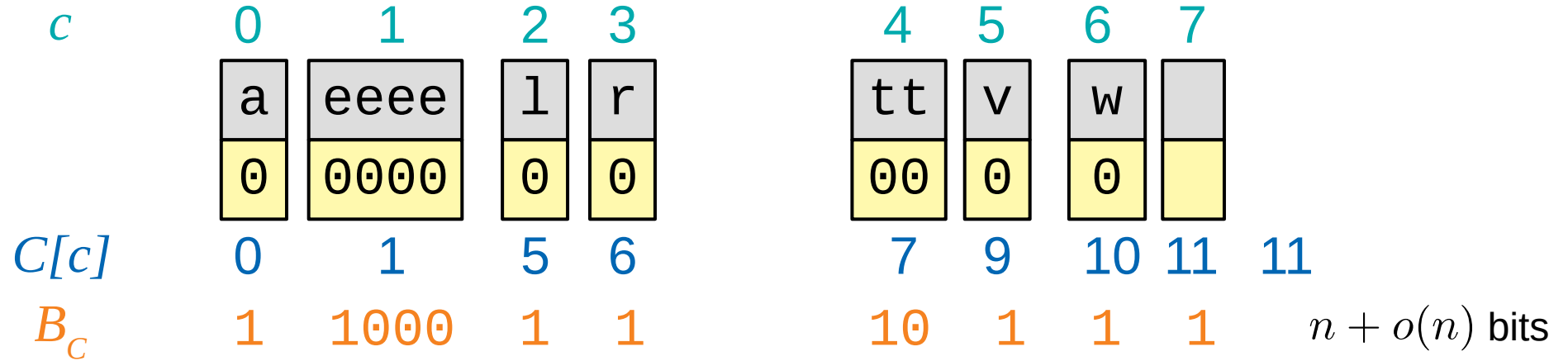|										
c	0	1	2	3		4	5	6	7	
	a	eeee	l	r		tt	v	w		
	0	0000	0	0		00	0	0		
C[c]	0	1	5	6		7	9	10	11	11

Given position i:

- Find $v = min\{c \mid C[c] > i\} - 1$ in constant time!
- v's first bit is located at $C[v]$

Locating in the WT
on the virtual bottom level

c	0	1	2	3		4	5	6	7
	a	eeee	l	r		tt	v	w	
	0	0000	0	0		00	0	0	
$C[c]$	0	1	5	6		7	9	10	11	11
B_C	1	1000	1	1		10	1	1	1

Given position i:

- Find $v = min\{c \mid C[c] > i\} - 1$ in constant time!
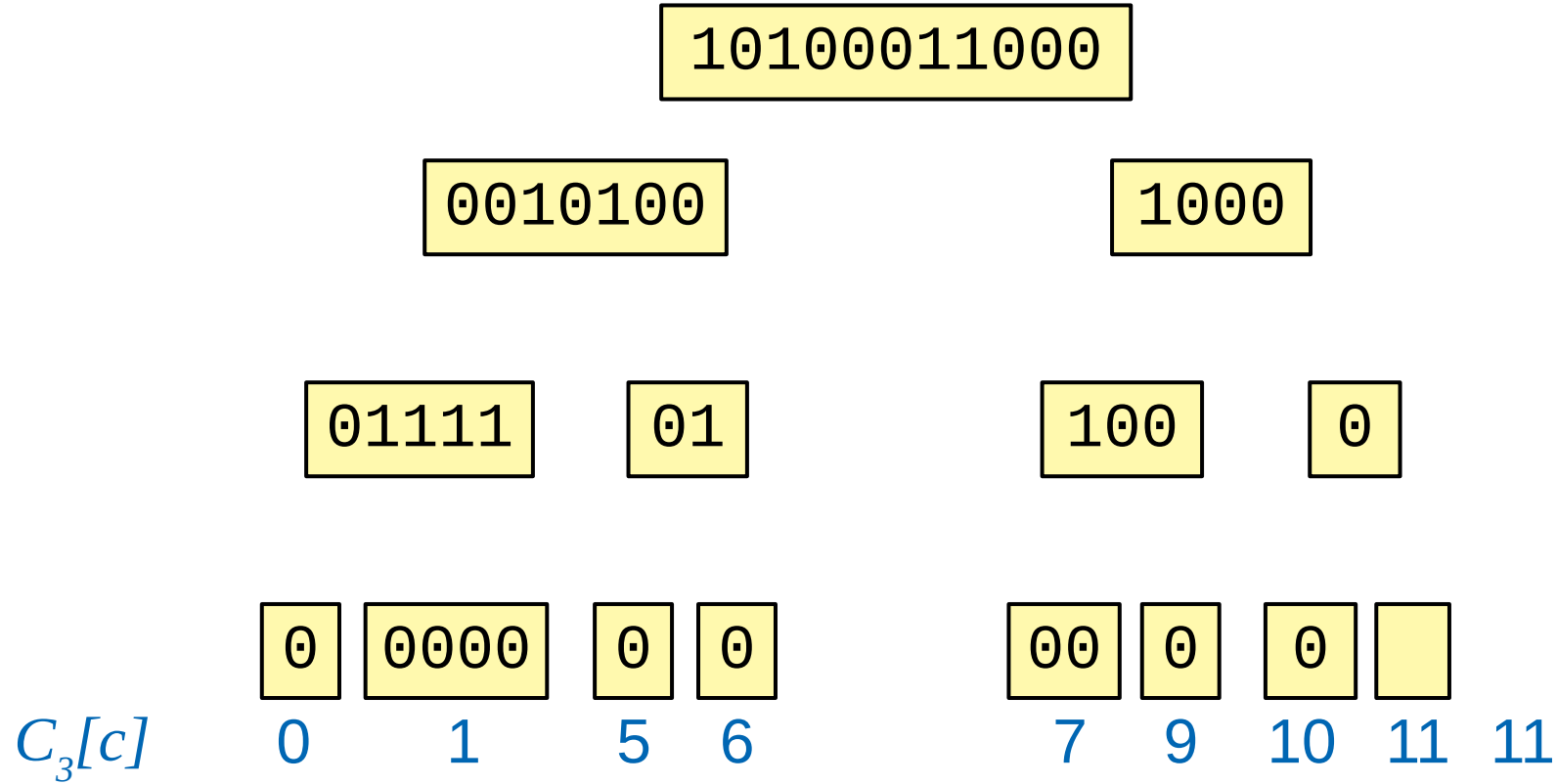- v's first bit is located at $C[v]$

