Wavelet Trees

CSCI 7000 - Advanced Data Structures
Final Project

Varad Raut, Sagar Pathare, Chinmay Agrawal

December 1, 2022

Navarro, G. (2014). Wavelet trees for all. Journal of Discrete Algorithms, 25, 2-20.

Contents

Motivation

- Let's look at some problems on a sequence.
  - Counting occurrences of an element till position `i`?
    - `O(n)`
  - Finding the position of the `i-th` occurrence of an element?
    - `O(n)`
- These problems occur at a lot of places.
- Can we do better?

Wavelet Trees

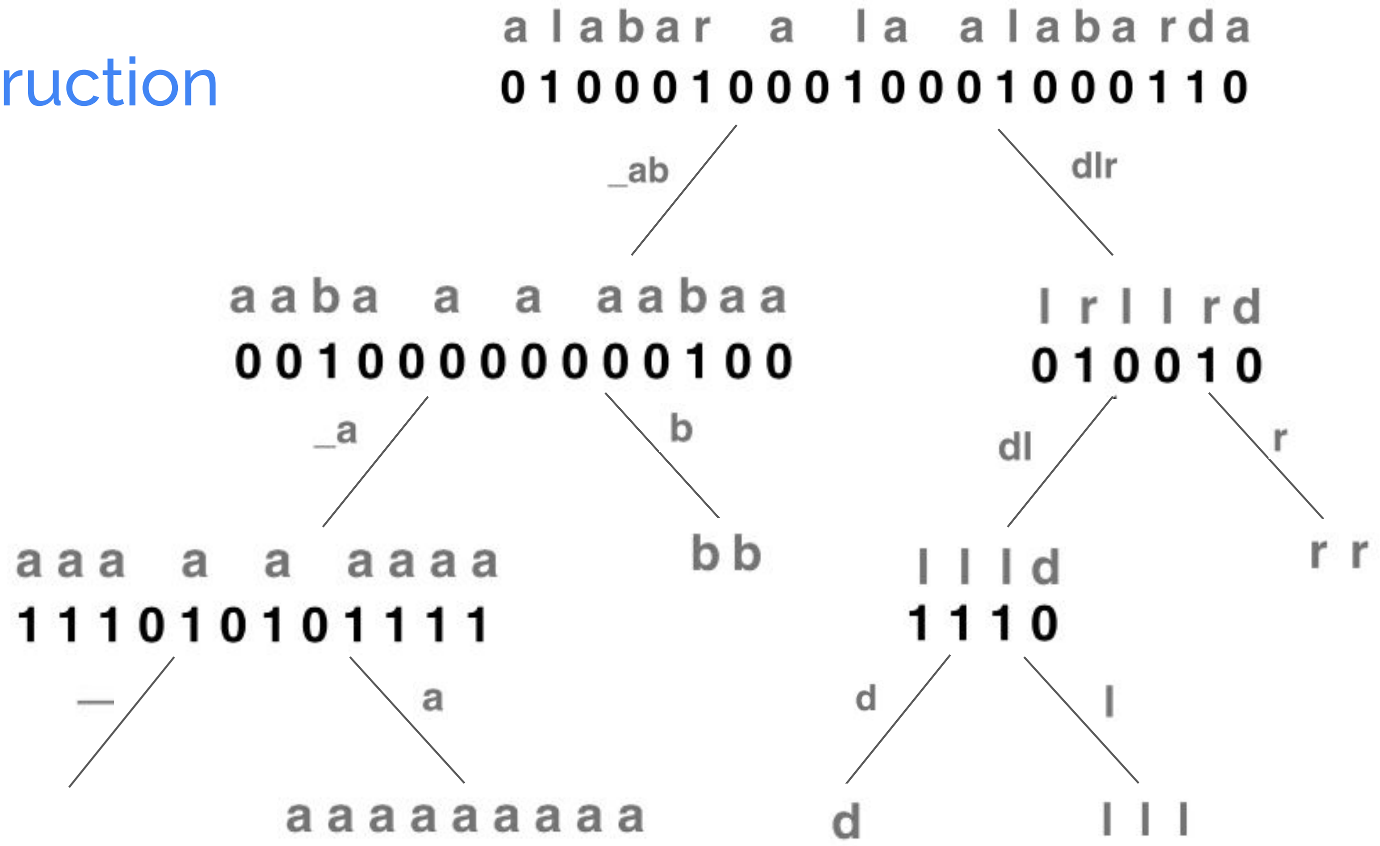
- Wavelet tree is as a space-efficient data structure to represent a sequence and answer some queries on it.
- The name originates from "wavelet packet decomposition" in signal processing.
- The “high” and “low” symbol values of the sequence are separated and the resulting subsequences are recursively subdivided.

