Dynamic Entropy-Compressed Sequences and Full-Text Indexes

Veli Mäkinen

Gonzalo Navarro

Department of Computer Science
University of Helsinki, Finland

Center for Web Research
Department of Computer Science
Universidad de Chile, Chile

Outline

Outline

Outline

Outline

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct $\Rightarrow$ little space on top of the raw data.
- Compressed $\Rightarrow$ space proportional to the data entropy.

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct ⇒ little space on top of the raw data.
- Compressed ⇒ space proportional to the data entropy.

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct $\Rightarrow$ little space on top of the raw data.
- Compressed $\Rightarrow$ space proportional to the data entropy.

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct ⇒ little space on top of the raw data.
- Compressed ⇒ space proportional to the data entropy.

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct $\Rightarrow$ little space on top of the raw data.
- Compressed $\Rightarrow$ space proportional to the data entropy.

Background

Succinct Data Structures

- Compacted to take little space.
- Must retain their functionality and direct accessibility.
- Improve performance considering memory hierarchy.
- Increasingly popular for technological reasons.
- Succinct $\Rightarrow$ little space on top of the raw data.
- Compressed $\Rightarrow$ space proportional to the data entropy.

Background

Binary Sequences

- Let $B = b_1 b_2 \ldots b_n$ be a binary sequence.
- $rank_b(B, i) =$ number of occurrences of b in $B[1, i]$.
- $select_b(B, j) =$ position of j-th occurrence of b in B.
- Useful for a myriad of compressed data structures.

Background

Binary Sequences

- Let $B = b_1 b_2 \ldots b_n$ be a binary sequence.
- $rank_b(B, i) =$ number of occurrences of b in $B[1, i]$.
- $select_b(B, j) =$ position of j-th occurrence of b in B.
- Useful for a myriad of compressed data structures.

Background

Binary Sequences

- Let $B = b_1 b_2 \dots b_n$ be a binary sequence.
- $rank_b(B, i) =$ number of occurrences of b in $B[1, i]$.
- $select_b(B, j) =$ position of j-th occurrence of b in B.
- Useful for a myriad of compressed data structures.

Background

Binary Sequences

- Let $B = b_1 b_2 \ldots b_n$ be a binary sequence.
- $rank_b(B, i) =$ number of occurrences of b in $B[1, i]$.
- $select_b(B, j) =$ position of j-th occurrence of b in B.
- Useful for a myriad of compressed data structures.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Compressed Binary Sequences

- Raman, Raman and Rao, SODA 2002.
- Divides the sequence into blocks...
- ... and uses variable-length representation for them.
- Take $nH_0 + o(n)$ bits of space.
- Solve *rank* and *select* in constant time.
- Static solution.

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Background

Dynamic Binary Sequences

- Chan, Hon and Lam, CPM 2004.
- A balanced tree with leaves managing $O(\log n)$ bits.
- Takes $O(n)$ bits of space.
- Solves *rank* and *select* in $O(\log n)$ time.
- Solves *insert* and *delete* in $O(\log n)$ time.
- Not the best possible solution (but ok for now).

Outline

Main Result

Our contribution

- A structure structure that takes $nH_0 + o(n)$ bits of space.
- It performs *rank*, *select*, *insert* and *delete* in $O(\log n)$ time.
- The first nH_0-size dynamic data structure for binary sequences.
- Many applications, details soon.

Main Result

Our contribution

- A structure structure that takes $nH_0 + o(n)$ bits of space.
- It performs *rank*, *select*, *insert* and *delete* in $O(\log n)$ time.
- The first nH_0-size dynamic data structure for binary sequences.
- Many applications, details soon.

Main Result

Our contribution

- A structure structure that takes $nH_0 + o(n)$ bits of space.
- It performs *rank*, *select*, *insert* and *delete* in $O(\log n)$ time.
- The first nH_0-size dynamic data structure for binary sequences.
- Many applications, details soon.

Main Result

Our contribution

- A structure structure that takes $nH_0 + o(n)$ bits of space.
- It performs *rank*, *select*, *insert* and *delete* in $O(\log n)$ time.
- The first nH_0-size dynamic data structure for binary sequences.
- Many applications, details soon.

Outline

A Succinct Version

Main line of work

Similar to Chan, Hon, and Lam, but...