85

Locating in the WT
on the virtual bottom level

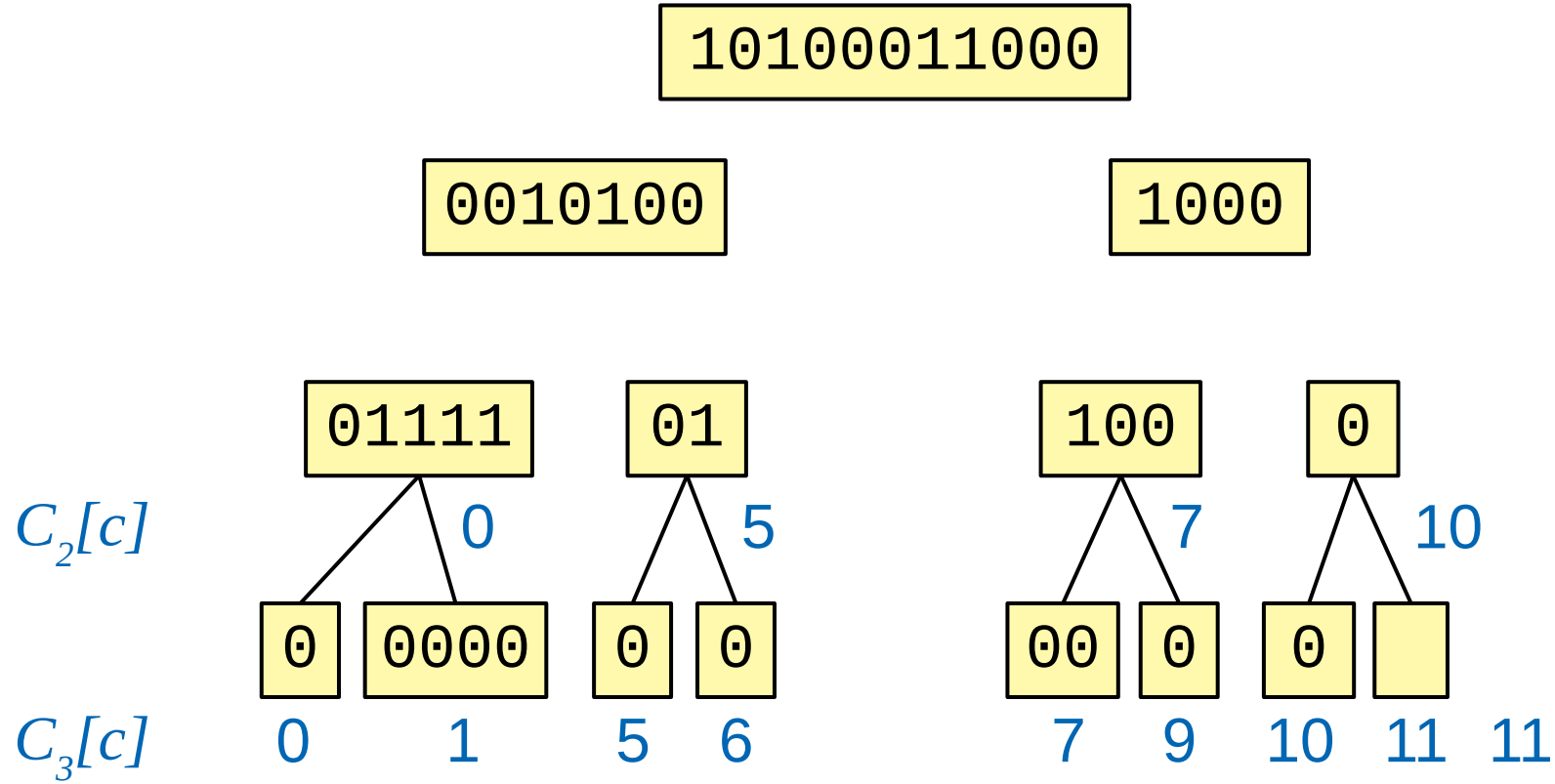| c	0	1	2	3		4	5	6	7	
	a	eeee	l	r		tt	v	w		
	0	0000	0	0		00	0	0		
$C[c]$	0	1	5	6		7	9	10	11	11
B_C	1	1000	1	1		10	1	1	1	

$n + o(n)$ bits

Given position i:

- Find $v = min\{c \mid C[c] > i\} - 1$ in constant time!

- v's first bit is located at $C[v]$

Locating in the WT
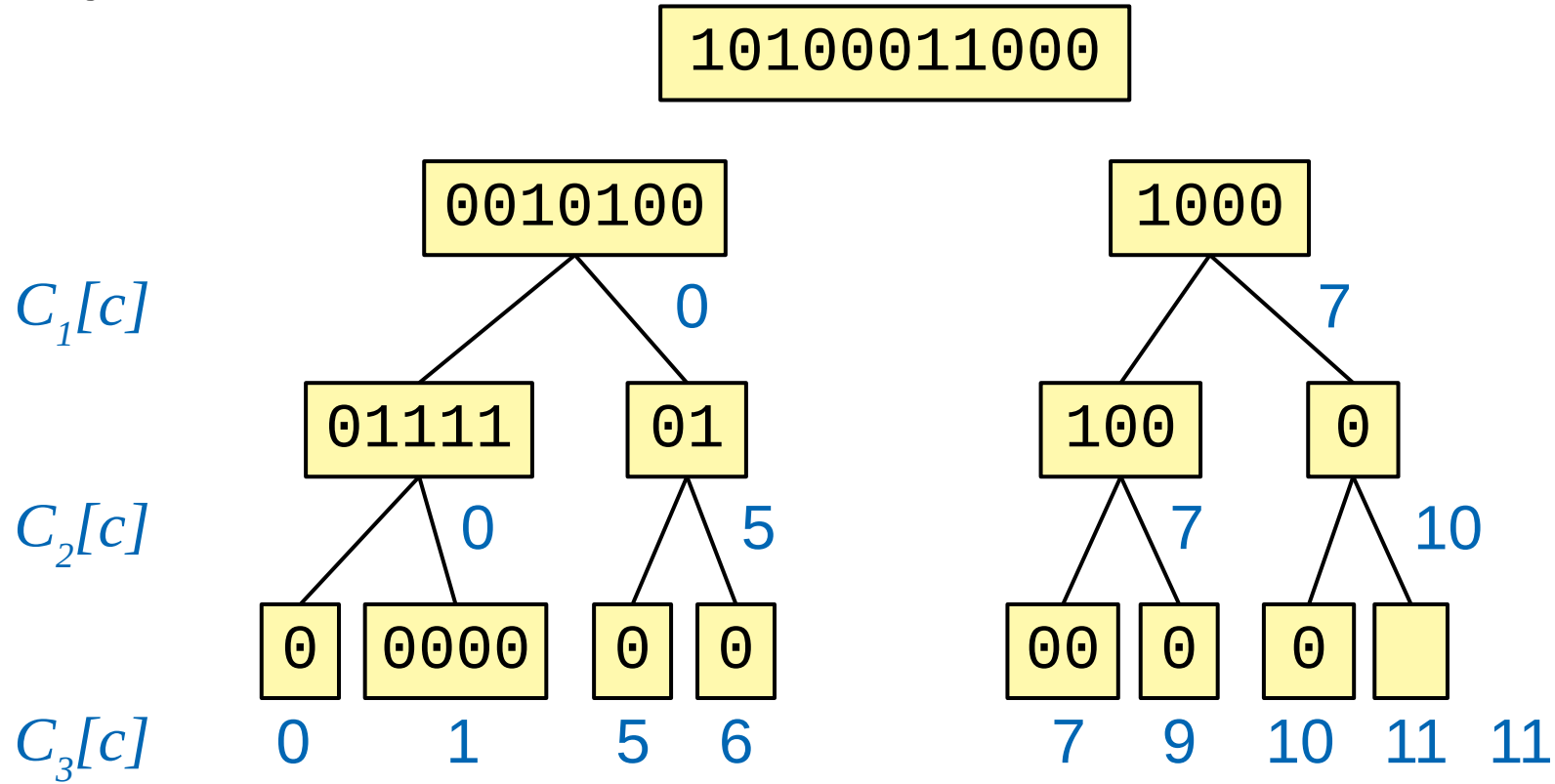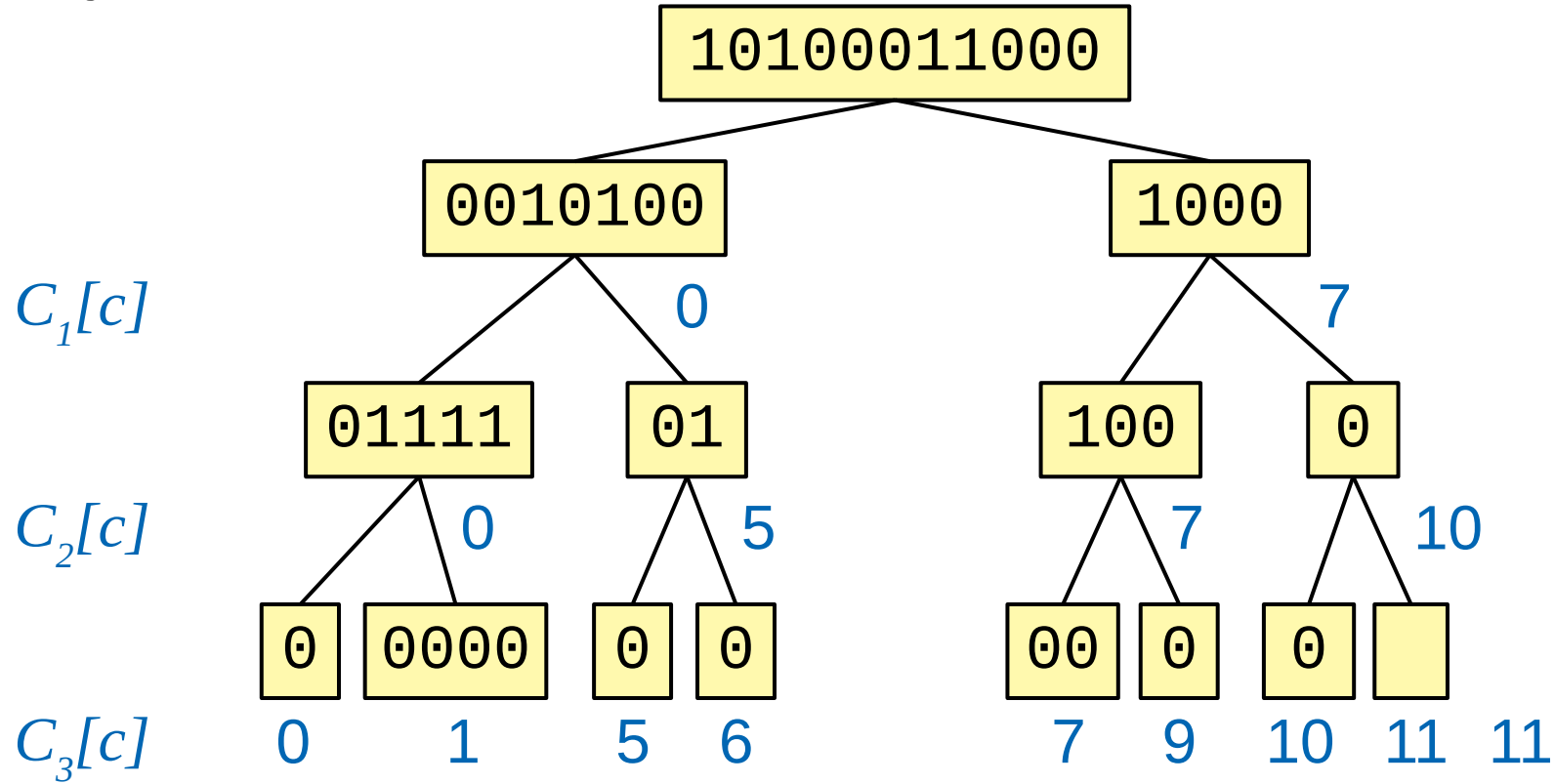## on any level

