Karlsruhe Institute of Technology

Text Indexing

Lecture 06: Wavelet Trees

Florian Kurpicz

PINGO

Karlsruhe Institute of Technology

https://pingo.scc.kit.edu/345678

Recap: Rank-Queries

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  $rank_1(i) = i - rank_0(i)$

Recap: Rank-Queries

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  $rank_1(i) = i - rank_0(i)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

Recap: Rank-Queries

Karlsruhe Institute of Technology

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  - $rank_1(i) = i - rank_0(i)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

Recap: Rank-Queries

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  $rank_1(i) = i - rank_0(i)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all length-s bit vectors, for every position i store number of 0s up to i
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

Recap: Rank-Queries

- for a bit vector of size n
- blocks of size $s = \lfloor \frac{\lg n}{2} \rfloor$
- super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
- information for 0s or 1s enough
  $rank_1(i) = i - rank_0(i)$

- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ bits of space

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
- $n/s \cdot \lg s' = O(\frac{n \lg \lg n}{\lg n}) = o(n)$ bits of space

- for all length-s bit vectors, for every position i store number of 0s up to i
- $2^{\frac{\lg n}{2}} \cdot s \cdot \lg s = O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space

- query in $O(1)$ time using three subqueries
  - one in super-block
  - one in block
  - one for remaining bitvector smaller than s

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lg n)$ this suffice

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

Select in $o(n)$ Space and $O(1)$ Time

- *select*$_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

Select in $o(n)$ Space and $O(1)$ Time

Karlsruhe Institute of Technology

- $select_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block

Select in $o(n)$ Space and $O(1)$ Time

Karlsruhe Institute of Technology

- $select_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ ℹ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
- $|B_{\lfloor i/b \rfloor}| \geq \lg^4 n$: store answers naively
  - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
  - there are at most $O(n/\lg^4 n)$ such blocks
  - total $O(n/\lg n) = o(n)$ bits of space

Select in $o(n)$ Space and $O(1)$ Time

- $select_0$ in a bit vector of size n that contains k zeros
- naive solutions
  - scan bit vector: $O(n)$ time and no space overhead
  - store k solutions in $S[1..k]$ and $select_0(i) = S[i]$ if $k \in O(n/lgn)$ this suffice

- better: k/b variable-sized super-blocks B_i, such that super-block contains $b = \lg^2 n$ zeros
- $select_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + select_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b))$

- storing all possible results for the (prefix) sum
- $O((k \lg n)/b) = o(n)$ bits of space

- select on block depends on size of block
- $|B_{\lfloor i/b \rfloor}| \geq \lg^4 n$: store answers naively
  - requires $O(b \lg n) = O(\lg^3 n)$ bits of space
  - there are at most $O(n/\lg^4 n)$ such blocks
  - total $O(n/\lg n) = o(n)$ bits of space
- $|B_{\lfloor i/b \rfloor}| < \lg^4 n$: divide super-block into blocks
  - same idea: variable-sized blocks containing $b' = \sqrt{\lg n}$ zeros
  - (prefix) sum $O((k \lg \lg n)/b') = o(n)$ bits
  - if size $\geq \lg n$ store all answers
  - if size $< \lg n$ store lookup table

Rank- and Select-Queries on Bit Vectors

Lemma: Binary Rank- and Select-Queries

Given a bit vector of size n, there exists data structures that can be computed in time $O(n)$ of size $o(n)$ bits that can answer rank and select queries on the bit vector in $O(1)$ time

Preliminaries

Definition: Bit Representation

Given a text T over an alphabet of size σ, each character can be represented using $\lceil \lg \sigma \rceil$ bits.

- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

0	1	2	3	4	5	6	7	
(0	(0	(0	(0	(1	(1	(1	(1	MSB
0	0	1	1	0	0	1	1	
0)$_2$	1)$_2$	0)$_2$	1)$_2$	0)$_2$	1)$_2$	0)$_2$	1)$_2$	LSB

Preliminaries

Definition: Bit Representation

Given a text T over an alphabet of size σ, each character can be represented using $\lceil \lg \sigma \rceil$ bits.

- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

0	1	2	3	4	5	6	7	
$(0$	$(0$	$(0$	$(0$	$(1$	$(1$	$(1$	$(1$	MSB
0	0	1	1	0	0	1	1	
$0)_2$	$1)_2$	$0)_2$	$1)_2$	$0)_2$	$1)_2$	$0)_2$	$1)_2$	LSB

- for simplicity characters are integers
- bit representation is integer in binary

Preliminaries

Definition: Bit Representation

Given a text T over an alphabet of size σ, each character can be represented using $\lceil \lg \sigma \rceil$ bits.

- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

	0	1	2	3	4	5	6	7	
	(0	(0	(0	(0	(1	(1	(1	(1	MSB
	0	0	1	1	0	0	1	1	
	0 $)_2$	1 $)_2$	0 $)_2$	1 $)_2$	0 $)_2$	1 $)_2$	0 $)_2$	1 $)_2$	LSB

- for simplicity characters are integers
- bit representation is integer in binary

Definition: Bit Prefix

A bit prefix of length k are the k MSBs of a characters bit representation

Wavelet Trees [GGV03] (1/2)

Karlsruhe Institute of Technology

Definition: Wavelet Tree

Given a text T of length n over an alphabet $\Sigma = [1, \sigma]$, a **wavelet tree** is a binary tree, where

- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child
- represent characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r]$
- a node is a leaf if $\ell + 2 \geq r$
- characters are represented using a bit vector
- an entry is 1 if the character is represented in the right child and 0 otherwise

Wavelet Trees [GGV03] (1/2)

Definition: Wavelet Tree

Given a text T of length n over an alphabet $\Sigma = [1, \sigma]$, a **wavelet tree** is a binary tree, where

- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child
- represent characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r]$
- a node is a leaf if $\ell + 2 \geq r$
- characters are represented using a bit vector
- an entry is 1 if the character is represented in the right child and 0 otherwise

Definition: Level-wise Wavelet Tree

A wavelet tree, where all bit vectors on the same depth in the tree are concatenated is called level-wise wavelet tree

Wavelet Trees [GGV03] (1/2)

Definition: Wavelet Tree

Given a text T of length n over an alphabet $\Sigma = [1, \sigma]$, a **wavelet tree** is a binary tree, where

- each node represents characters in $[\ell, r] \subseteq [1, \sigma]$,
- if a node represents characters in $[\ell, r]$, then its left and right child
- represent characters in $[\ell, (\ell + r)/2)$ and $[(\ell + r)/2, r]$
- a node is a leaf if $\ell + 2 \geq r$
- characters are represented using a bit vector
- an entry is 1 if the character is represented in the right child and 0 otherwise

Definition: Level-wise Wavelet Tree

A wavelet tree, where all bit vectors on the same depth in the tree are concatenated is called level-wise wavelet tree

- in practice, level-wise wavelet trees have less overhead
- navigation still easy