- Leaves must hold more than $O(\log n)$ bits, so the overhead of tree pointers can be $o(n)$.
- Then leaves cannot be processed bitwise in $O(\log n)$ time.
- In the leaves, we cannot have space for $(1+\epsilon)S$ bits and use just S.
- Then we cannot easily handle leaf overflows in constant time.

A Succinct Version

Main line of work

Similar to Chan, Hon, and Lam, but...

- Leaves must hold more than $O(\log n)$ bits, so the overhead of tree pointers can be $o(n)$.
- Then leaves cannot be processed bitwise in $O(\log n)$ time.
- In the leaves, we cannot have space for $(1+\epsilon)S$ bits and use just S.
- Then we cannot easily handle leaf overflows in constant time.

A Succinct Version

Main line of work

Similar to Chan, Hon, and Lam, but...

- Leaves must hold more than $O(\log n)$ bits, so the overhead of tree pointers can be $o(n)$.
- Then leaves cannot be processed bitwise in $O(\log n)$ time.
- In the leaves, we cannot have space for $(1+\epsilon)S$ bits and use just S.
- Then we cannot easily handle leaf overflows in constant time.

A Succinct Version

Main line of work

Similar to Chan, Hon, and Lam, but...

- Leaves must hold more than $O(\log n)$ bits, so the overhead of tree pointers can be $o(n)$.
- Then leaves cannot be processed bitwise in $O(\log n)$ time.
- In the leaves, we cannot have space for $(1+\epsilon)S$ bits and use just S.
- Then we cannot easily handle leaf overflows in constant time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n) \log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n) \log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n)\log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n) \log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n)\log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

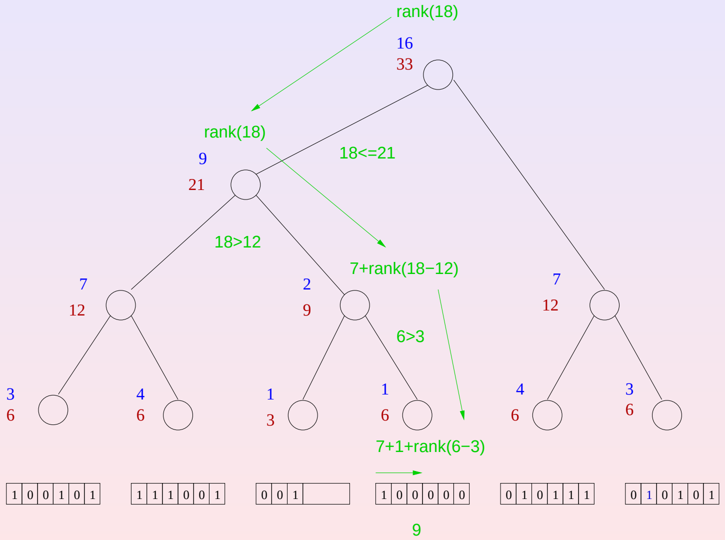
Data structure and queries

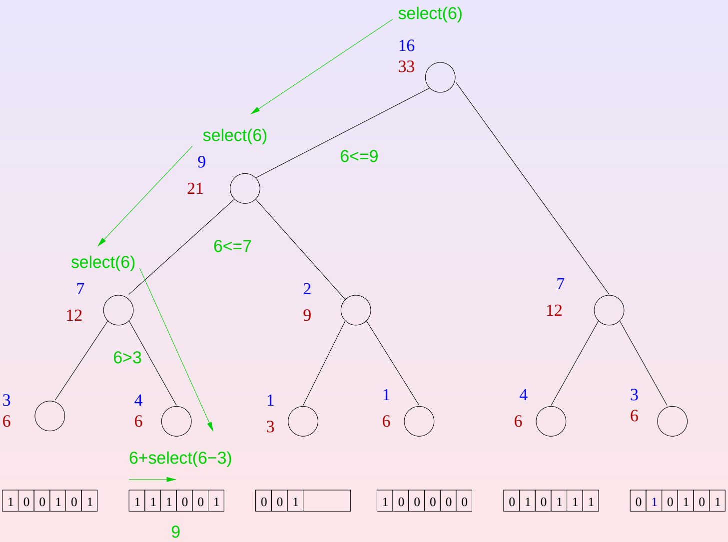
- Balanced binary tree.
- Leaves handle blocks of $f(n) \log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.