10100011000

0010100   1000

01111   01   100   0

0   0000   0   0   00   0   0

$C_3[c]$ 0   1   5   6   7   9   10   11   11

Locating in the WT
on any level

10100011000

0010100

1000

01111

01

100

0

$C_2[c]$

0   5

7   10

0   0000   0   0

00   0   0

$C_3[c]$ 0   1   5   6

7   9   10   11   11

Locating in the WT
on any level

10100011000

0010100

$C_1[c]$ 0

01111   01

$C_2[c]$ 0   5

0   0000   0   0

$C_3[c]$ 0   1   5   6

1000

7

100   0

7   10

00 0 0

7 9 10 11 11

Locating in the WT
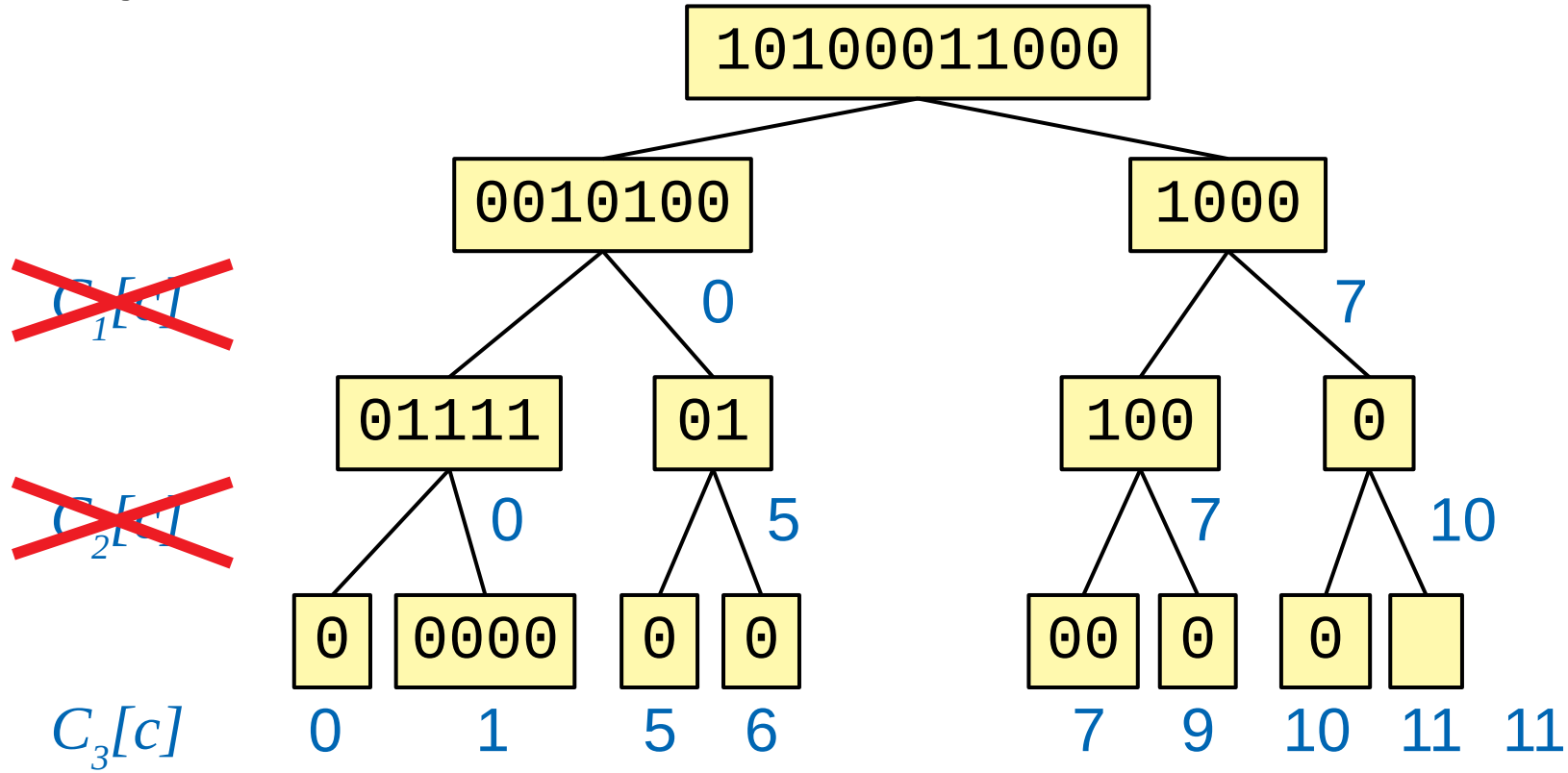## on any level

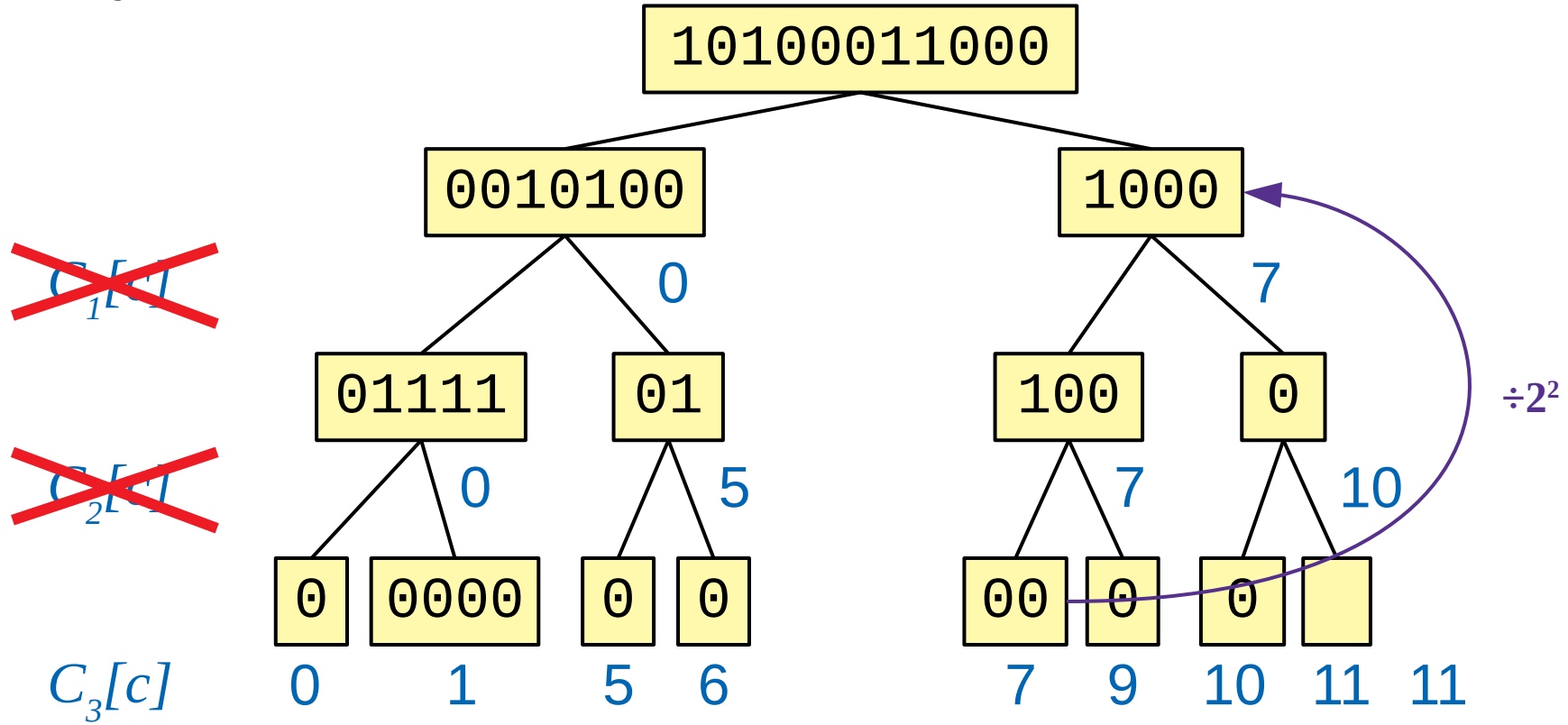$C_1[c]$

$C_2[c]$

$C_3[c]$

Locating in the WT
on any level

Locating in the WT
on any level

Locating in the WT
on any level

$C_1[c]$

$C_2[c]$

$C_3[c]$

10100011000

0010100   1000

0   7

01111   01   100   0

0   5   7   10

0   0000   0   0   00   0   0   7   9   10   11

0   1   5   6   7   9   10   11   11

$\times 2^2$ $\div 2^2$

Locating in the WT
