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
  - $\text{rank}_1(i) = i - \text{rank}_0(i)$

- For all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $\frac{n}{s'} \cdot \lg n = O(\frac{n}{\lg n}) = o(n)$ 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
  - $\text{rank}_1(i) = i - \text{rank}_0(i)$
- for all $\lfloor \frac{n}{s'} \rfloor$ super blocks, store number of 0s from beginning of bit vector to end of super-block
- $n/s' \cdot \lg n = O(\frac{n \lg n}{\lg n}) = o(n)$ 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
    $\text{rank}_1(i) = i - \text{rank}_0(i)$

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

- for all $\lfloor \frac{n}{s} \rfloor$ blocks, store number of 0s from beginning of super block to end of block
  - $n/s \cdot \lg s' = O\left(\frac{n \lg \lg n}{\lg n}\right) = o(n)$ 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 $s = \left\lfloor \frac{\lg n}{2} \right\rfloor$
  - super blocks of size $s' = s^2 = \Theta(\lg^2 n)$
  - information for 0s or 1s enough
    - $\text{rank}_1(i) = i - \text{rank}_0(i)$

- for all $\left\lfloor \frac{n}{s} \right\rfloor$ 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 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 $select_0(i) = S[i]$ 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 $select_0(i) = S[i]$ if $k \in O(n/\log n)$ this suffice

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

- \( \text{select}_0 \) 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 \( \text{select}_0(i) = S[i] \) if \( k \in O(n/\lg n) \) this suffice

- better: \( k/b \) variable-sized super-blocks \( B_i \), such that super-block contains \( b = \lg^2 n \) zeros
  - \( \text{select}_0(i) = \sum_{j=0}^{\lfloor i/b \rfloor - 1} |B_j| + \text{select}_0(B_{\lfloor i/b \rfloor}, j - (\lfloor i/b \rfloor b)) \)

- 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

- $select_0$ 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 $k \in O(n/\lg n)$ 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

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 $select_0(i) = S[i]$ if $k \in O(n/\log n)$ this suffice

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

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

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

Given a text $T$ over an alphabet of size $\sigma$, 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**
Definition: Bit Representation
Given a text $T$ over an alphabet of size $\sigma$, 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

for simplicity characters are integers
bit representation is integer in binary
**Preliminaries**

**Definition: Bit Representation**
Given a text $T$ over an alphabet of size $\sigma$, each character can be represented using $\lceil \log_2 \sigma \rceil$ bits.
- the leftmost bit is the **most significant bit** and
- the rightmost bit is the **least significant bit**

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

**Example:**
- Character 0: 00000000 (MSB)
- Character 1: 00000001
- Character 2: 00000010

**Definition: Bit Prefix**
A bit prefix of length $k$ are the $k$ MSBs of a character's bit representation.
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.
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)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$[0, 7]$

\[
\begin{array}{cccccccc}
0 & 1 & 6 & 7 & 1 & 5 & 4 & 2 & 6 & 3 \\
0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
\end{array}
\]
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 0 1 1 0 0 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 0 0 0 1 0
```
Wavelet Trees (2/2)
Wavelet Trees (2/2)

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
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
```

```
[0, 7]
```

```
[0, 3]
```

```
[4, 7]
```
Wavelet Trees (2/2)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

[0, 7]  

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

[0, 3]  

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

[4, 7]  

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Wavelet Trees (2/2)

[0, 7]

[0, 3]

[0, 1]

[2, 3]

[4, 7]
Wavelet Trees (2/2)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example Wavelet Trees**

- **[0, 7]**
- **[0, 3]**
- **[4, 7]**
- **[0, 1]**
- **[2, 3]**
- **[4, 5]**
- **[6, 7]**
Wavelet Trees (2/2)

![Wavelet Tree Diagram]

- Rank function $\text{rank}_6(9)$
- Example partitioning:
  - $[0, 7]$
  - $[0, 3]$
  - $[4, 7]$
  - $[0, 1]$
  - $[2, 3]$
  - $[4, 5]$
  - $[6, 7]$