A Succinct Version

Data structure and queries

- Balanced binary tree.
- Leaves handle blocks of $f(n) \log n$ bits.
- Internal nodes know subtree size and weight.
- Extra space of the tree is $O(n/f(n))$ bits.
- *rank* and *select* are solved by a top-down traversal...
- ... plus a linear scan over the leaf sequence.
- 4-Russians technique permits scanning in $O(f(n))$ time.


rank(18)
16
33
rank(18)
9
21
18<=21
18>12
7+rank(18−12)
7
12
2
9
7
12
6>3
3
6
4
6
1
3
1
6
4
6
3
6
7+1+rank(6−3)
1 0 0 1 0 1
1 1 1 0 0 1
0 0 1
1 0 0 0 0 0
0 1 0 1 1 1
0 1 0 1 0 1
9


select(6)
16
33
select(6)
9
21
6<=9
select(6)
7
12
6<=7
2
9
7
12
6>3
3
6
4
6
1
3
1
6
4
6
3
6
6+select(6−3)
1 0 0 1 0 1
1 1 1 0 0 1
0 0 1
1 0 0 0 0 0
0 1 0 1 1 1
0 1 0 1 0 1
9

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

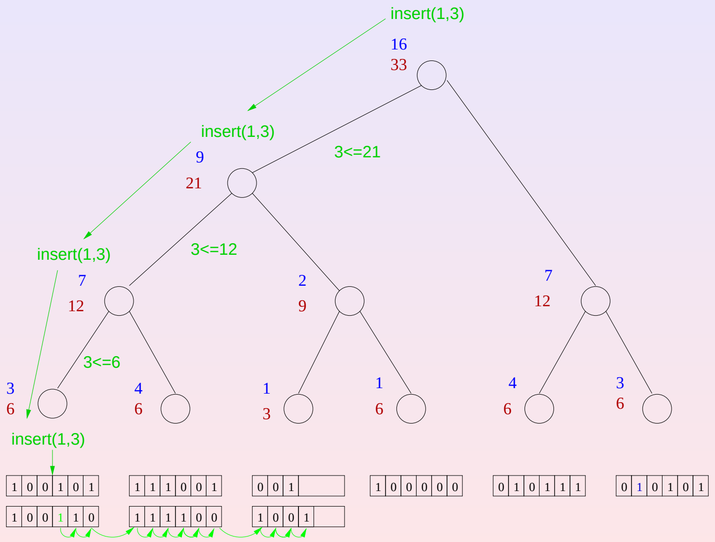
Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.

A Succinct Version

Insertions

- Find the position where the bit should be inserted.
- Insert it and move the following bits to the right.
- The bit that falls off is inserted into the next leaf.
- The process is repeated until...
- ... we reach a partial leaf, which is not totally full.
- Only one out of $f(n)$ leaves can be partial.
- Extra space is $n/f(n)$ bits.
- Insertion time is $O(\log n + f(n)^2)$.


insert(1,3)
16
33
insert(1,3)
3<=21
9
21
insert(1,3)
3<=12
7
12
2
9
7
12
3<=6
3
6
4
6
1
3
1
6
4
6
3
6
insert(1,3)
1 0 0 1 0 1
1 1 1 0 0 1
0 0 1
1 0 0 0 0 0
0 1 0 1 1 1
0 1 0 1 0 1
1 0 0 1 1 0
1 1 1 1 0 0
1 0 0 1

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Partial Leaves

- How to guarantee that there are at most $n/f(n)$ of them?
- Upon insertion, look at the next $2f(n)$ leaves.
- If a partial one is found, propagate insertion up to it.
- Otherwise, propagate $f(n)$ times and create a partial leaf.
- When a partial leaf gets filled, it is not called partial anymore.
- Deletion is similar.
- When a partial leaf gets empty, it must be removed.

A Succinct Version

Tree Manipulation

- Each insertion/deletion requires at most one tree node insertion/deletion.
- But $O(f(n))$ subtree size/weight must be updated towards the root.
- Yet, those are contiguous, so there are $O(\log n + f(n))$ nodes involved.