Wavelet Trees

- Let $S[1,n] = \{s_1, s_2, ..., s_n\}$ be a sequence of symbols $s_i \in \Sigma$, where Σ is a finite alphabet of size σ

- Example
  - Sequence, S = alabar a la alabarda
  - Alphabet, $\Sigma = \{\_, a, b, d, l, r\}$
  - Alphabet size, $\sigma = 6$

Construction



Binary Operations

- Rank
  - `rank`$_0$`(i)` returns the count of unset bits till position `i`
  - `rank`$_1$`(i)` returns the count of set bits till position `i`

- Select
  - `select`$_0$`(i)` returns the position of the `i-th` unset bit
  - `select`$_1$`(i)` returns the position of the `i-th` set bit

Binary Operations

bitmap = 1001

- `rank`$_0$`(4)` = count of unset bits till position `4` = 2
- `rank`$_1$`(2)` = count of set bits till position `2` = 1

- `select`$_0$`(1)` = position of the `1st` unset bit = 2
- `select`$_1$`(2)` = position of the `2nd` set bit = 4

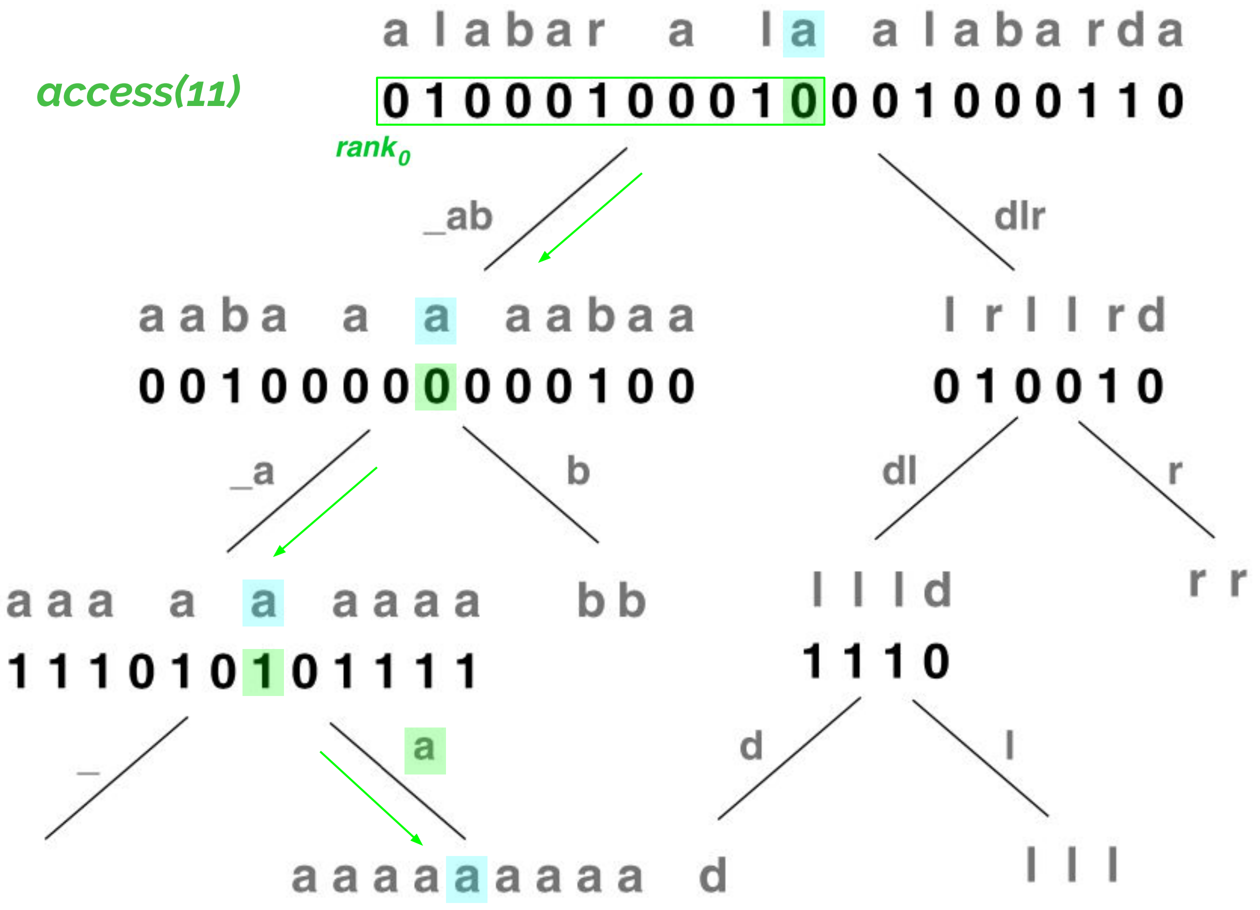
Wavelet Tree Operations

- Access
  - `access(i)` returns `S[i]`
- Rank
  - $\texttt{rank}_c\texttt{(i)}$ returns the count of occurences of element `c` till position `i`
- Select
  - $\texttt{select}_c\texttt{(i)}$ returns the position of the `i-th` occurrence of element `c`

Access

- `access(i)` returns `S[i]`
- Check for the bit at `i` and calculate $\texttt{rank}_{0/1}$
- Update `i` and traverse down to the leaf based on `i`.
- Return `i-th` leaf element.