Matrix representation:

```
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 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)

```
[0, 7]

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

[0, 3]

0 1 1 2 3
0 0 0 1 1

[4, 7]

6 7 5 4 6
1 1 0 0 1

[0, 1]

0 1 1
0 1 1

[2, 3]

2 3
0 1

[4, 5]

5 4
1 0

[6, 7]

6 7 6
0 1 0

rank_{6}(9) = 110
```
Wavelet Trees (2/2)

![Wavelet Trees Diagram]

[0, 7] [0, 3] [4, 7]

[0, 1] [2, 3] [4, 5] [6, 7]
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

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

[0, 7]

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

[0, 3]

0 1 1 2 3
0 0 0 1 1

[4, 7]

6 7 5 4 6
1 1 0 0 1

[0, 1]

0 1 1
0 1 1

[2, 3]

2 3
0 1

[4, 5]

5 4
1 0

[6, 7]

6 7 6
0 1 0
The Intervals of a Wavelet Tree

- In each node, all represented characters share a bit prefix.
- On depth $\ell$ 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 $\ell$ the longest common bit prefix has length $\ell - 1$.
- The bit prefixes form intervals.
in each node, all represented characters share a bit prefix
on depth $\ell$ 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-Queries
- use rank queries on bit vectors
- at depth $\ell$ as for $\ell$-th MSB
- follow through tree according to bit
- as seen on a previous slide
### Rank-Queries
- Use rank queries on bit vectors
- At depth $\ell$ as for $\ell$-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-Queries
- Use rank queries on bit vectors
- At depth $\ell$ as for $\ell$-th MSB
- Follow through the 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)

<table>
<thead>
<tr>
<th>Rank-Queries</th>
<th>Select-Queries</th>
<th>Access-Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>use rank queries on bit vectors</td>
<td>identify leaf containing character</td>
<td>follow bits through the wavelet tree</td>
</tr>
<tr>
<td>at depth $\ell$ as for $\ell$-th MSB</td>
<td>select corresponding occurrence in leaf</td>
<td>return read bits</td>
</tr>
<tr>
<td>follow through tree according to bit</td>
<td>backtrack position up the tree to the root</td>
<td>same as rank but returning bit pattern instead of final rank</td>
</tr>
<tr>
<td>as seen on a previous slide</td>
<td>requires up and down traversal of the wavelet tree</td>
<td>see example on the board</td>
</tr>
<tr>
<td></td>
<td>see example on the board</td>
<td></td>
</tr>
</tbody>
</table>

**PINGO** what is the query time of rank queries in wavelet trees?
Lemma: Query Times Wavelet Tree

Given a text $T$ over an alphabet of size $\sigma$, 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 $\alpha$
- $reverse(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

**Definition: Bit-Reversal Permutation**

The **bit-reversal permutation** $\rho_k$ 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 $\alpha$
- $\text{reverse}(\alpha)$ reverses the bits
- the MSB becomes the least significant bit

**Definition: Bit-Reversal Permutation**

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

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

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

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

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

**Definition: Bit-Reversal Permutation**

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

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

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

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

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

Definition: Wavelet Matrix [CNP15]

Given a text $T$ of length $n$ over an alphabet of size $\sigma$, a wavelet matrix consists of