A Succinct Version

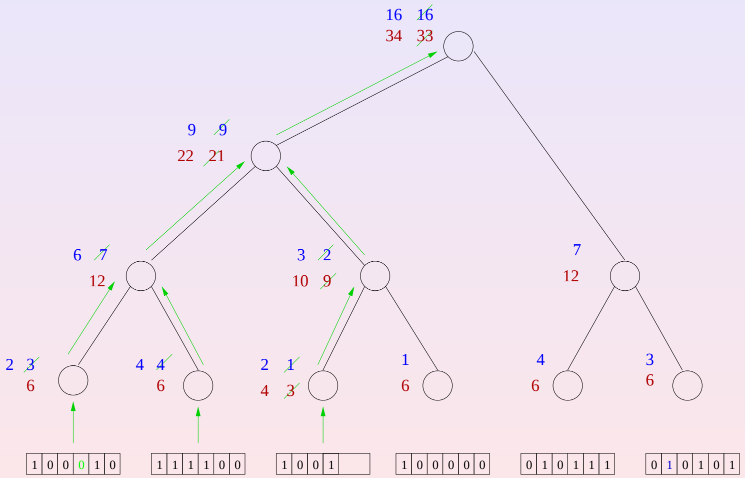
Tree Manipulation

- Each insertion/deletion requires at most one tree node insertion/deletion.
- But $O(f(n))$ subtree size/weight must be updated towards the root.
- Yet, those are contiguous, so there are $O(\log n + f(n))$ nodes involved.

A Succinct Version

Tree Manipulation

- Each insertion/deletion requires at most one tree node insertion/deletion.
- But $O(f(n))$ subtree size/weight must be updated towards the root.
- Yet, those are contiguous, so there are $O(\log n + f(n))$ nodes involved.


16 16
34 33
9 9
22 21
6 7
12
3 2
10 9
7
12
2 3
6
4 4
6
2 1
4 3
1
6
4
6
3
6
1 0 0 0 1 0
1 1 1 1 0 0
1 0 0 1
1 0 0 0 0 0
0 1 0 1 1 1
0 1 0 1 0 1

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

A Succinct Version

Analysis

- Space: $n + O(n/f(n))$ bits.
- Time: $O(\log n + f(n)^2)$ at most.
- We choose $f(n) = \sqrt{\log n}$ to achieve
  - $n + o(n)$ bits of space
  - $O(\log n)$ time for all operations
- Changing $\log n$: Too technical, see paper.

Outline

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - c = number of bits set in the block.
  - o = index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - $c =$ number of bits set in the block.
  - $o =$ index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - $c =$ number of bits set in the block.
  - $o =$ index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - $c =$ number of bits set in the block.
  - $o =$ index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - $c =$ number of bits set in the block.
  - $o =$ index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Identifier Encoding

- B is divided into blocks of $b = (\log n)/2$ bits.
- Each block is represented as a pair (c, o).
  - $c =$ number of bits set in the block.
  - $o =$ index of the block within those of class c.
- Overall space is $nH_0 + o(n)$.
- Inspired in Raman, Raman, and Rao's static solution.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n)\log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n) \log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n) \log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n) \log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n)\log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- Rank and select are again simple (with 4-Russian methods).
- A bit insertion can double the length of a block.
- Ex: 0000 1111 0000 1111 ...
  into 1000 0111 1000 0111 1....
- We can propagate up to $f(n) \log n$ bits to the next leaf.
- New leaves are created when there are sufficient bits for that.
- Within a propagation path, we can create $O(f(n))$ new leaves.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Identifier Encoding

Operations

- On a red-black tree this requires $O(f(n))$ time plus recolorings.
- Just as recomputing subtree size/weights, recoloring costs $O(\log n + f(n))$.
- Change of $\log n$, etc. very similar.
- We get our final result:
  - $nH_0 + o(n)$ bits of space.
  - $O(\log n)$ time for all operations.

Outline

Searchable partial sums with indels

The problem

- Sequence A of n numbers of k bits, $k = O(\log n)$.
- Sum: $\sum_{j=1}^{i} A_j$.
- Search: at which index the sum exceeds j.
- Insert and delete numbers.

Searchable partial sums with indels

The problem

- Sequence A of n numbers of k bits, $k = O(\log n)$.
- Sum: $\sum_{j=1}^{i} A_j$.
- Search: at which index the sum exceeds j.
- Insert and delete numbers.

Searchable partial sums with indels

The problem

- Sequence A of n numbers of k bits, $k = O(\log n)$.
- Sum: $\sum_{j=1}^{i} A_j$.
- Search: at which index the sum exceeds j.
- Insert and delete numbers.