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

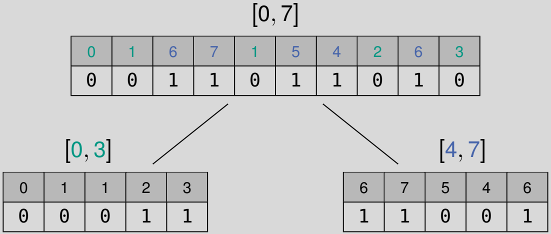
Wavelet Trees (2/2)

$[0, 7]$

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

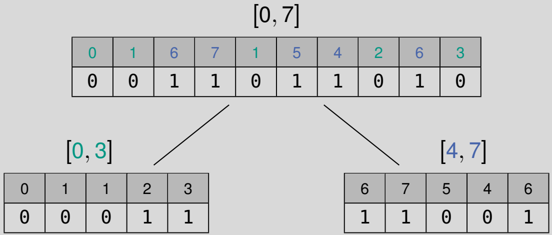
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet Trees (2/2)



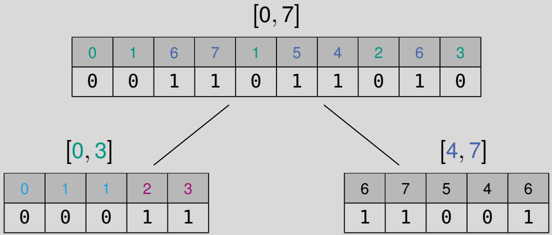
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet Trees (2/2)



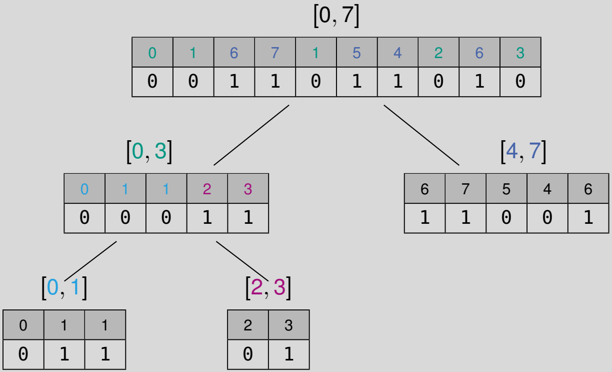
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet Trees (2/2)



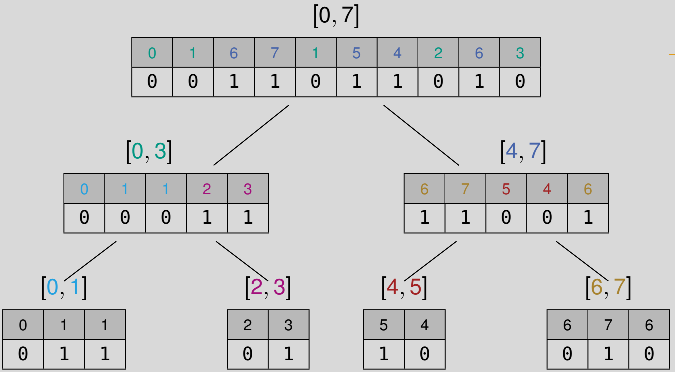
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

Wavelet Trees (2/2)



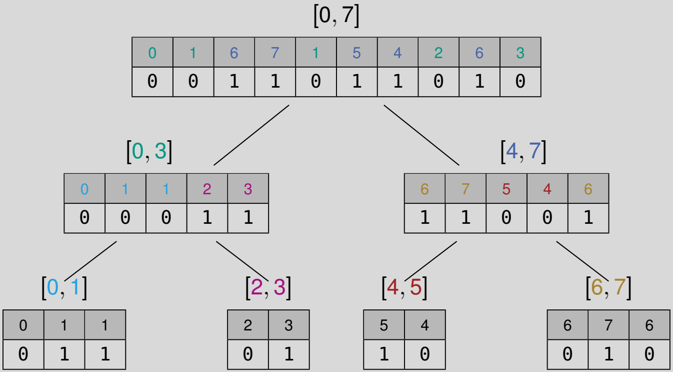
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

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)



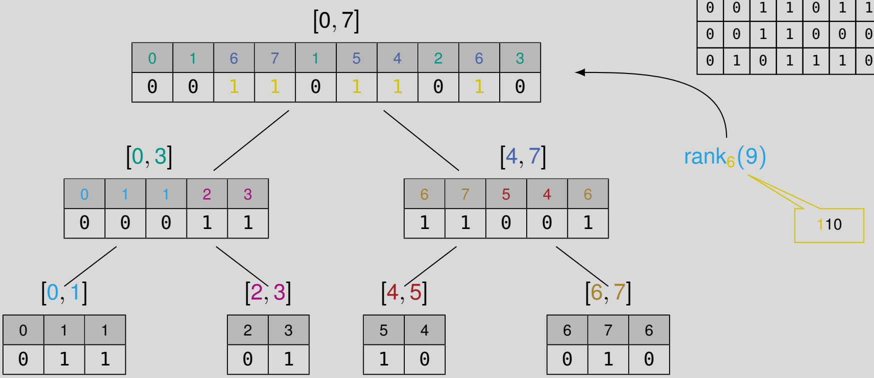
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

$\mathrm{rank}_6(9)$

110

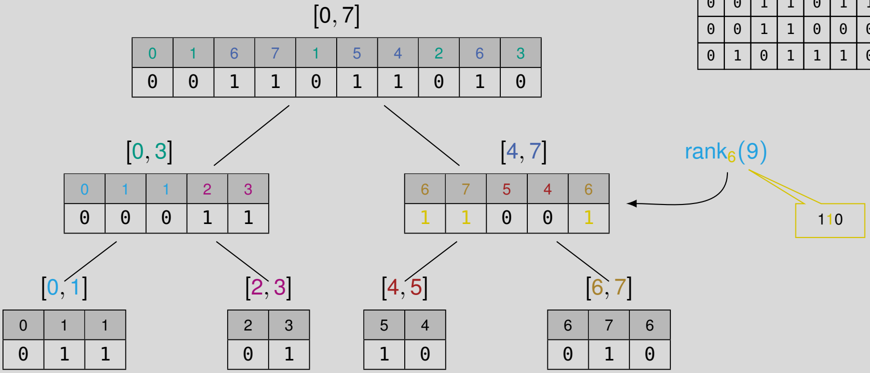
Wavelet Trees (2/2)

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

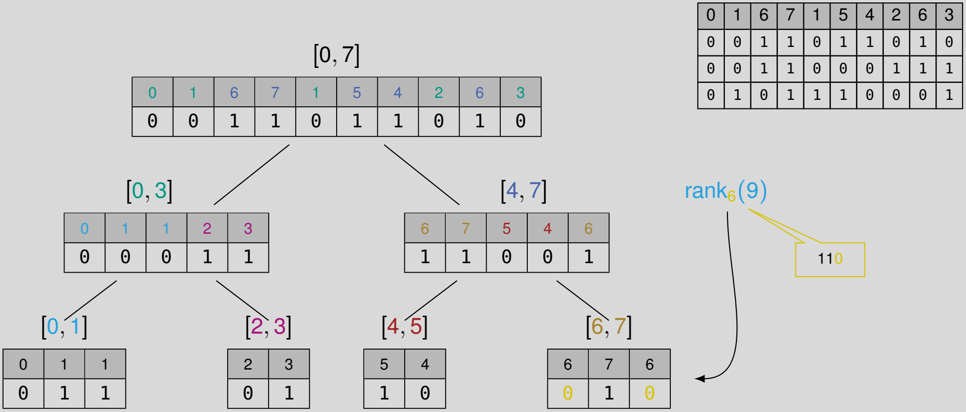


Wavelet Trees (2/2)