on any level

$C_1[c]$ (crossed out)

$C_2[c]$ (crossed out)

$C_3[c]$	0	1	5	6	7	9	10	11	11
B_C	1	1000	1	1	10	1	1	1

×2²   ÷2²

Locating in the WT
on any level

$C_1[c]$

$C_2[c]$

$C_3[c]$

10100011000									

0010100   1000

0   7

01111   01   100   0

0   5   7   10

×2²   ÷2²

0	0000	0	0		00	0	0	

0   1   5   6   7   9   10   11   11

B_C 1   1000   1   1   10   1   1   1

95

Locating in the WT
Summary

Bit vector B_C with rank/select suffices!

Locating in the WT
Summary

Bit vector B_C with rank/select suffices!

➢ Construct in time $O(n+\sigma)$

Locating in the WT
Summary

Bit vector B_C with rank/select suffices!

- ➢ Construct in time $O(n+\sigma)$
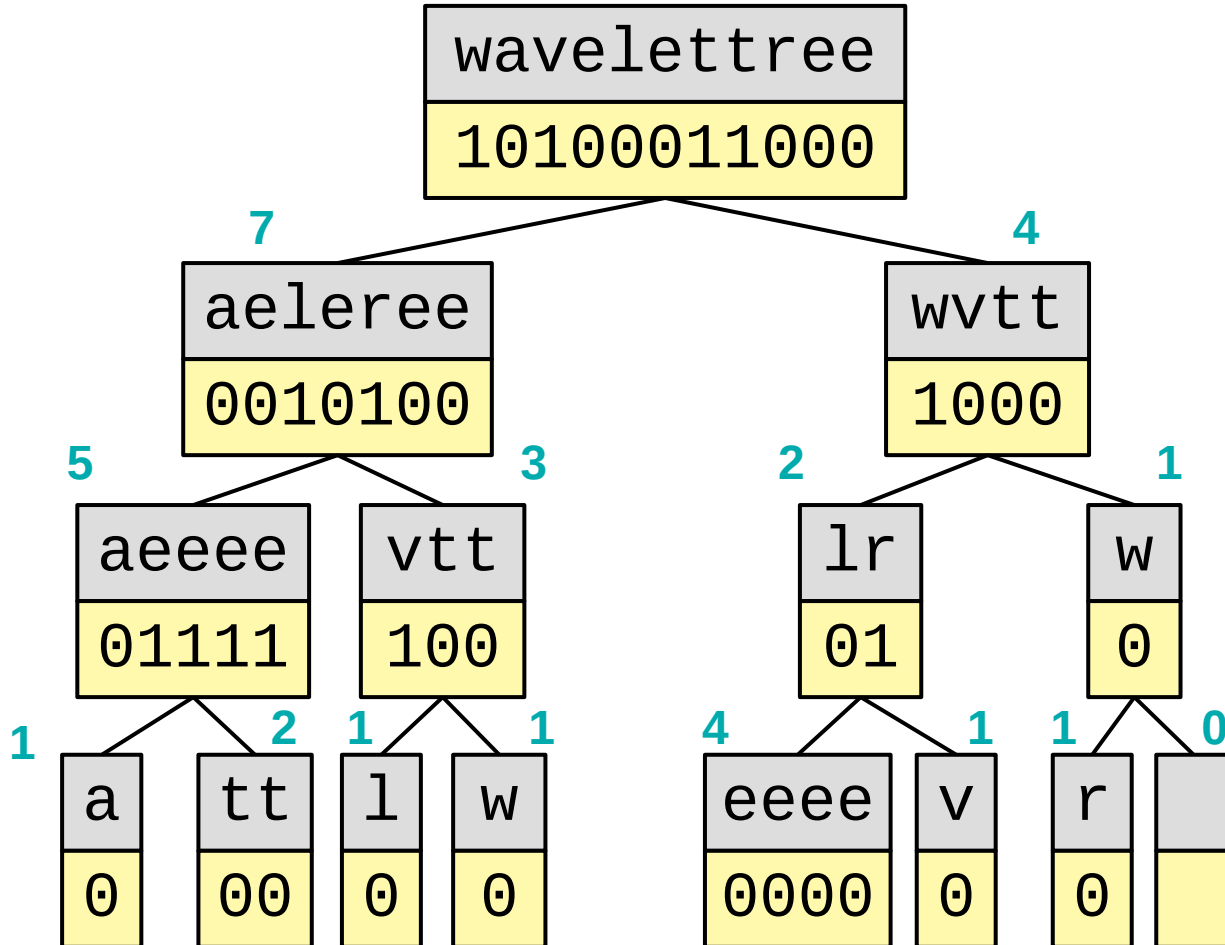- ➢ Store in $n + o(n)$ bits

Locating in the WT
Summary

Bit vector B_C with rank/select suffices!

- ➢ Construct in time $O(n+\sigma)$
- ➢ Store in $n + o(n)$ bits
- ➢ Translate position in constant time

Bit vector B_C with rank/select suffices!