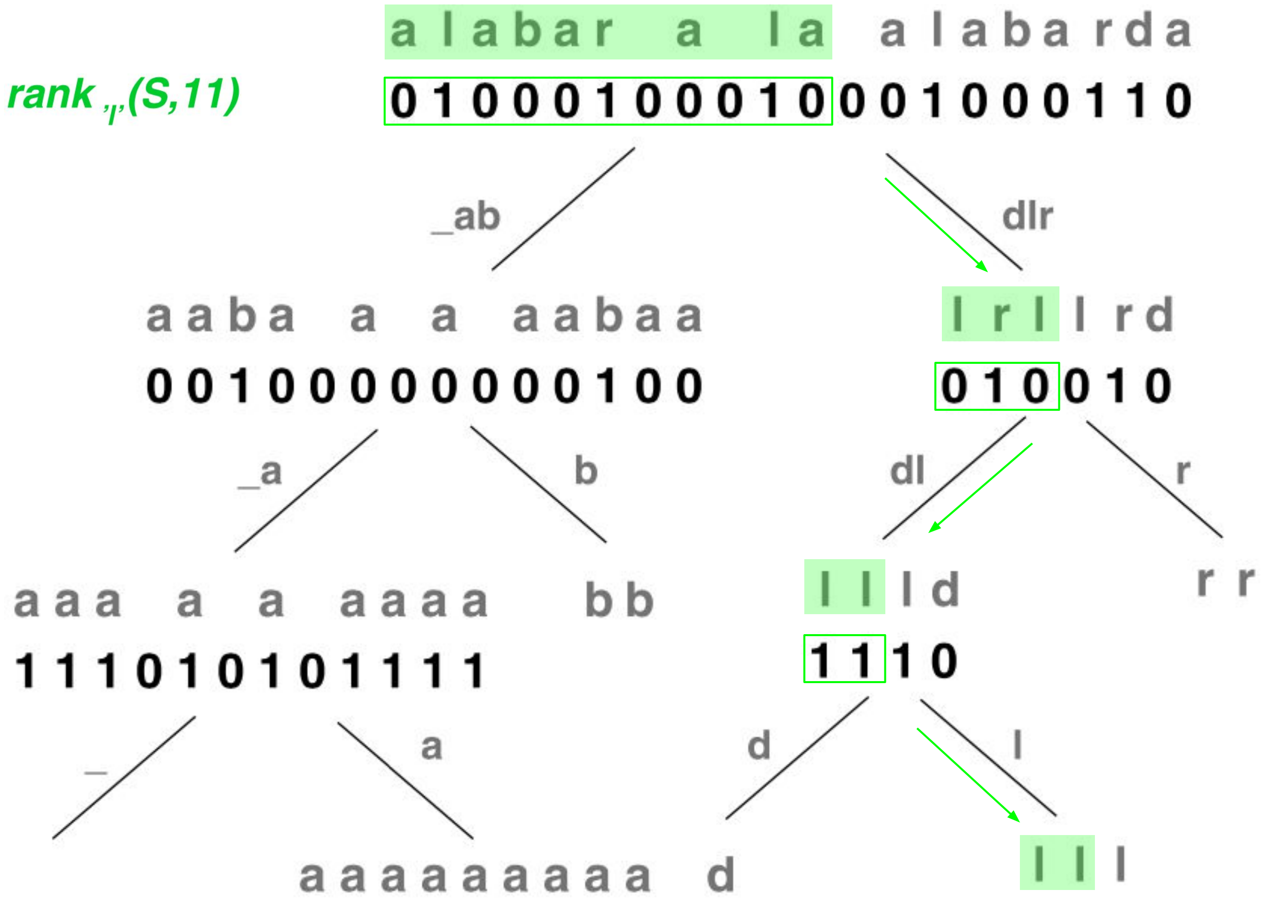
Access


access(11)
a l a b a r   a   l a   a l a b a r d a
0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0
rank0
_ab
dlr
a a b a   a   a   a a b a a
0 0 1 0 0 0 0 0 0 0 0 1 0 0
l r l l r d
0 1 0 0 1 0
_a
b
dl
r
a a a   a   a   a a a a
1 1 1 0 1 0 1 0 1 1 1 1
b b
l l l d
1 1 1 0
r r
_
a
d
l
a a a a a a a a a
d
l l l

Rank

- $\texttt{rank}_c\texttt{(i)}$ returns the count of occurences of element `c` till position `i`
- Update `i` using $\texttt{rank}_{0/1}$ based on the child node.
- Traverse depending on the bit of the element in the current node.
- Return `i`.