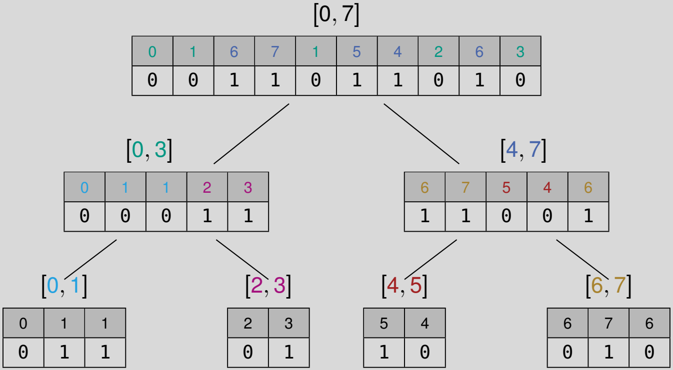
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1



Wavelet Trees (2/2)

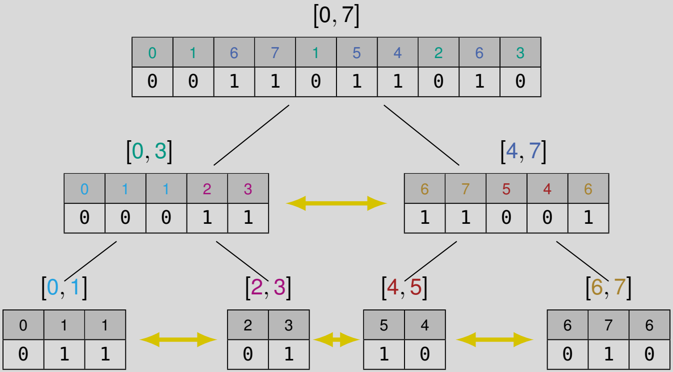


Wavelet Trees (2/2)



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)



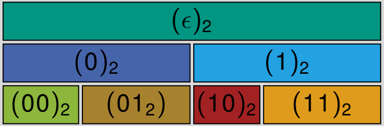
0	1	6	7	1	5	4	2	6	3
0	0	1	1	0	1	1	0	1	0
0	0	1	1	0	0	0	1	1	1
0	1	0	1	1	1	0	0	0	1

The Intervals of a Wavelet Tree

- in each node, all represented characters share a bit prefix
- on depth ℓ the longest common bit prefix has length $\ell - 1$
- the bit prefixes form intervals

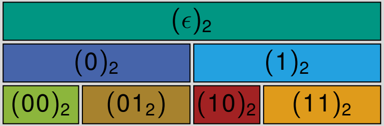
The Intervals of a Wavelet Tree

- in each node, all represented characters share a bit prefix
- on depth ℓ the longest common bit prefix has length $\ell - 1$
- the bit prefixes form intervals



The Intervals of a Wavelet Tree

- in each node, all represented characters share a bit prefix
- on depth ℓ the longest common bit prefix has length $\ell - 1$
- the bit prefixes form intervals



- finding characters in the wavelet tree requires finding the correct interval
- finding the position of a character requires finding the position in the last interval

Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

Rank-Queries

- use rank queries on bit vectors
- at depth ℓ as for ℓ-th MSB
- follow through tree according to bit

- as seen on a previous slide

Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

Rank-Queries

- use rank queries on bit vectors
- at depth ℓ as for ℓ-th MSB
- follow through tree according to bit

- as seen on a previous slide

Select-Queries

- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root

- requires up and down traversal of the wavelet tree
- see example on the board

Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

Rank-Queries

- use rank queries on bit vectors
- at depth ℓ as for ℓ-th MSB
- follow through tree according to bit

- as seen on a previous slide

Select-Queries

- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root

- requires up and down traversal of the wavelet tree
- see example on the board

Access-Queries

- follow bits through the wavelet tree
- return read bits

- same as rank but returning bit pattern instead of final rank
- see example on the board

Rank-, Select-, and Access-Queries in Wavelet Trees (1/2)

Rank-Queries

- use rank queries on bit vectors
- at depth ℓ as for ℓ-th MSB
- follow through tree according to bit

- as seen on a previous slide

- **PINGO** what is the query time of rank queries in wavelet trees?

Select-Queries

- identify leaf containing character
- select corresponding occurrence in leaf
- backtrack position up the tree to the root

- requires up and down traversal of the wavelet tree
- see example on the board

Access-Queries

- follow bits through the wavelet tree
- return read bits

- same as rank but returning bit pattern instead of final rank
- see example on the board

Rank-, Select-, and Access-Queries in Wavelet Trees (2/2)

Lemma: Query Times Wavelet Tree

Given a text T over an alphabet of size σ, the wavelet tree of the text can answer *rank*, *select*, and *access* queries in $O(\lg \sigma)$ time

Proof (Sketch)

All queries require

- just a constant number of rank and select queries on the bit vectors and
- at most one traversals from the root of the tree to a leaf and
- one traversal from a leaf to the root of the tree

Bit Reversal Permutation

- given a bit representation of a character α
- *reverse*(α) reverses the bits
- the MSB becomes the least significant bit

Definition: Bit-Reversal Permutation

The **bit-reversal permutation** ρ_k is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = \mathit{reverse}(i)$$

for $i \in [0, 2^k)$

Bit Reversal Permutation

- given a bit representation of a character α
- $reverse(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

- $\rho_2 = (0, 2, 1, 3) = ((00)_2, (10)_2, (01)_2, (11)_2)$
- $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k - 1),$
  $2\rho_k(0) + 1, \ldots, 2\rho_k(2^k - 1) + 1)$

Definition: Bit-Reversal Permutation

The **bit-reversal permutation** ρ_k is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = reverse(i)$$

for $i \in [0, 2^k)$

Bit Reversal Permutation

- given a bit representation of a character α
- $\mathit{reverse}(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

Definition: Bit-Reversal Permutation

The **bit-reversal permutation** ρ_k is a permutation of the numbers $[0, 2^k)$ with

$$\rho_k(i) = \mathit{reverse}(i)$$

for $i \in [0, 2^k)$

- $\rho_2 = (0, 2, 1, 3) = ((00)_2, (10)_2, (01)_2, (11)_2)$
- $\rho_{k+1} = (2\rho_k(0), \ldots, 2\rho_k(2^k - 1), 2\rho_k(0) + 1, \ldots, 2\rho_k(2^k - 1) + 1)$

- same intervals as a wavelet tree
- used in the wavelet matrix

Alternative Representation

Karlsruhe Institute of Technology

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level ℹ the intervals discussed before still exist

Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level ℹ the intervals discussed before still exist

Definition: Wavelet Matrix [CNP15]

Given a text T of length n over an alphabet of size σ a wavelet matrix consists of