- ➤ Construct in time $O(n+\sigma)$
- ➤ Store in $n + o(n)$ bits
- ➤ Translate position in constant time

➢ Smaller than Fischer, Kurpicz & Löbel [ALENEX, 2018]
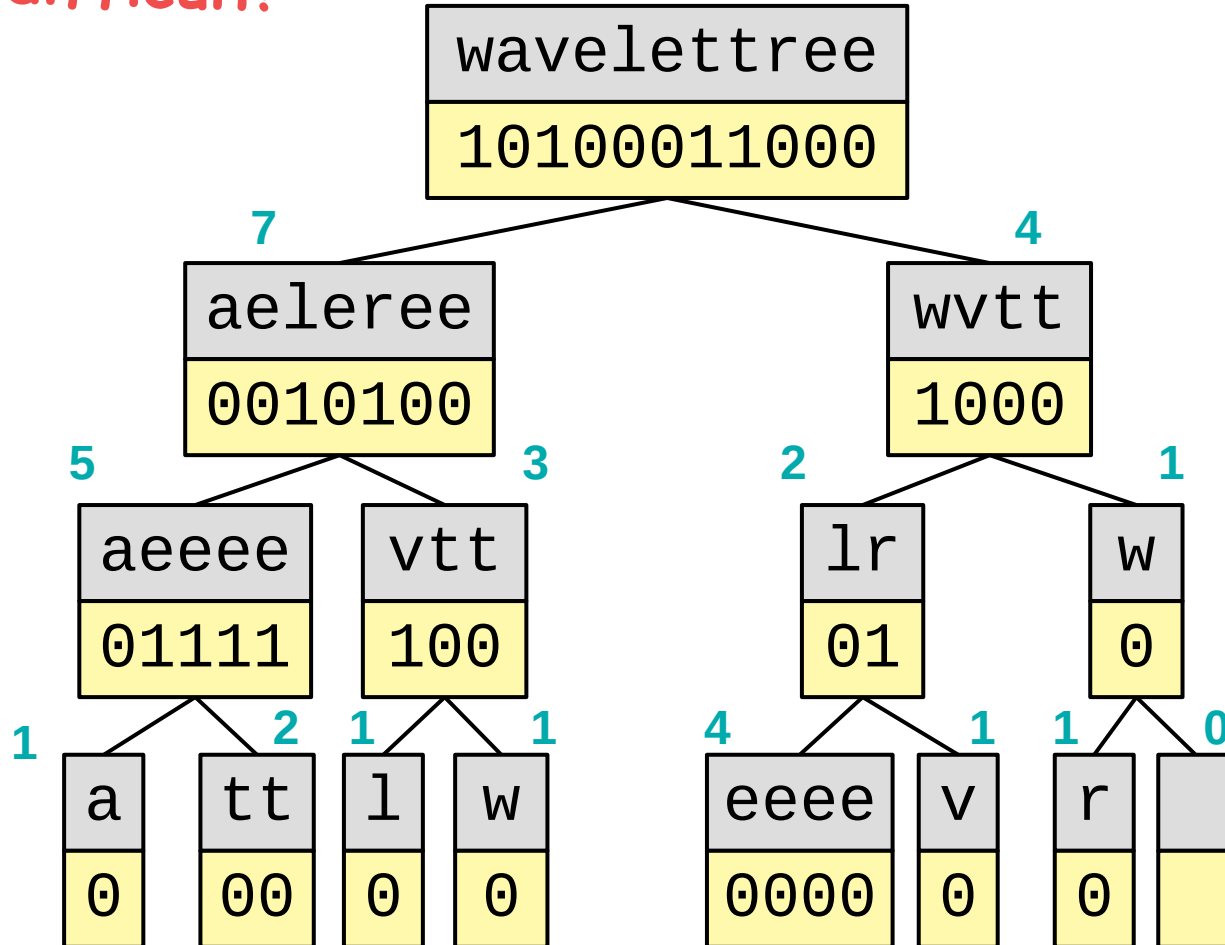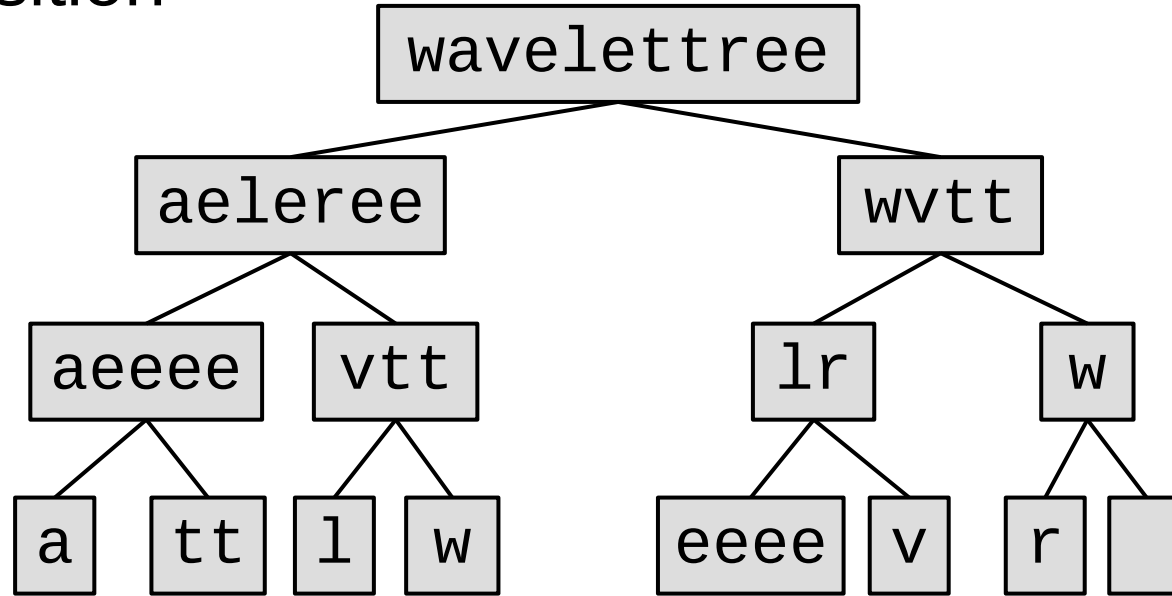
Locating in the WM
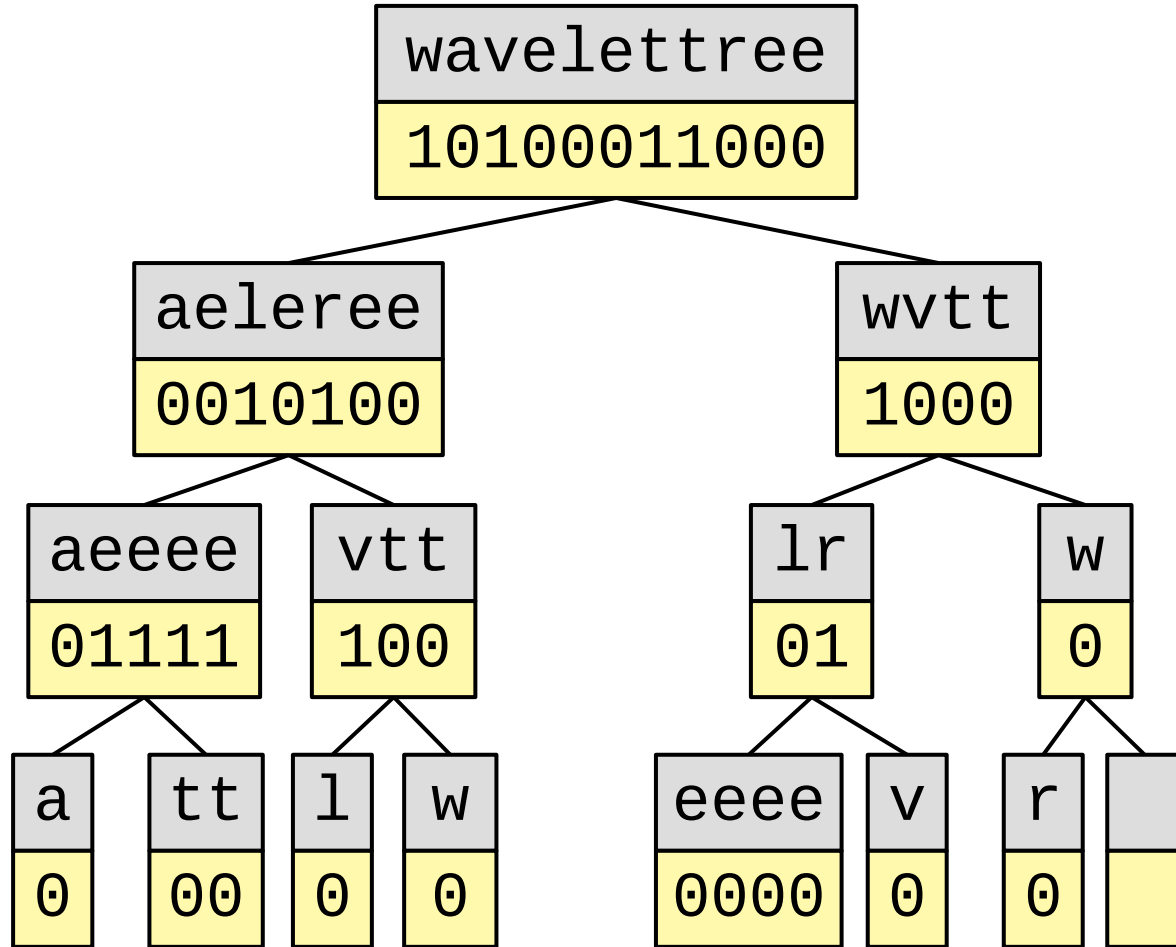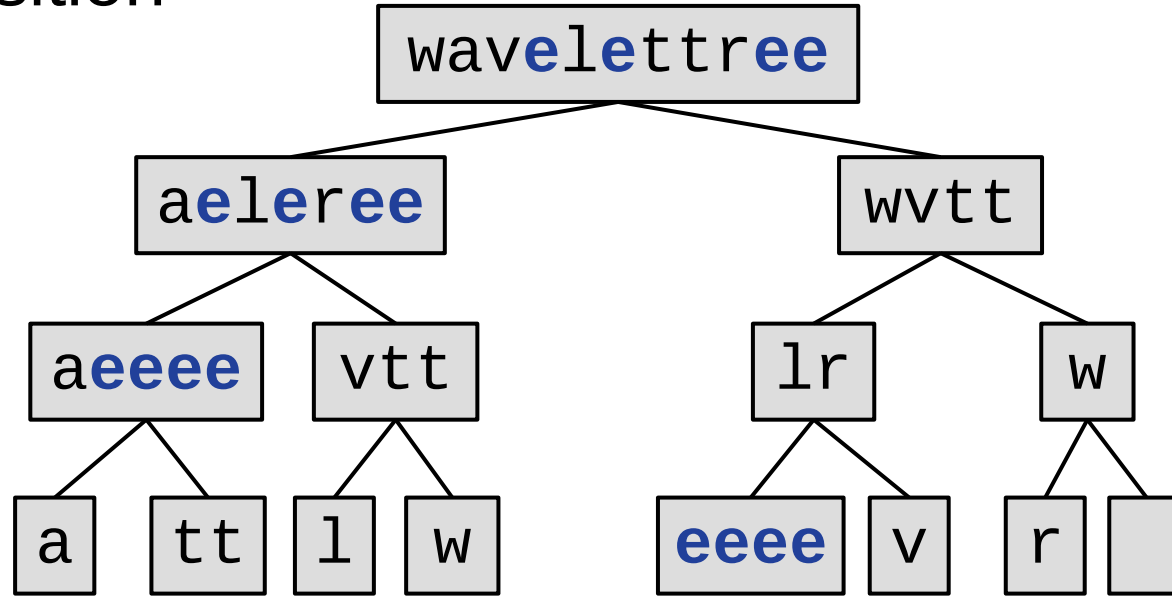
Locating in the WM