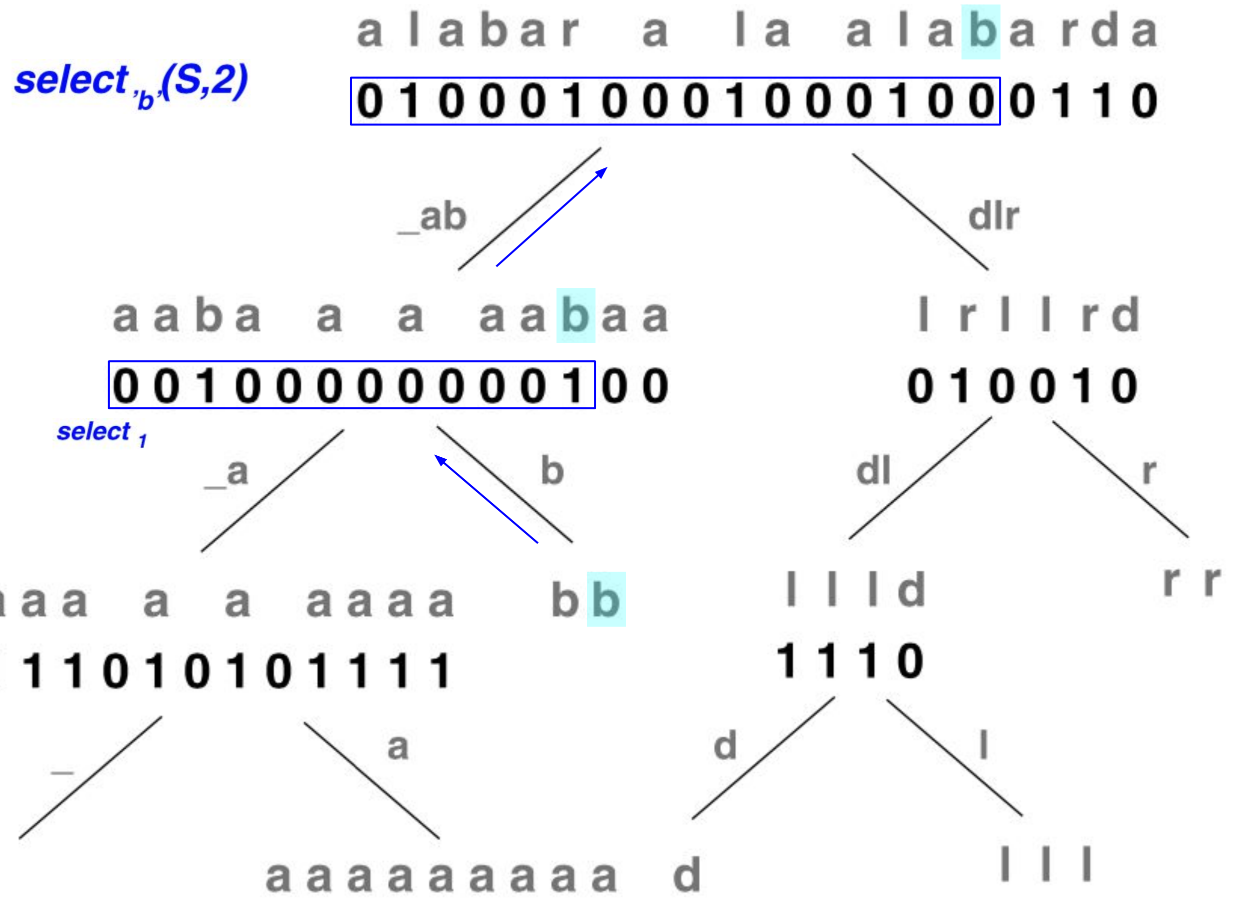
Rank


rank 'l'(S,11)
a l a b a r   a   l a   a l a b a r d a
0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0
_ab
dlr
a a b a   a   a   a a b a a
0 0 1 0 0 0 0 0 0 0 0 1 0 0
l r l l r d
0 1 0 0 1 0
_a
b
dl
r
a a a   a   a   a a a a
1 1 1 0 1 0 1 0 1 1 1 1
b b
l l l d
1 1 1 0
r r
_
a
d
l
a a a a a a a a a
d
l l l

Select

- `select`$_c$`(i)` returns the position of the `i-th` occurrence of element `c`
- Start from the i-th position of the element in the corresponding leaf node.
- Traverse up to the root while finding the position of `0/1` using select$_{0/1}$ depending on if node is left/right child, up to i-th occurrence.
- Return the final position.