- bit vectors BV_ℓ for $\ell \in [1, \lceil \lg \sigma \rceil]$ of size n and
- an array $Z[1..\lceil \lg \sigma \rceil]$

Such that

- $Z[\ell]$ contains the number of zero bits in BV_ℓ
- BV_1 contains all MSBs in text order
- BV_ℓ contains the ℓ-th MSB the character at position i in $BV_{\ell-1}$ at position
  - $rank_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell - 1] + rank_1(i)$ if $BV_{\ell-1} = 1$

Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level ℹ the intervals discussed before still exist

- better suited for large alphabets
- seemingly less structure
- retaining all important properties

Definition: Wavelet Matrix [CNP15]

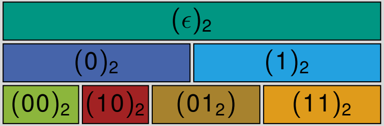
Given a text T of length n over an alphabet of size σ a wavelet matrix consists of

- bit vectors BV_ℓ for $\ell \in [1, \lceil \lg \sigma \rceil]$ of size n and
- an array $Z[1..\lceil \lg \sigma \rceil]$

Such that

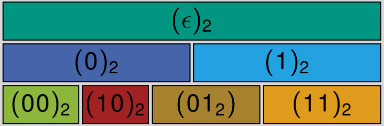
- $Z[\ell]$ contains the number of zero bits in BV_ℓ
- BV_1 contains all MSBs in text order
- BV_ℓ contains the ℓ-th MSB the character at position i in $BV_{\ell-1}$ at position
  - $rank_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell-1] + rank_1(i)$ if $BV_{\ell-1} = 1$

Intervals of a Wavelet Matrix

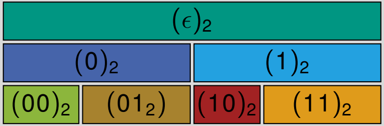


- a wavelet matrix has the same intervals a wavelet tree has
- intervals not bounded by parent ℹ no tree structure

Intervals of a Wavelet Matrix

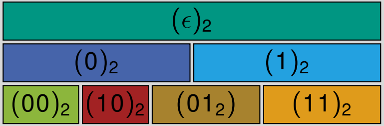


- a wavelet matrix has the same intervals a wavelet tree has
- intervals not bounded by parent ℹ no tree structure

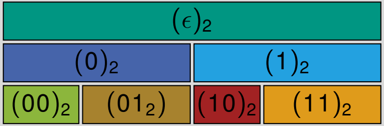


- intervals of a wavelet tree (for comparison)

Intervals of a Wavelet Matrix



- a wavelet matrix has the same intervals a wavelet tree has
- intervals not bounded by parent ⓘ no tree structure



- intervals of a wavelet tree (for comparison)

- **PINGO** is answering queries with a wavelet matrix as simple as with a wavelet tree?

Example Wavelet Tree and Wavelet Matrix

	0	1	3	7	1	5	4	2	6	3
BV_0	0	0	0	1	0	1	1	0	1	0
	0	1	3	1	2	3	7	5	4	6
BV_1	0	0	1	0	1	1	1	0	0	1
	0	1	1	3	2	3	5	4	7	6
BV_2	0	1	1	1	0	1	1	0	1	0

	0	1	3	7	1	5	4	2	6	3
BV_0	0	0	0	1	0	1	1	0	1	0
	0	1	3	1	2	3	7	5	4	6
BV_1	0	0	1	0	1	1	1	0	0	1
	0	1	1	5	4	3	2	3	7	6
BV_2	0	1	1	1	0	1	0	1	1	0

$Z[0] = 6$ $\quad Z[1] = 5$ $\quad Z[2] = 4$

- queries on the wavelet matrix work similar
- example on the board

Naive Wavelet Tree and Wavelet Matrix Construction (1/2)

	0	1	3	7	1	5	4	2	6	3
BV_0	0	0	0	1	0	1	1	0	1	0
	0	1	3	1	2	3	7	5	4	6
BV_1	0	0	1	0	1	1	1	0	0	1
	0	1	1	3	2	3	5	4	7	6
BV_2	0	1	1	1	0	1	1	0	1	0

Wavelet Tree

- first level are MSBs of characters of text
- for each level $\ell > 1$
  - stably sort text using Radix sort by bit prefixes of length $\ell - 1$
  - take ℓ-th MSB of sorted sequence
  - sorted sequence is new text

Naive Wavelet Tree and Wavelet Matrix Construction (1/2)

	0	1	3	7	1	5	4	2	6	3
BV_0	0	0	0	1	0	1	1	0	1	0
	0	1	3	1	2	3	7	5	4	6
BV_1	0	0	1	0	1	1	1	0	0	1
	0	1	1	3	2	3	5	4	7	6
BV_2	0	1	1	1	0	1	1	0	1	0

Wavelet Tree

- first level are MSBs of characters of text
- for each level $\ell > 1$
  - stably sort text using Radix sort by bit prefixes of length $\ell - 1$
  - take ℓ-th MSB of sorted sequence
  - sorted sequence is new text

	0	1	3	7	1	5	4	2	6	3
BV_0	0	0	0	1	0	1	1	0	1	0
	0	1	3	1	2	3	7	5	4	6
BV_1	0	0	1	0	1	1	1	0	0	1
	0	1	1	5	4	3	2	3	7	6
BV_2	0	1	1	1	0	1	0	1	1	0

$Z[0] = 6$ $Z[1] = 5$ $Z[2] = 4$

Wavelet Matrix

- first level are MSBs of characters of text
- for each level $\ell > 1$
  - stably sort text by $\ell - 1$ MSB
  - take ℓ-th MSB of sorted sequence
  - sorted sequence is new text

Wavelet Tree and Wavelet Matrix Construction (2/2)

- to make both fully functional bit vectors are augmented with binary rank and select support

Lemma: Running Time and Memory Requirements Wavelet Tree and Wavelet Matrix

Given a text T over an alphabet of size σ, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil \lg \sigma \rceil$ bits of space and can be constructed in $O(n \lg \sigma)$ time

Wavelet Tree and Wavelet Matrix Construction (2/2)

Karlsruhe Institute of Technology

- to make both fully functional bit vectors are augmented with binary rank and select support

Lemma: Running Time and Memory Requirements Wavelet Tree and Wavelet Matrix

Given a text T over an alphabet of size σ, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil \lg \sigma \rceil$ bits of space and can be constructed in $O(n \lg \sigma)$ time

- **PINGO** is there a asymptotically faster construction method?

Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every τ-th level is a big level
- big levels contain enough information to compute small levels below
- small levels computed by splitting big levels
- $O(b/\lg n)$ characters at a time with $b = o(\lg n)$
- sketch on board

Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every τ-th level is a big level
- big levels contain enough information to compute small levels below
- small levels computed by splitting big levels
- $O(b/\lg n)$ characters at a time with $b = o(\lg n)$
- sketch on board

Lemma: Better Wavelet Tree Construction

Given a text T over an alphabet of size σ, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil \lg \sigma \rceil$ bits of space and can be constructed in $O(n \lg \sigma / \sqrt{\lg n})$ time

Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every τ-th level is a big level
- big levels contain enough information to compute small levels below
- small levels computed by splitting big levels
- $O(b/\lg n)$ characters at a time with $b = o(\lg n)$
- sketch on board

Lemma: Better Wavelet Tree Construction

Given a text T over an alphabet of size σ, the wavelet tree and wavelet matrix require $(1 + o(1))n\lceil \lg \sigma \rceil$ bits of space and can be constructed in $O(n \lg \sigma / \sqrt{\lg n})$ time

- can be implemented using AVX/SSE instructions [Din+23; Kan18]

Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes

Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes

Huffman Codes (Recap)

- idea is to create a binary tree
- each character α is a leaf and has weight $\mathit{Hist}[\alpha]$
- create node for two nodes without parent with smallest weight
- give new node total weight of children
- repeat until only one node without parent remains
- label edges:
  - left edge: 0
  - right edge: 1
- path to children gives code for character

Huffman-shaped Wavelet Trees

- wavelet trees can be compressed
- more precise: the text can be compressed
- use Huffman codes
- wavelet trees cannot handle holes
- use canonical Huffman codes

Canonical Huffman Codes (Recap)

- start with Huffman codes, code word 0, and length 1
- to get canonical code for current length, then add 1 to code word
- to update length add 1 and append required amount of zeros to code word

Huffman Codes (Recap)

- idea is to create a binary tree
- each character α is a leaf and has weight $\mathit{Hist}[\alpha]$
- create node for two nodes without parent with smallest weight
- give new node total weight of children
- repeat until only one node without parent remains
- label edges:
  - left edge: 0
  - right edge: 1
- path to children gives code for character

Huffman-shaped Wavelet Trees

α	$hc(\alpha)$	$chc(\alpha)$
1	$(11)_2$	$(11)_2$
3	$(01)_2$	$(10)_2$
6	$(100)_2$	$(011)_2$
7	$(101)_2$	$(010)_2$
0	$(0000)_2$	$(0011)_2$
2	$(0001)_2$	$(0010)_2$
4	$(0010)_2$	$(0001)_2$
5	$(0011)_2$	$(0000)_2$

- Huffman codes (hc)
- canonical Huffman codes (chc) that are bit-wise negated

Huffman-shaped Wavelet Trees

α	$hc(\alpha)$	$chc(\alpha)$
1	$(11)_2$	$(11)_2$
3	$(01)_2$	$(10)_2$
6	$(100)_2$	$(011)_2$
7	$(101)_2$	$(010)_2$
0	$(0000)_2$	$(0011)_2$
2	$(0001)_2$	$(0010)_2$
4	$(0010)_2$	$(0001)_2$
5	$(0011)_2$	$(0000)_2$

- Huffman codes (hc)
- canonical Huffman codes (chc) that are bit-wise negated

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

- intervals are only missing to the right (white space)
- no holes allow for easy querying

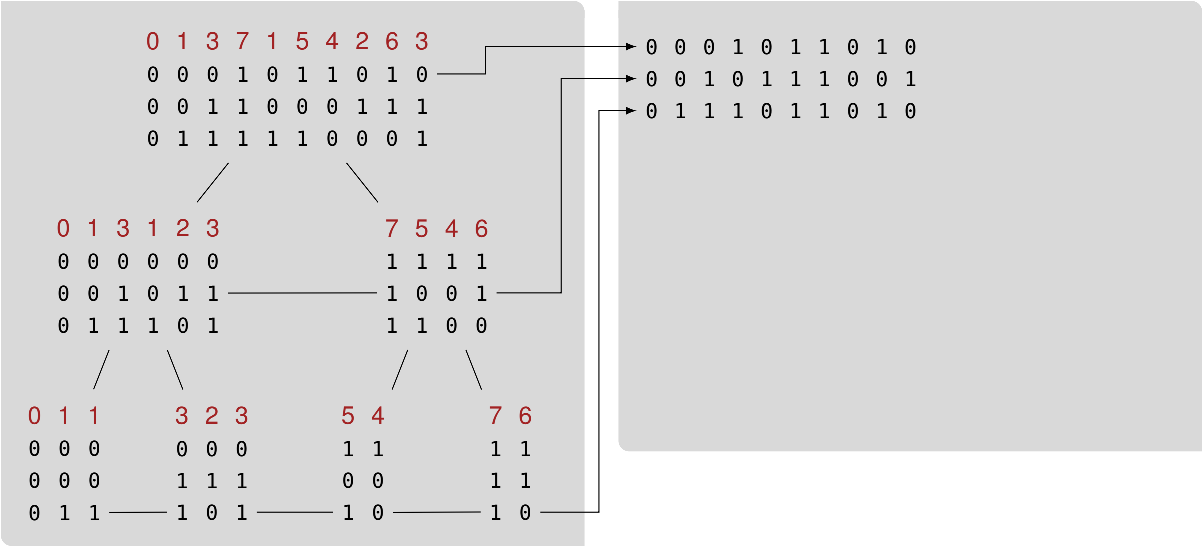
Practical Sequential Wavelet Tree Construction