- bit vectors $BV_\ell$ 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 $BV_\ell$
- $BV_1$ contains all MSBs in text order
- $BV_\ell$ contains the $\ell$-th MSB the character at position $i$ in $BV_{\ell-1}$ at position
  - $\text{rank}_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell-1] + \text{rank}_1(i)$ if $BV_{\ell-1} = 1$
Alternative Representation

- alternative representation of wavelet trees
- removing tree structure
- only two areas per level
- 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 $\sigma$ a wavelet matrix consists of
- bit vectors $BV_\ell$ 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 $BV_\ell$
- $BV_1$ contains all MSBs in text order
- $BV_\ell$ contains the $\ell$-th MSB the character at position $i$ in $BV_{\ell-1}$ at position
  - $\text{rank}_0(i)$ if $BV_{\ell-1} = 0$ and
  - $Z[\ell - 1] + \text{rank}_1(i)$ 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, no tree structure.
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)
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?
queries on the wavelet matrix work similar
example on the board
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 $\ell$-th MSB of sorted sequence
  - sorted sequence is new text
Naive Wavelet Tree and Wavelet Matrix Construction (1/2)

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 $\ell$-th MSB of sorted sequence
  - sorted sequence is new text

Wavelet Matrix
- first level are MSBs of characters of text
- for each level $\ell > 1$
  - stably sort text by $\ell - 1$ MSB
  - take $\ell$-th MSB of sorted sequence
  - sorted sequence is new text
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 $\sigma$, the wavelet tree and wavelet matrix require $(1 + o(1))n \lfloor \lg \sigma \rfloor$ bits of space and can be constructed in $O(n \lg \sigma)$ time.
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 \( \sigma \), 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 an asymptotically faster construction method?
Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every \( \tau \)-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 boardbold
Better Wavelet Tree Construction [Bab+15; MNV16]

- using requires broadword programming
- every $\tau$-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 $\sigma$, 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 \[\text{[Bab+15; MNV16]}\]

- using requires broadword programming
- every $\tau$-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 $\sigma$, 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 \[\text{[Din+23; Kan18]}\]
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 $\alpha$ is a leaf and has weight $\text{Hist}[\alpha]$
- 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

Huffman Codes (Recap)

- Idea is to create a binary tree
- Each character $\alpha$ is a leaf and has weight $\text{Hist}[\alpha]$
- 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

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$hc(\alpha)$</th>
<th>$chc(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(11)$_2$</td>
<td>(11)$_2$</td>
</tr>
<tr>
<td>3</td>
<td>(01)$_2$</td>
<td>(10)$_2$</td>
</tr>
<tr>
<td>6</td>
<td>(100)$_2$</td>
<td>(011)$_2$</td>
</tr>
<tr>
<td>7</td>
<td>(101)$_2$</td>
<td>(010)$_2$</td>
</tr>
<tr>
<td>0</td>
<td>(0000)$_2$</td>
<td>(0011)$_2$</td>
</tr>
<tr>
<td>2</td>
<td>(0001)$_2$</td>
<td>(0010)$_2$</td>
</tr>
<tr>
<td>4</td>
<td>(0010)$_2$</td>
<td>(0001)$_2$</td>
</tr>
<tr>
<td>5</td>
<td>(0011)$_2$</td>
<td>(0000)$_2$</td>
</tr>
</tbody>
</table>

- Huffman codes (hc)
- canonical Huffman codes (hc) that are bit-wise negated
Huffman-shaped Wavelet Trees

<table>
<thead>
<tr>
<th>α</th>
<th>hc(α)</th>