Locating in the WM

is more difficult!

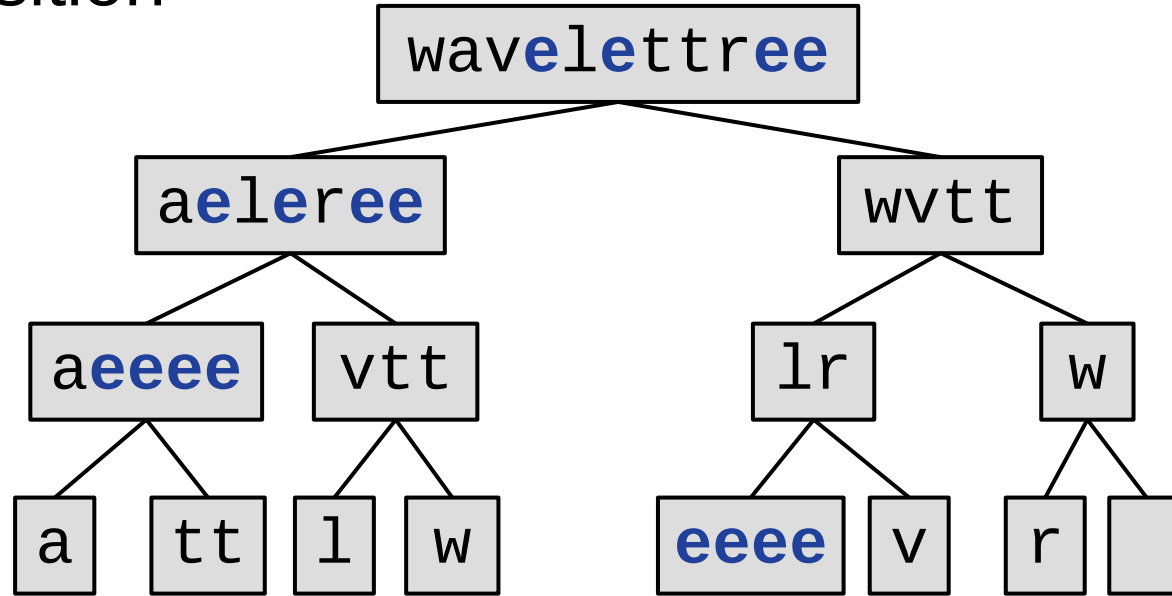# Locating in the WM
Node for position

Locating in the WM
Node for position
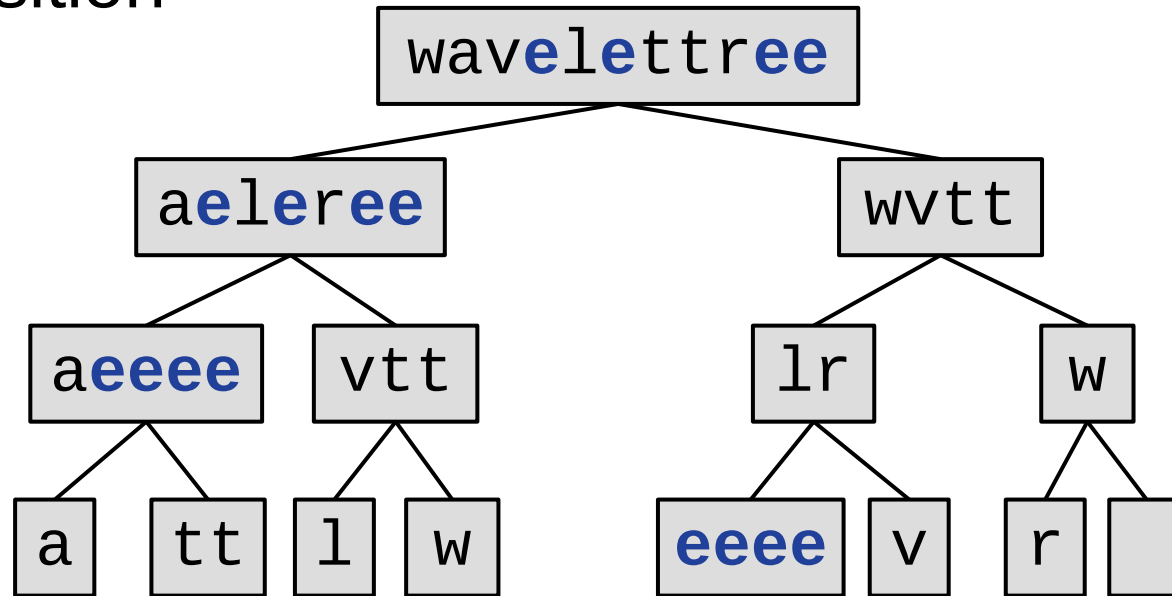
Locating in the WM
Node for position