Select


select 'b' (S,2)
a l a b a r _ a _ l a _ a l a b a r d a
0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 1 0
_ab
dlr
a a b a _ a _ a _ a a b a a
0 0 1 0 0 0 0 0 0 0 0 1 0 0
select 1
l r l l r d
0 1 0 0 1 0
_a
b
dl
r
a a a _ a _ a _ a a a a
1 1 1 0 1 0 1 0 1 1 1 1
b b
l l l d
1 1 1 0
r r
_
a
d
l
a a a a a a a a a
d
l l l

Space Analysis

- Balanced Wavelet Tree (on the alphabet)
- Height of tree = $\lceil \log_2 \sigma \rceil$
- Exactly n bits at each level. At most n bits at the last level.
- Total space = $n \lceil \log_2 \sigma \rceil$ bits (upper bound)

- What about space taken by pointers?
    - $\sigma - 1$ internal nodes, each pointer taking `wordsize` bits of space
    - Total space = $O(\sigma \cdot \mathtt{wordsize})$ bits

Time Analysis - Access

Looking for symbol at position `i` in the Tree. Starting from the root, at each level,

- Find child: $O(1)$ using the bitmap at each level
- Find position of `i` in the child: `O(time for rank)` using $\mathrm{rank}_{0/1}$ on the bitmaps

There are $\lceil \log_2 \sigma \rceil$ levels.

Thus, total time =

- $O(n\log_2\sigma)$ - naive rank using linear scan
- $O(\log_2\sigma)$ - precomputed rank

Time Analysis - Rank

Finding rank of symbol `s`, up to given index `i` in the Wavelet Tree. Starting from the root, at each level,

- Find child: `O(1)` using the bitmap at each level
- Find corresponding index for `i` in the child: `O(time for rank)` using $\mathtt{rank}_{0/1}$ on the bitmaps

There are $\lceil \log_2 \sigma \rceil$ levels.

Thus, total time =

- $O(n\log_2 \sigma)$ - naive rank using linear scan
- $O(\log_2 \sigma)$ - precomputed rank

Time Analysis - Select

Finding index of the `i-th` occurrence of symbol `s` in the original sequence.
Traversing upwards from the leaf, at each level,

- Find child bit from which we traversed up: $O(1)$.
- Find index of `i-th` occurrence of child bit: `O(time for select)` using $\mathtt{select}_{0/1}$ on the bitmaps.

There are $\lceil \log_2 \sigma \rceil$ levels.

Thus, total time =

- $O(n \log_2 \sigma)$ - naive select using linear scan
- $O(\log_2 \sigma)$ - precomputed select

Binary Rank & Select Optimizations

- `POPCNT` Instruction - `O(1)` space, `O(n/wordsize)` time.
    - Native CPU instruction for `population count` (number of set bits in a machine word).

- Naive Precomputation - $O(n\log_2 n)$ space, `O(1)` time.
    - At each node, precompute ranks for each bit in the bitmap.
    - Store it in an array of size = len(bitmap)

Binary Rank & Select Optimizations

- Smart Precomputation - `o(n)` space, `O(1)` time.
  - Essentially, divide the bitmap into equal-length `samples`.
  - Store rank value for each `sample`.
  - Further divide each `sample` into equal-length `subsamples`.
  - Store `offset` rank values (from previous `sample`) for each `subsample`.
  - Use `popcount` within the `subsample` to get the final rank value.

Succinctness

- Succinct data structure: Uses amount of space almost as much as the information theoretical lower bound, e.g. Heap
- Succinctness in Wavelet Trees: Compressed bitmap representation
- The compression that can be achieved for a sequence S is given by its Empirical Zero-Order Entropy, $H_0(S)$.
- What is entropy?
    - Essentially, it's the number of bits required to represent each member of a set.
    - For a set of size n, worst-case entropy, $H_{wc}(S) = \log_2 n$

Empirical Zero-Order Entropy

- Compression of a sequence S is achieved by using the difference in frequencies of individual elements in S.
- Empirical zero-order entropy gives the amount of compression that can be achieved.

$$H_0(S) = \sum_{c \in \Sigma} (n_c/n) \log_2(n/n_c)$$

where n_c is the number of occurrences of c in S

Empirical Zero-Order Entropy

- If bitmaps of a wavelet tree are compressed to H_0, then overall space occupied by the tree = $n \cdot H_0(S)$
- Thus, any zero-order entropy coded bitmap representation with `O(1)` `rank` and `select` can be used to get a succinct wavelet tree.
- Example: ‘succinct indexable dictionary’ by Raman, Raman, and Rao.