<th>chc(α)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(11)₂</td>
<td>(11)₂</td>
</tr>
<tr>
<td>3</td>
<td>(01)₂</td>
<td>(10)₂</td>
</tr>
<tr>
<td>6</td>
<td>(100)₂</td>
<td>(011)₂</td>
</tr>
<tr>
<td>7</td>
<td>(101)₂</td>
<td>(010)₂</td>
</tr>
<tr>
<td>0</td>
<td>(0000)₂</td>
<td>(0011)₂</td>
</tr>
<tr>
<td>2</td>
<td>(0001)₂</td>
<td>(0010)₂</td>
</tr>
<tr>
<td>4</td>
<td>(0010)₂</td>
<td>(0001)₂</td>
</tr>
<tr>
<td>5</td>
<td>(0011)₂</td>
<td>(0000)₂</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>7</th>
<th>1</th>
<th>5</th>
<th>4</th>
<th>2</th>
<th>6</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Commoncrawl

DNA

Proteins

Wikipedia

Experiments: Sequential Wavelet Tree Construction

Commoncrawl

DNA

Proteins

Wikipedia

Experiments: Sequential Wavelet Tree Construction
## Experiments: Vectorized Wavelet Tree Construction [Din+23]

<table>
<thead>
<tr>
<th>File</th>
<th>lut</th>
<th>ext</th>
<th>shuf64</th>
<th>shuf128</th>
<th>shuf256</th>
<th>shuf512</th>
<th>pc</th>
<th>pc-ss</th>
</tr>
</thead>
<tbody>
<tr>
<td>dblp.xml</td>
<td>433.44</td>
<td>722.21</td>
<td>614.24</td>
<td>834.92</td>
<td>1197.80</td>
<td><strong>1477.77</strong></td>
<td>608.43</td>
<td>752.48</td>
</tr>
<tr>
<td>dna</td>
<td>529.32</td>
<td>883.00</td>
<td>563.11</td>
<td>668.93</td>
<td>862.49</td>
<td><strong>1011.45</strong></td>
<td>594.02</td>
<td>745.68</td>
</tr>
<tr>
<td>english</td>
<td>456.91</td>
<td>770.55</td>
<td>677.96</td>
<td>906.42</td>
<td>1304.80</td>
<td><strong>1642.69</strong></td>
<td>623.08</td>
<td>704.90</td>
</tr>
<tr>
<td>pitches</td>
<td>448.02</td>
<td>749.24</td>
<td>686.88</td>
<td>886.62</td>
<td>1276.36</td>
<td><strong>1584.19</strong></td>
<td>578.70</td>
<td>328.47</td>
</tr>
<tr>
<td>proteins</td>
<td>375.73</td>
<td>575.99</td>
<td>565.63</td>
<td>707.23</td>
<td>985.35</td>
<td><strong>1178.02</strong></td>
<td>633.58</td>
<td>761.41</td>
</tr>
<tr>
<td>sources</td>
<td>451.24</td>
<td>757.75</td>
<td>650.22</td>
<td>882.45</td>
<td>1296.80</td>
<td><strong>1632.85</strong></td>
<td>594.22</td>
<td>754.72</td>
</tr>
<tr>
<td>cc.16gib</td>
<td>453.97</td>
<td>729.58</td>
<td>653.25</td>
<td>875.61</td>
<td>1265.27</td>
<td><strong>1604.84</strong></td>
<td>628.46</td>
<td>752.97</td>
</tr>
<tr>
<td>dna.16gib</td>
<td>436.89</td>
<td>644.08</td>
<td>483.45</td>
<td>451.33</td>
<td>537.36</td>
<td>593.96</td>
<td><strong>669.70</strong></td>
<td>650.33</td>
</tr>
<tr>
<td>wiki.16gib</td>
<td>447.95</td>
<td>714.42</td>
<td>634.91</td>
<td>871.14</td>
<td>1267.69</td>
<td><strong>1604.39</strong></td>
<td>591.01</td>
<td>753.05</td>
</tr>
<tr>
<td>ru.8gib</td>
<td>317.20</td>
<td>642.51</td>
<td>506.04</td>
<td>660.23</td>
<td>938.68</td>
<td><strong>1121.03</strong></td>
<td>346.96</td>
<td>170.44</td>
</tr>
</tbody>
</table>
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

$O(n \log \sigma / \sqrt{\log n})$ work and $O(\sigma + \log n)$ time

[Shu20]
Partial wavelet trees...
Experiments: Parallel Wavelet Tree Construction

Experiments: Parallel Wavelet Tree Construction

- Common crawl throughput (Gbit/s)
- PEs p
- 256 MiB per PE
- 512 MiB per PE
- 1024 MiB per PE

Graphs showing throughput for different data sets and PEs.
Conclusion and Outlook

This Lecture

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

Linear Time Construction

ST | SA
---|---
LZ | LCP | BWT

WT

Institute of Theoretical Informatics, Algorithm Engineering
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

ST → SA → WT
LZ → LCP → BWT
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

Diagram:
- ST
- SA
- WT
- LZ
- LCP
- BWT
Bibliography I


[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.

Bibliography II