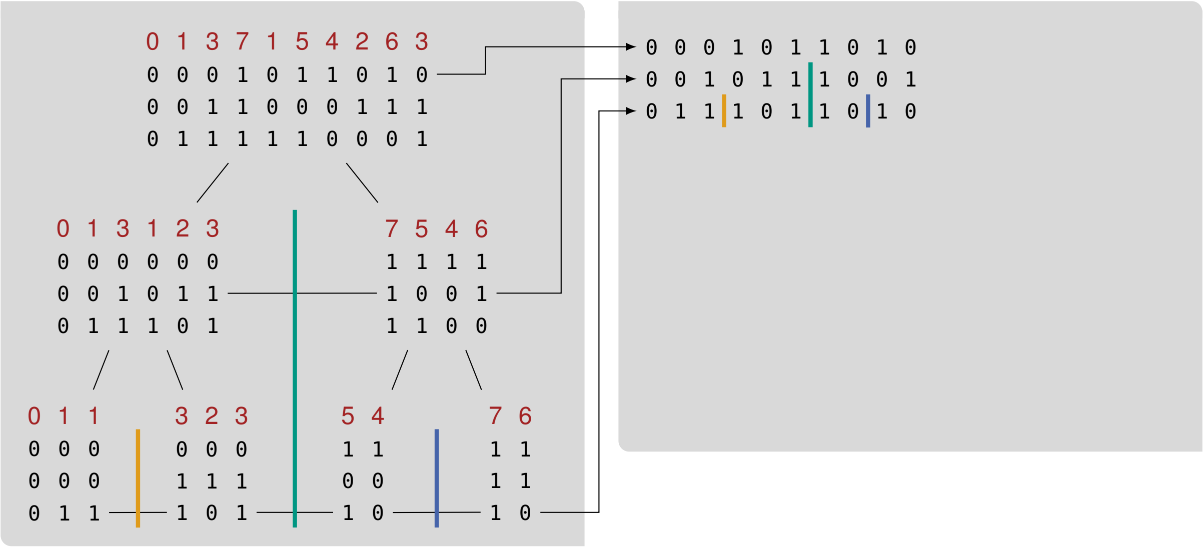
Karlsruhe Institute of Technology

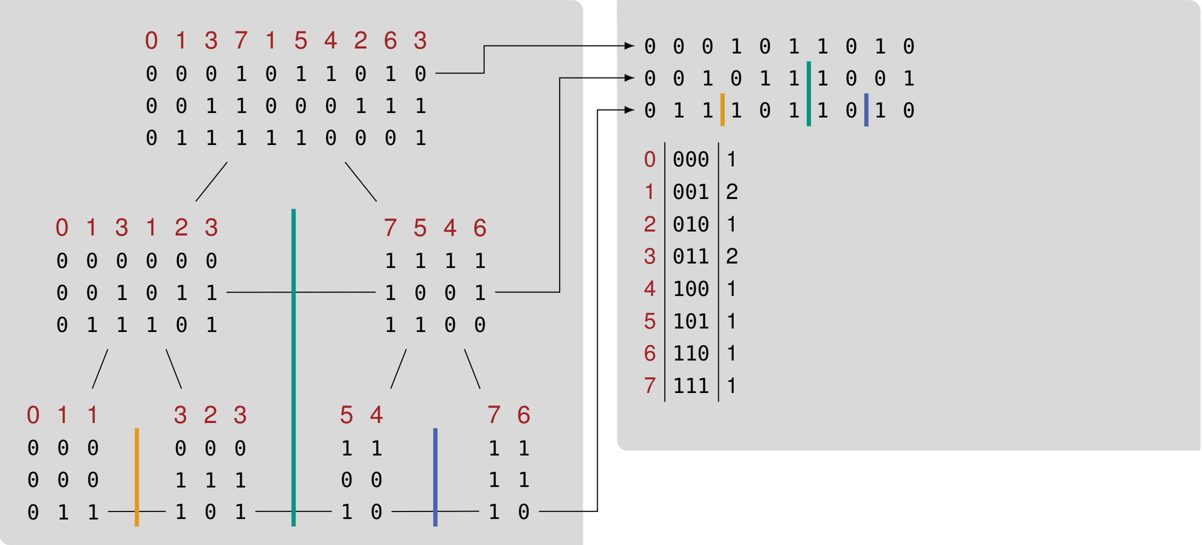
Bottom-Up Construction [FKL18]

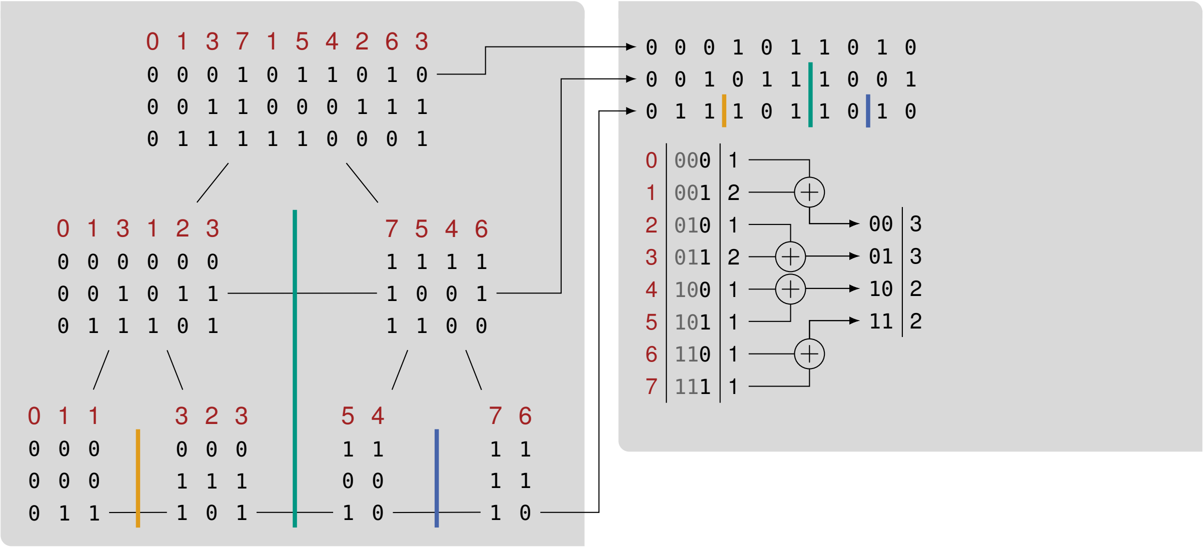
- scan the text and create histogram
- while scanning compute first level
- use histogram to compute borders of intervals
- scan text again and fill bit vectors

- example on the next slide

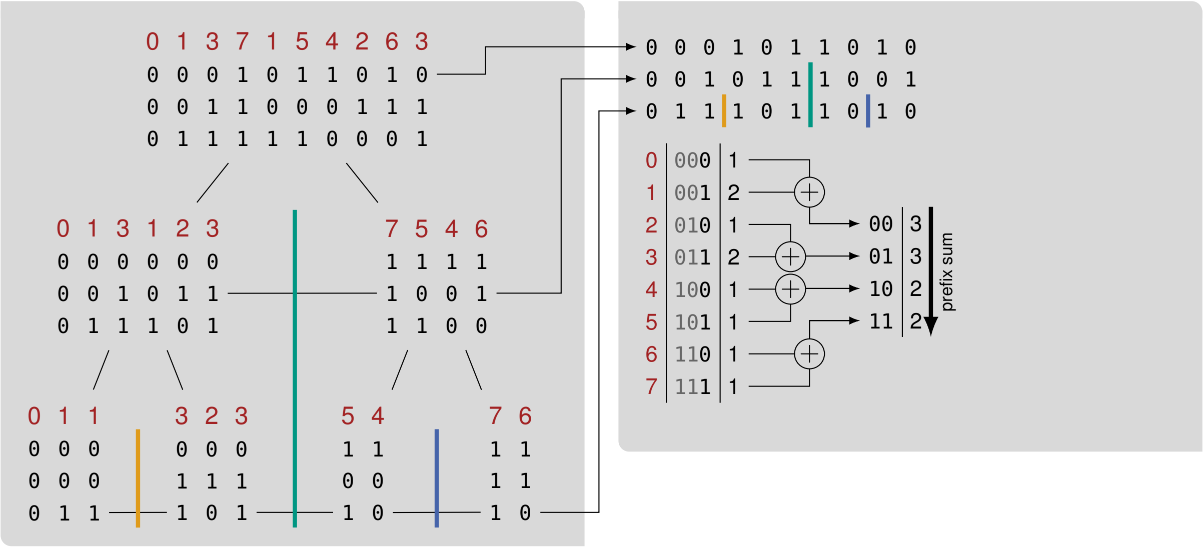

0 1 3 7 1 5 4 2 6 3
0 0 0 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 1 1 1 1 0 0 0 1
0 1 3 1 2 3
0 0 0 0 0 0
0 0 1 0 1 1
0 1 1 1 0 1
7 5 4 6
1 1 1 1
1 0 0 1
1 1 0 0
0 1 1
0 0 0
0 0 0
0 1 1
3 2 3
0 0 0
1 1 1
1 0 1
5 4
1 1
0 0
1 0
7 6
1 1
1 1
1 0
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0