Representations

- As a sequence of values
  - The wavelet tree on a sequence $S[1,n] = \{s_1, s_2, ..., s_n\}$ represents the values s_i.
- As a reordering
  - The wavelet tree structure describes a stable ordering of the symbols in S.
- As a grid of points
  - We have an $n \times n$ grid with n points so that no two points share the same row or column.

Applications

- As a sequence of values
  - Positional inverted indexes, Full-text indexes, Graphs, etc.
- As a reordering
  - Generic numeric sequences, Permutations, Document retrieval indexes, etc.
- As a grid of points
  - Discrete grids, Binary relations, Colored range queries, etc.

Positional inverted indexes

- A positional inverted index is a data structure that stores, for each word, the list of the positions where it appears in the collection.

- Example
  - `Sequence,` S = to be or not to be
  - `Alphabet,` Σ = to be or not
  - `Alphabet size,` $\sigma = 4$

Positional inverted indexes

to be or not to be

1	to	1, 5
2	be	2, 6
3	or	3
4	not	4

access(3) = 3 = or

access(5) = 1 = to

1 2 3 4 5 6

1 2 3 4 1 2

$\text{rank}_{\text{to}}(2) = \text{rank}_1(2) = 1$

$\text{rank}_{\text{be}}(6) = \text{rank}_2(6) = 2$

$\text{select}_{\text{be}}(1) = \text{select}_2(1) = 2$

$\text{select}_{\text{to}}(2) = \text{select}_1(2) = 5$

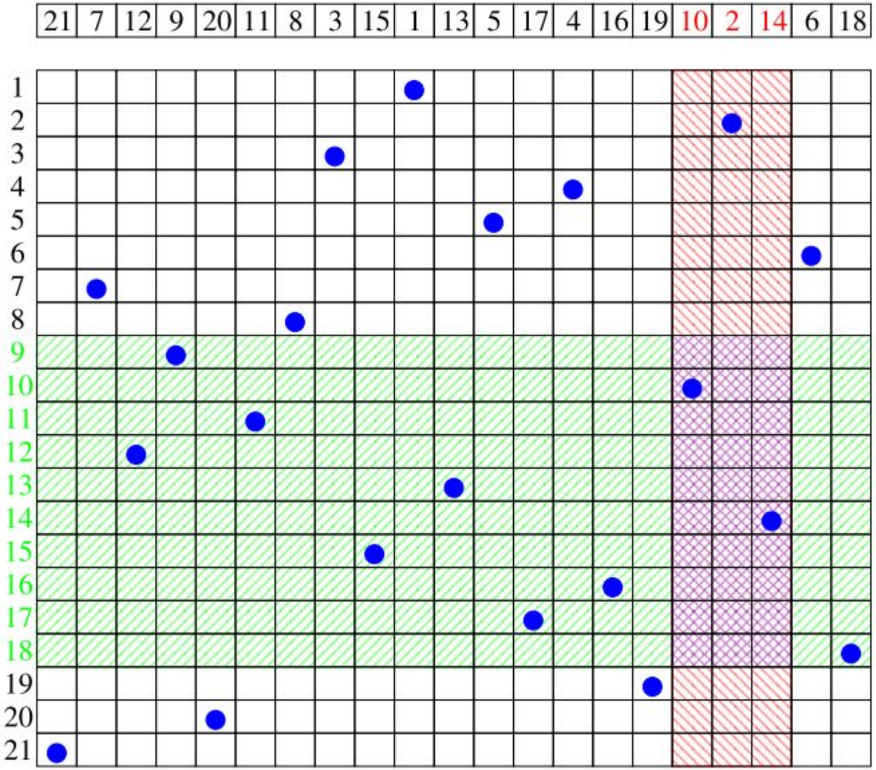
Range quantile query

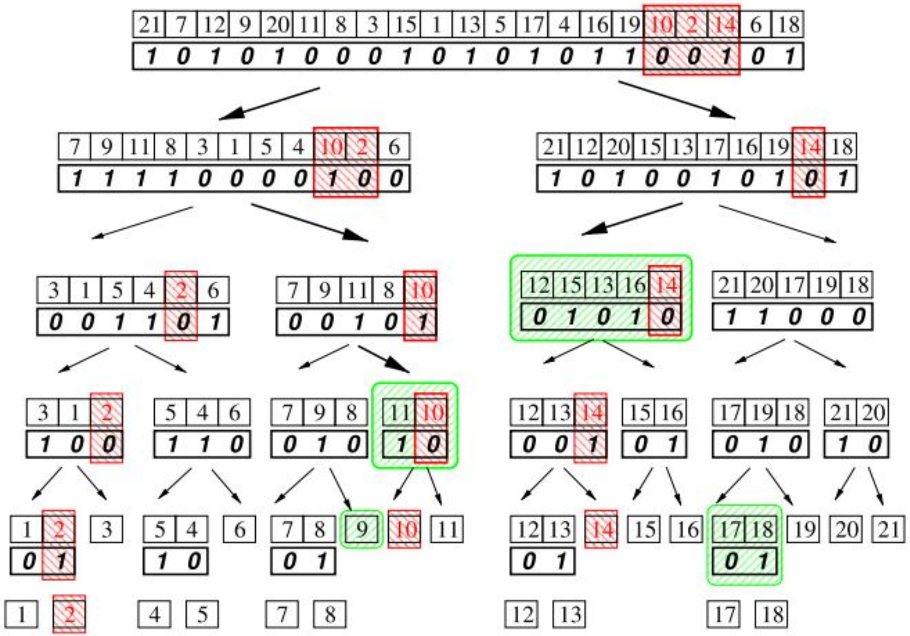
- For a sequence of numbers `S[1,n]` on the domain `[1..σ]`, given a range `[l,r]` and a value `k`, we can compute the `k-th` smallest element in `S[l,r]`.

- It uses the `rank` operation to successively narrow down the range until we arrive at a leaf, whose label is the `k-th` smallest element in `S[l,r]`.