Searchable partial sums with indels

The problem

- Sequence A of n numbers of k bits, $k = O(\log n)$.
- Sum: $\sum_{j=1}^{i} A_j$.
- Search: at which index the sum exceeds j.
- Insert and delete numbers.

Our solution

Applying our simplest structure

- We have nk bits, handled in chunks of k bits.
- Internal tree nodes store number and sum of subtree chunks.
- Sum is solved like *rank*, Search like *select*.
- Insertion and deletion of chunks is totally analogous to bits.
- Easy to solve all operations in $O(\log n)$ time.

Our solution

Applying our simplest structure

- We have nk bits, handled in chunks of k bits.
- Internal tree nodes store number and sum of subtree chunks.
- Sum is solved like *rank*, Search like *select*.
- Insertion and deletion of chunks is totally analogous to bits.
- Easy to solve all operations in $O(\log n)$ time.

Our solution

Applying our simplest structure

- We have nk bits, handled in chunks of k bits.
- Internal tree nodes store number and sum of subtree chunks.
- Sum is solved like *rank*, Search like *select*.
- Insertion and deletion of chunks is totally analogous to bits.
- Easy to solve all operations in $O(\log n)$ time.

Our solution

Applying our simplest structure

- We have nk bits, handled in chunks of k bits.
- Internal tree nodes store number and sum of subtree chunks.
- Sum is solved like *rank*, Search like *select*.
- Insertion and deletion of chunks is totally analogous to bits.
- Easy to solve all operations in $O(\log n)$ time.

Our solution

Applying our simplest structure

- We have nk bits, handled in chunks of k bits.
- Internal tree nodes store number and sum of subtree chunks.
- Sum is solved like *rank*, Search like *select*.
- Insertion and deletion of chunks is totally analogous to bits.
- Easy to solve all operations in $O(\log n)$ time.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\text{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log \log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\mathrm{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\mathrm{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\mathrm{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\text{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\mathrm{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Our solution

Comparison to the existing solution

- The best current result by Hon, Sadakane and Sung, ISAAC 2003.
- It requires $kn + o(kn)$ bits of space.
- Solves queries in $O(\log_b n)$ time for $b = \Omega(\mathrm{polylog}(n))$.
- But updates require $O(b)$ amortized time.
- b seems to be at least $\Theta((\log n / \log\log n)^2)$.
- Our query complexities are worse.
- But our update complexities are better, and worst-case.

Outline

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Structure

- By Grossi, Gupta, and Vitter, SODA 2003.
- Solves rank/select queries over a general sequence S.
- Let σ be the alphabet size.
- Balanced binary tree, dividing the alphabet into two at each node.
- Leaves correspond to individual symbols.
- Each node stores a bitmap telling which branch each position took.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log\sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n\log\sigma)$ bits of space...
- ... and it solves all operations in $O(\log\sigma)$ time.
- But the structure is static.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log \sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n \log \sigma)$ bits of space...
- ... and it solves all operations in $O(\log \sigma)$ time.
- But the structure is static.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log \sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n \log \sigma)$ bits of space...
- ... and it solves all operations in $O(\log \sigma)$ time.
- But the structure is static.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log \sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n \log \sigma)$ bits of space...
- ... and it solves all operations in $O(\log \sigma)$ time.
- But the structure is static.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log \sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n \log \sigma)$ bits of space...
- ... and it solves all operations in $O(\log \sigma)$ time.
- But the structure is static.

Wavelet trees

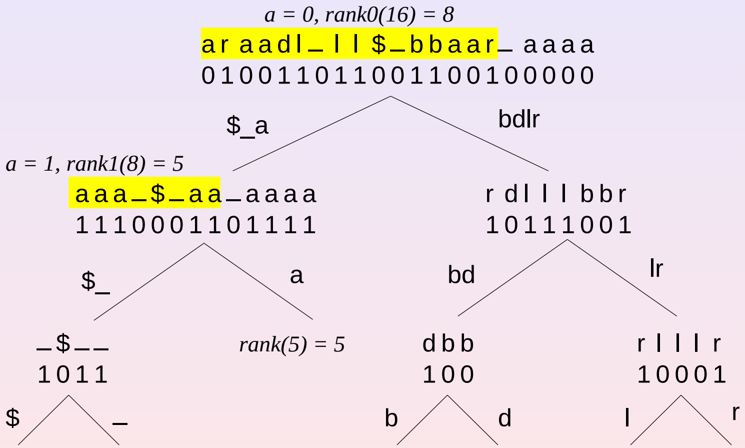
Operations

- A *rank* query over S is solved via $\log\sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n\log\sigma)$ bits of space...
- ... and it solves all operations in $O(\log\sigma)$ time.
- But the structure is static.