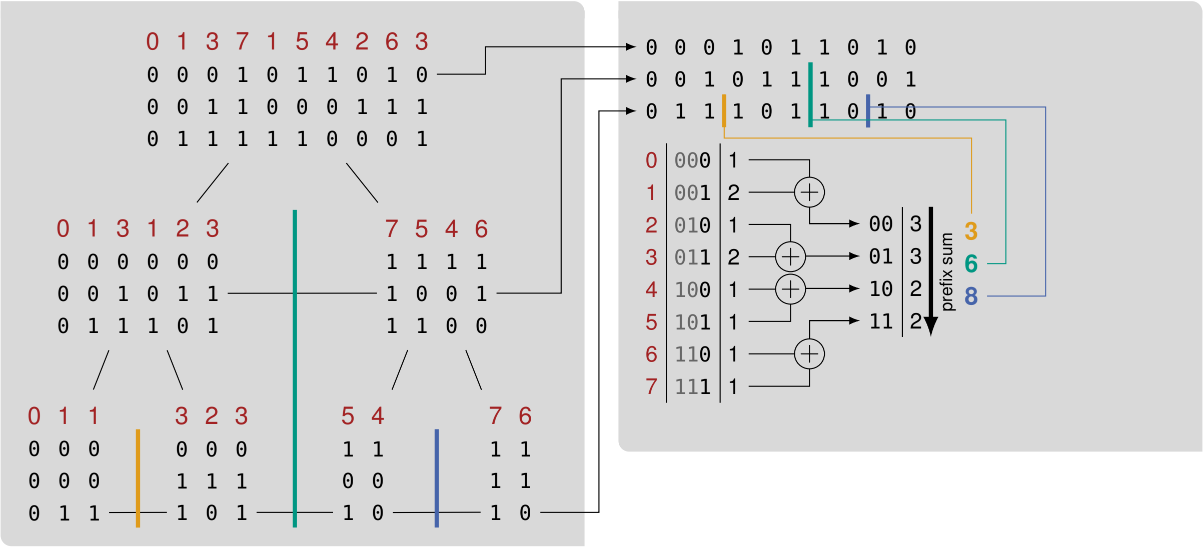

0 1 3 7 1 5 4 2 6 3
0 0 0 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 1 1 1 1 0 0 0 1
0 1 3 1 2 3
0 0 0 0 0 0
0 0 1 0 1 1
0 1 1 1 0 1
7 5 4 6
1 1 1 1
1 0 0 1
1 1 0 0
0 1 1
0 0 0
0 0 0
0 1 1
3 2 3
0 0 0
1 1 1
1 0 1
5 4
1 1
0 0
1 0
7 6
1 1
1 1
1 0
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0


0 1 3 7 1 5 4 2 6 3
0 0 0 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 1 1 1 1 0 0 0 1
0 1 3 1 2 3
0 0 0 0 0 0
0 0 1 0 1 1
0 1 1 1 0 1
7 5 4 6
1 1 1 1
1 0 0 1
1 1 0 0
0 1 1
0 0 0
0 0 0
0 1 1
3 2 3
0 0 0
1 1 1
1 0 1
5 4
1 1
0 0
1 0
7 6
1 1
1 1
1 0
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0
0 000 1
1 001 2
2 010 1
3 011 2
4 100 1
5 101 1
6 110 1
7 111 1



0 1 3 7 1 5 4 2 6 3
0 0 0 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 1 1 1 1 0 0 0 1
0 1 3 1 2 3
0 0 0 0 0 0
0 0 1 0 1 1
0 1 1 1 0 1
7 5 4 6
1 1 1 1
1 0 0 1
1 1 0 0
0 1 1
0 0 0
0 0 0
0 1 1
3 2 3
0 0 0
1 1 1
1 0 1
5 4
1 1
0 0
1 0
7 6
1 1
1 1
1 0
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0
0 000 1
1 001 2
2 010 1
3 011 2
4 100 1
5 101 1
6 110 1
7 111 1
00 3
01 3
10 2
11 2


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

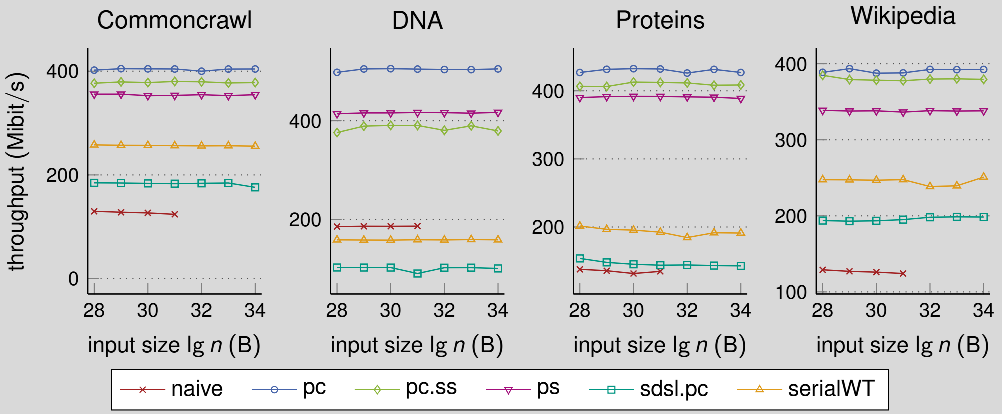

0 1 3 7 1 5 4 2 6 3
0 0 0 1 0 1 1 0 1 0
0 0 1 1 0 0 0 1 1 1
0 1 1 1 1 1 0 0 0 1
0 1 3 1 2 3
0 0 0 0 0 0
0 0 1 0 1 1
0 1 1 1 0 1
7 5 4 6
1 1 1 1
1 0 0 1
1 1 0 0
0 1 1
0 0 0
0 0 0
0 1 1
3 2 3
0 0 0
1 1 1
1 0 1
5 4
1 1
0 0
1 0
7 6
1 1
1 1
1 0
0 0 0 1 0 1 1 0 1 0
0 0 1 0 1 1 1 0 0 1
0 1 1 1 0 1 1 0 1 0
0 000 1
1 001 2
2 010 1
3 011 2
4 100 1
5 101 1
6 110 1
7 111 1
00 3
01 3
10 2
11 2
prefix sum
3
6
8

Experimental Setup

- 64 GB RAM
- two Intel Xeon E5-2640v4 CPUs (10 cores at 2.4 GHz base frequency, 3.4 GHz maximum turbo frequency, and cache sizes: 32 KB L1D and L1I, 256 KB L2, 25.6 MB L3)

- same texts as in chapter 04
- results are average of 5 runs

Experiments: Sequential Wavelet Tree Construction



Experiments: Vectorized Wavelet Tree Construction [Din+23]