- Nontrivial compared to Positional inverted indexes

Counting points in a rectangle





Implementation

- We implemented a wavelet tree library in C++ that represents a wavelet tree as a `sequence`, along with `precomputation`.
- Thanks to `C++ generics`, it supports creation of a wavelet tree of a sequence of any data type that satisfies the following two conditions-
    - Can be `hashed`
    - Can be compared using `<= operator`
- We implemented two applications
    - Positional inverted indexes
    - Range quantile queries

Implementation

```
Sequence, S: a l a b a r _ a _ l a _ a l a b a r d a
Alphabet, Σ: _ a b d l r
Alphabet size, σ: 6

traverse():

|────── level: 0
        subsequence: a l a b a r _ a _ l a _ a l a b a r d a
        bitmap:  0  1  0  0  0  1  0  0  0  1  0  0  0  1  0  0  0  1  1  0
        rank_0:  1  1  2  3  4  4  5  6  7  7  8  9 10 10 11 12 13 13 13 14
        select_0:  0  2  3  4  6  7  8 10 11 12 14 15 16 19
        select_1:  1  5  9 13 17 18

        |────── level: 1
                subsequence: a a b a _ a _ a _ a a b a a
                bitmap:  0  0  1  0  0  0  0  0  0  0  0  1  0  0
                rank_0:  1  2  2  3  4  5  6  7  8  9 10 10 11 12
                select_0:  0  1  3  4  5  6  7  8  9 10 12 13
                select_1:  2 11
```

```
access(10) = a
access(15) = b
access(-3) = Index out of range!
access(50) = Index out of range!

rank(l, 11) = 2
rank(b, 16) = 2
rank(z, 3) = Invalid input!

select(b, 2) = 15
select(a, 6) = 12
select(z, 3) = Invalid input!
```

Conclusion

- Wavelet tree is a versatile `succinct` data structure with a plethora of `applications`.

Thank You!