Wavelet trees

Operations

- A *rank* query over S is solved via $\log\sigma$ *rank*s over the bit vectors (top-down).
- Similarly with *select* (bottom-up).
- Symbol S_i can be discovered also with a top-down process.
- If the bit streams are represented using Raman et al's structure...
- ... the whole wavelet tree takes $nH_0(S) + o(n\log\sigma)$ bits of space...
- ... and it solves all operations in $O(\log\sigma)$ time.
- But the structure is static.


a = 0, rank0(16) = 8
ar aadl_ l l $_bbaar_ aaaa
0100110110011001000000
$_a
bdlr
a = 1, rank1(8) = 5
aaa_$_aa_aaaa
1110001101111
r dl l l bbr
10111001
$_
a
bd
lr
_$__
1011
rank(5) = 5
dbb
100
r l l l r
10001
$
_
b
d
l
r

Wavelet trees

A dynamic version

- Use our dynamic solution for bit vectors instead.
- The operations now require $O(\log n \log \sigma)$ time.
- Within the same time we can insert and delete symbols from S.
- The space is the same $nH_0(S) + o(n \log \sigma)$ bits.

Wavelet trees

A dynamic version

- Use our dynamic solution for bit vectors instead.
- The operations now require $O(\log n \log \sigma)$ time.
- Within the same time we can insert and delete symbols from S.
- The space is the same $nH_0(S) + o(n \log \sigma)$ bits.

Wavelet trees

A dynamic version

- Use our dynamic solution for bit vectors instead.
- The operations now require $O(\log n \log \sigma)$ time.
- Within the same time we can insert and delete symbols from S.
- The space is the same $nH_0(S) + o(n \log \sigma)$ bits.

Wavelet trees

A dynamic version

- Use our dynamic solution for bit vectors instead.
- The operations now require $O(\log n \log \sigma)$ time.
- Within the same time we can insert and delete symbols from S.
- The space is the same $nH_0(S) + o(n \log \sigma)$ bits.

Outline

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer $rank$ queries over $bwt(T)$.
- One can count the occurrences of P in T with m $rank$s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer *rank* queries over $bwt(T)$.
- One can count the occurrences of P in T with m *rank*s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer $rank$ queries over $bwt(T)$.
- One can count the occurrences of P in T with m $rank$s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer $rank$ queries over $bwt(T)$.
- One can count the occurrences of P in T with m $rank$s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer *rank* queries over $bwt(T)$.
- One can count the occurrences of P in T with m *rank*s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer *rank* queries over $bwt(T)$.
- One can count the occurrences of P in T with m *rank*s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

FM-indexes

The FM-index concept

- By Ferragina and Manzini, FOCS 2000
- Let T be a text (sequence) of length n.
- Let P be a pattern (sequence) of length $m << n$.
- Apply the Burrows-Wheeler Transform (a permutation) to T, $bwt(T)$.
- Answer *rank* queries over $bwt(T)$.
- One can count the occurrences of P in T with m *rank*s on $bwt(T)$.
- To locate the occurrences: sample the suffix array.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A simple realization

- A wavelet tree on $S = bwt(T)$.
- Space is $nH_0(T) + o(n \log \sigma)$.
- Time for counting is $O(m \log \sigma)$.
- But it is static.
- We would like to handle a collection of texts $\mathcal{C}$:
  - Insert a new text to $\mathcal{C}$.
  - Delete a text from $\mathcal{C}$.
  - Count/locate occurrences of P in $\mathcal{C}$.