on all levels: same symbol c = same node u

Locating in the WM
Node for position

on all levels: same symbol c = same node u

➤ if we know c when writing bit j, we only need its first occ. to find u

Locating in the WM
Node for position

➢ if we know c when writing bit j, we only need its first occ. to find u

Locating in the WM
Node for position

➢ if we know c when writing bit j, we only need its first occ. to find u

➢ in the WT, that's at $C[c]$ on the bottom level

Locating in the WM
Node for position

➢ if we know c when writing bit j, we only need its first occ. to find u

     ➢ in the WT, that's at $C[c]$ on the bottom level

     ➢ we can find v in the WT using B_c

Locating in the WM
Node for position

➢ if we know c when writing bit j, we only need its first occ. to find u

➢ in the WT, that's at $C[c]$ on the bottom level
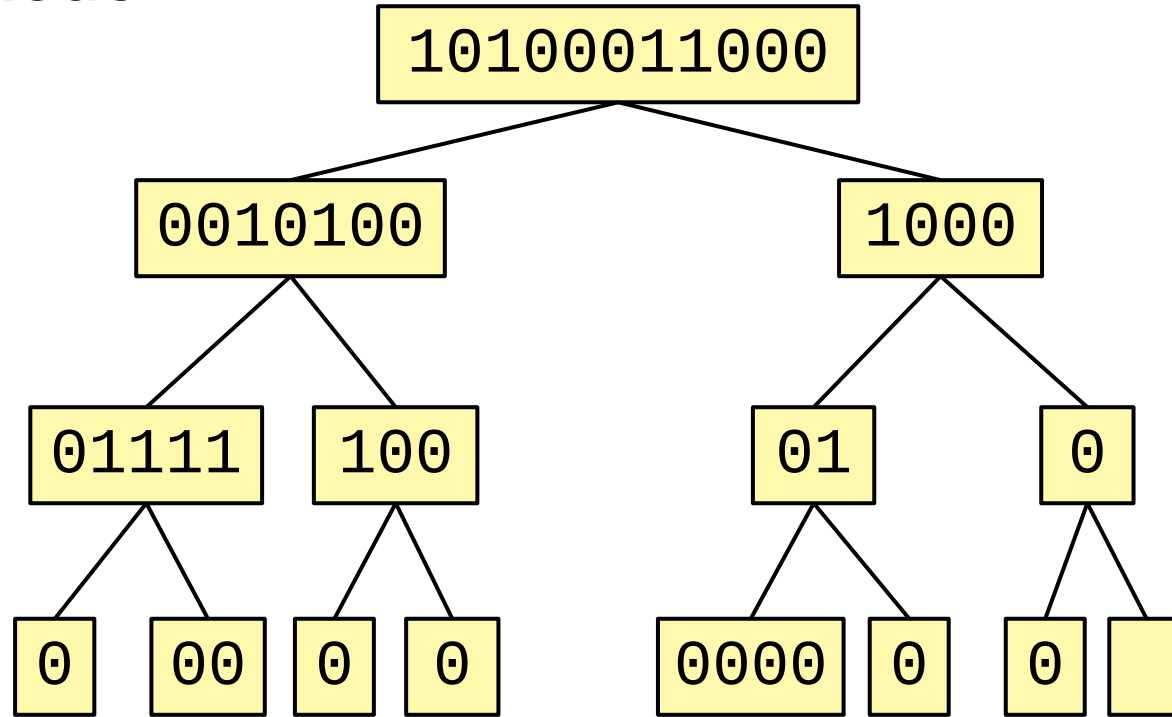
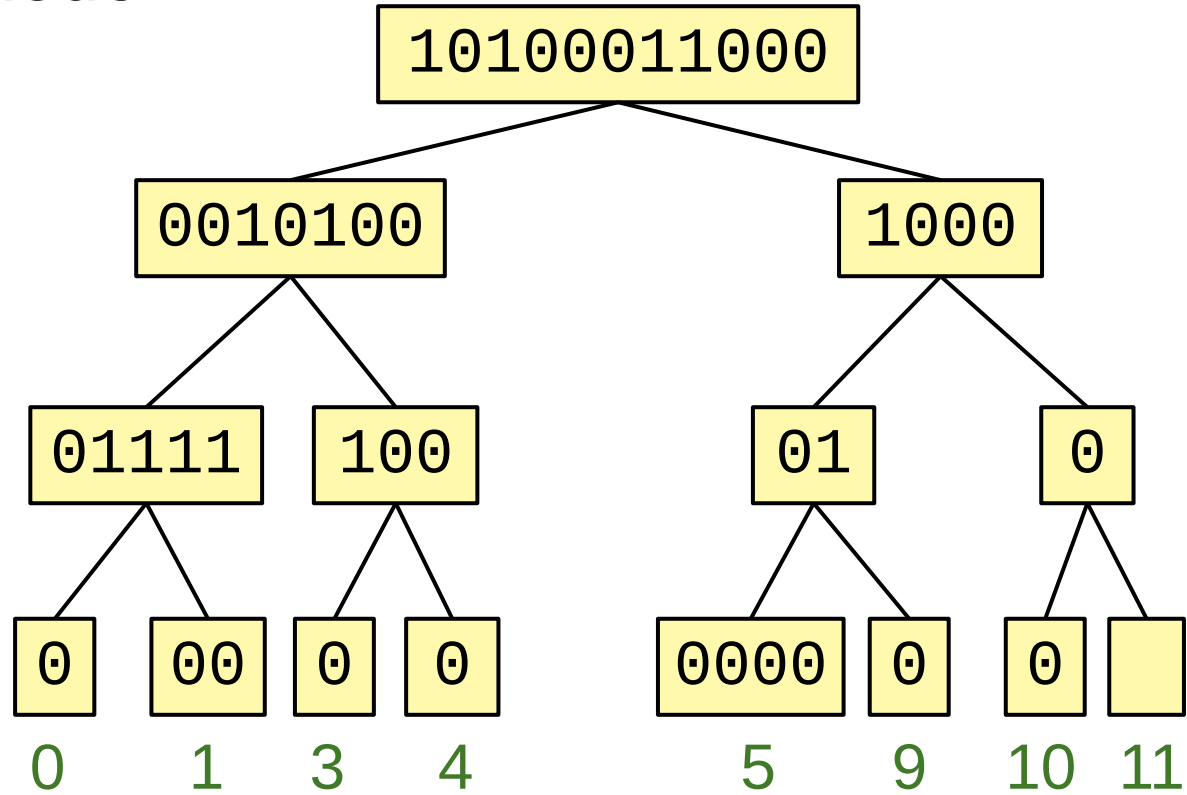➢ we can find v in the WT using B_c

➢ u is just v bit-reversed!
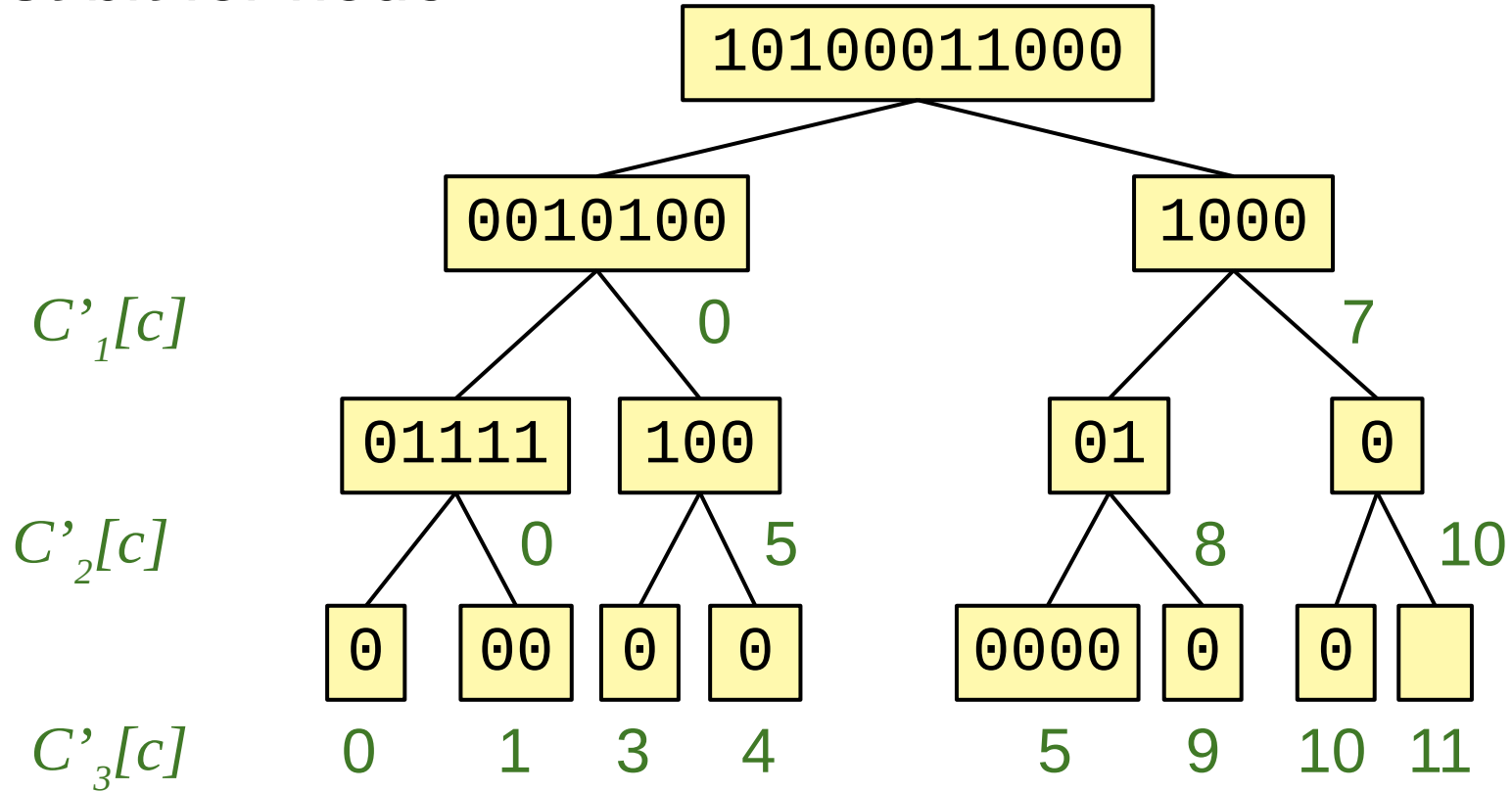
Locating in the WM
First bit for node