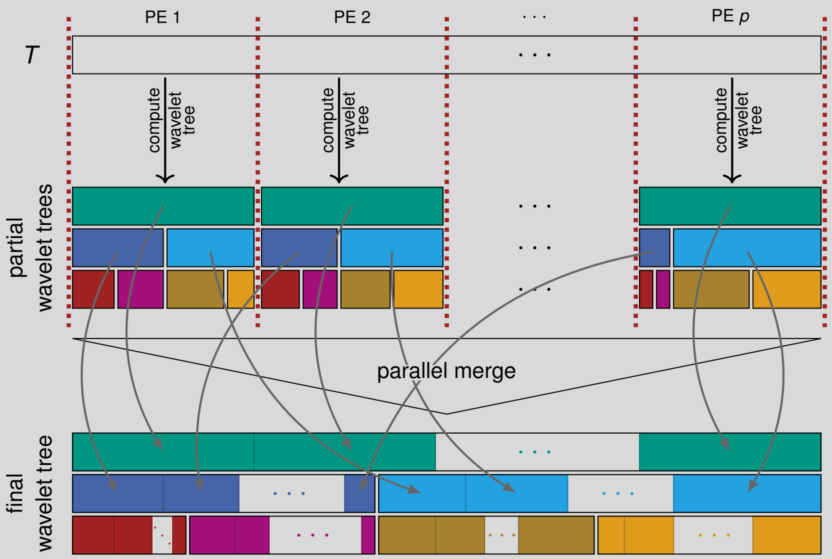
File	lut	ext	shuf64	shuf128	shuf256	shuf512	pc	pc-ss
dblp.xml	433.44	722.21	614.24	834.92	1 197.80	**1 477.77**	608.43	752.48
dna	529.32	883.00	563.11	668.93	862.49	**1 011.45**	594.02	745.68
english	456.91	770.55	677.96	906.42	1 304.80	**1 642.69**	623.08	704.90
pitches	448.02	749.24	686.88	886.62	1 276.36	**1 584.19**	578.70	328.47
proteins	375.73	575.99	565.63	707.23	985.35	**1 178.02**	633.58	761.41
sources	451.24	757.75	650.22	882.45	1 296.80	**1 632.85**	594.22	754.72
cc.16gib	453.97	729.58	653.25	875.61	1 265.27	**1 604.84**	628.46	752.97
dna.16gib	436.89	644.08	483.45	451.33	537.36	593.96	**669.70**	650.33
wiki.16gib	447.95	714.42	634.91	871.14	1 267.69	**1 604.39**	591.01	753.05
ru.8gib	317.20	642.51	506.04	660.23	938.68	**1 121.03**	346.96	170.44

Parallel Wavelet Tree Construction in Practice

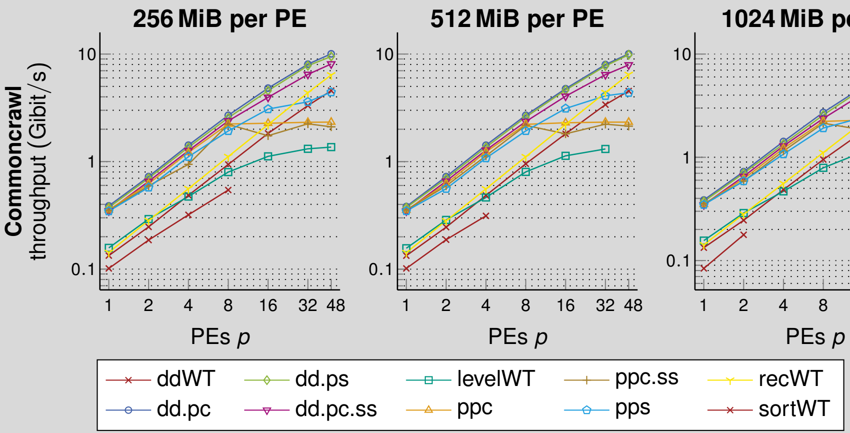
Domain Decomposition [Fue+17]

- create wavelet tree in parallel using p PEs
- each PE gets a consecutive slice of text
- each PE builds partial wavelet tree for its text
- merge partial wavelet trees in parallel

- can utilize any sequential algorithm
- very fast in practice
- $O(n \lg \sigma / \sqrt{\lg n})$ work and $O(\sigma + \lg n)$ time [Shu20]


PE 1
PE 2
...
PE p
T
compute wavelet tree
compute wavelet tree
compute wavelet tree
partial wavelet trees
parallel merge
final wavelet tree

Experiments: Parallel Wavelet Tree Construction

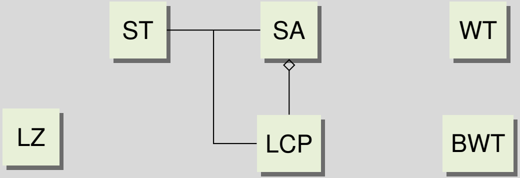


Conclusion and Outlook

This Lecture

- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees

Linear Time Construction



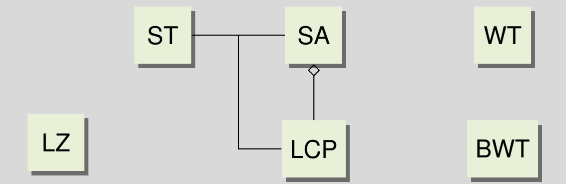
Conclusion and Outlook

This Lecture

- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees

- select on bit vectors
- practical algorithms for wavelet tree construction

Linear Time Construction



Conclusion and Outlook

This Lecture

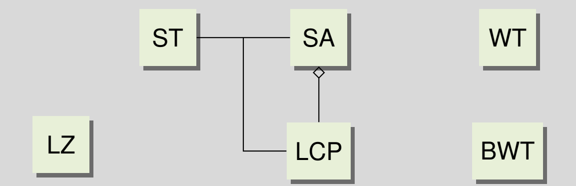
- wavelet tree and wavelet matrix
- Huffman-shaped wavelet trees

- select on bit vectors
- practical algorithms for wavelet tree construction

Next Lecture

- FM-index
- r-Index

Linear Time Construction



Bibliography I

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

[CNP15] Francisco Claude, Gonzalo Navarro, and Alberto Ordóñez Pereira. "The Wavelet Matrix: An Efficient Wavelet Tree for Large Alphabets". In: *Inf. Syst.* 47 (2015), pages 15–32. DOI: 10.1016/j.is.2014.06.002.

[Din+23] Patrick Dinklage, Johannes Fischer, Florian Kurpicz, and Jan-Philipp Tarnowski. "Bit-Parallel (Compressed) Wavelet Tree Construction". In: *DCC*. IEEE, 2023, pages 81–90. DOI: 10.1109/DCC55655.2023.00016.

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

Bibliography II

[Fue+17] José Fuentes-Sepúlveda, Erick Elejalde, Leo Ferres, and Diego Seco. "Parallel Construction of Wavelet Trees on Multicore Architectures". In: *Knowl. Inf. Syst.* 51.3 (2017), pages 1043–1066. DOI: 10.1007/s10115-016-1000-6.

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

[Kan18] Yusaku Kaneta. "Fast Wavelet Tree Construction in Practice". In: *SPIRE*. Volume 11147. Lecture Notes in Computer Science. Springer, 2018, pages 218–232. DOI: 10.1007/978-3-030-00479-8_18.

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

[Shu20] Julian Shun. "Improved parallel construction of wavelet trees and rank/select structures". In: *Inf. Comput.* 273 (2020), page 104516. DOI: 10.1016/j.ic.2020.104516.