Text indexes

A dynamic version

- By Chan, Hon, and Lam, CPM 2004.
- Two versions

Aspect	Version 1	Version 2
Space	$O(n\sigma)$	$O(n \log \sigma)$
Count	$m \log n$	$m \log^2 n$
Locate	$\log^2 n$	$\log^2 n$
Insert	$\sigma \log n$	$\log n$
Delete	$\sigma \log^2 n$	$\log n$

Text indexes

A dynamic version

- By Chan, Hon, and Lam, CPM 2004.
- Two versions

Aspect	Version 1	Version 2
Space	$O(n\sigma)$	$O(n \log \sigma)$
Count	$m \log n$	$m \log^2 n$
Locate	$\log^2 n$	$\log^2 n$
Insert	$\sigma \log n$	$\log n$
Delete	$\sigma \log^2 n$	$\log n$

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Our dynamic version

- A dynamic wavelet tree over S.
- Counting can be done in $O(m \log n \log \sigma)$ time.
- Locating takes $O(\log^2 n \log \log n)$ per occurrence.
- Insert/delete: $O(\log n \log \sigma)$ time.
- The space is now $nH_0(\mathcal{C}) + o(n \log \sigma)$ bits.
- It permits constructing the FM-index for T in $O(n \log n \log \sigma)$ time and $nH_0(T) + o(n \log \sigma)$ bits of space.

Text indexes

Higher-order entropy (newer than the paper)

- Very recent result:
  The wavelet tree of $bwt(T)$, compressed with Raman et al.'s technique, takes actually $nH_h(T) + o(n \log \sigma)$ bits of space, for any $h \leq \alpha \log_\sigma n$, $\alpha < 1$.
- So does our index, as well as the simple static version presented.
- So does our construction space.
- Time complexities do not change.

Text indexes

Higher-order entropy (newer than the paper)

- Very recent result:
  The wavelet tree of $bwt(T)$, compressed with Raman et al.'s technique, takes actually $nH_h(T) + o(n \log \sigma)$ bits of space, for any $h \leq \alpha \log_\sigma n$, $\alpha < 1$.
- So does our index, as well as the simple static version presented.
- So does our construction space.
- Time complexities do not change.

Text indexes

Higher-order entropy (newer than the paper)

- Very recent result:
  The wavelet tree of $bwt(T)$, compressed with Raman et al.'s technique, takes actually $nH_h(T) + o(n \log \sigma)$ bits of space, for any $h \leq \alpha \log_\sigma n$, $\alpha < 1$.
- So does our index, as well as the simple static version presented.
- So does our construction space.
- Time complexities do not change.

Text indexes

Higher-order entropy (newer than the paper)

- Very recent result:
  The wavelet tree of $bwt(T)$, compressed with Raman et al.'s technique, takes actually $nH_h(T) + o(n \log \sigma)$ bits of space, for any $h \leq \alpha \log_\sigma n$, $\alpha < 1$.
- So does our index, as well as the simple static version presented.
- So does our construction space.
- Time complexities do not change.

Summary

Main Results

- First dynamic compressed structure for bit sequences, supporting *rank*, *select*, *insert* and *delete*, using nH_0 bits.
- Improved solutions to searchable partial sums with indels.
- First compressed structure for general sequences: Dynamic wavelet trees.
- First dynamic compressed index for text collections taking nH_h bits of space.
- First compressed construction of full-text indexes taking nH_h bits of space.

Summary

Main Results

- First dynamic compressed structure for bit sequences, supporting *rank*, *select*, *insert* and *delete*, using nH_0 bits.
- Improved solutions to searchable partial sums with indels.
- First compressed structure for general sequences: Dynamic wavelet trees.
- First dynamic compressed index for text collections taking nH_h bits of space.
- First compressed construction of full-text indexes taking nH_h bits of space.

Summary

Main Results

- First dynamic compressed structure for bit sequences, supporting *rank*, *select*, *insert* and *delete*, using nH_0 bits.
- Improved solutions to searchable partial sums with indels.
- First compressed structure for general sequences: Dynamic wavelet trees.
- First dynamic compressed index for text collections taking nH_h bits of space.
- First compressed construction of full-text indexes taking nH_h bits of space.

Summary

Main Results

- First dynamic compressed structure for bit sequences, supporting *rank*, *select*, *insert* and *delete*, using nH_0 bits.
- Improved solutions to searchable partial sums with indels.
- First compressed structure for general sequences: Dynamic wavelet trees.
- First dynamic compressed index for text collections taking nH_h bits of space.
- First compressed construction of full-text indexes taking nH_h bits of space.

Summary

Main Results

- First dynamic compressed structure for bit sequences, supporting *rank*, *select*, *insert* and *delete*, using nH_0 bits.
- Improved solutions to searchable partial sums with indels.
- First compressed structure for general sequences: Dynamic wavelet trees.
- First dynamic compressed index for text collections taking nH_h bits of space.
- First compressed construction of full-text indexes taking nH_h bits of space.