Locating in the WM
First bit for node

$C'_1[c]$

$C'_2[c]$

$C'_3[c]$

Locating in the WM
First bit for node

$$10100011000$$

0010100 1000

$C'_1[c]$ 0   7

01111 100 01 0

$C'_2[c]$ 0   5   8   10

0 00 0 0 0000 0 0

$C'_3[c]$ 0   1   3   4   5   9   10   11
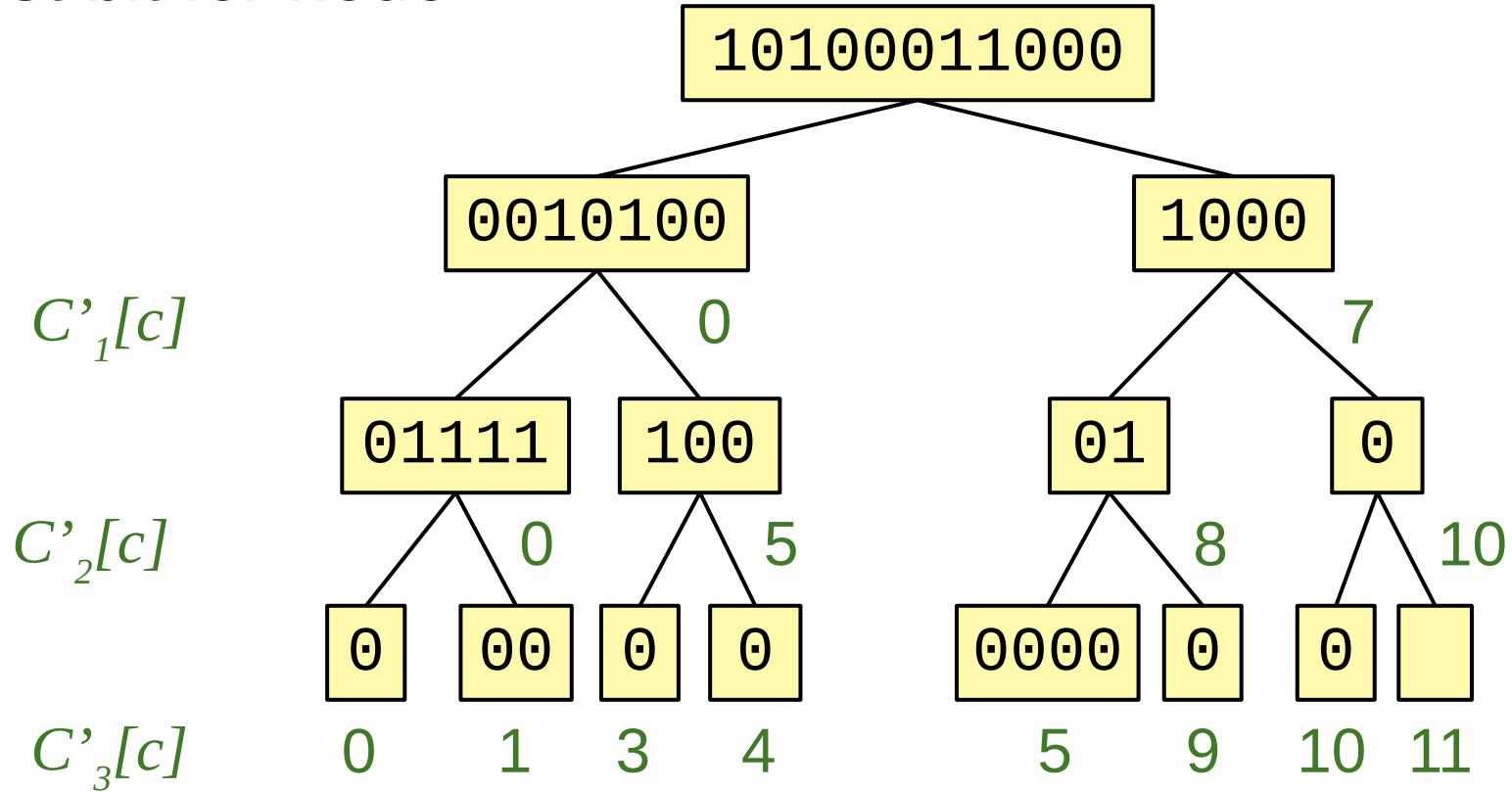
➢ at most $2\sigma - 1$ entries in an array C'

Locating in the WM
Summary

B_C with rank/select and C' required!

Locating in the WM
Summary

B_C **with rank/select and** C' **required!**

➢ Construct in time $O(n+\sigma)$

Locating in the WM
Summary

B_C with rank/select and C' required!

> Construct in time $O(n+\sigma)$
> Store in $< 2\sigma \lceil lg\ n \rceil + n + o(n)$ bits

Locating in the WM
Summary

B_C with rank/select and C' required!

- Construct in time $O(n+\sigma)$
- Store in $< 2\sigma \lceil lg\ n \rceil + n + o(n)$ bits
- Translate position in constant time

The *c* Oracle

The *c* Oracle

… breaks compatibility to a class of WM constructors!

The *c* Oracle

… breaks compatibility to a class of WM constructors!

(ex.: [Kaneta, SPIRE 2018] + our translator)

The *c* Oracle

… breaks compatibility to a class of WM constructors!

(ex.: [Kaneta, SPIRE 2018] + our translator)

It is possible without, but …

The *c* Oracle

… breaks compatibility to a class of WM constructors!

(ex.: [Kaneta, SPIRE 2018] + our translator)

It is possible without, but …

➢ binary search on *C'* to find *u*

… breaks constant-time access

The c Oracle

… breaks compatibility to a class of WM constructors!

(ex.: [Kaneta, SPIRE 2018] + our translator)

It is possible without, but …

➢ binary search on C' to find u

… breaks constant-time access

➢ bit vectors $B_{C'}$ on each level to mark node boundaries

… makes help structure larger than WT itself

Open Questions

Open Questions

- Can we improve when not using the c oracle?

Open Questions

- Can we improve when not using the c oracle?

- Can we compact C'?

Conclusions

- **Translation of WT to WM** using additional $O(n+\sigma)$ time, constant overhead and $n+o(n)$ bits of space

  - *Improves previous results!*

- **Translation of WM to WT** using additional $O(n+\sigma)$ time, constant overhead and $2\sigma\lceil lg\ n\rceil + n(1+o(1))$ bits of space

  - *Requires a symbol oracle!*

- **WT seems to contain more information than the WM**