Paweł Gawrychowski\textsuperscript{2}, Tomasz Kociumaka\textsuperscript{1}, Adam Karczmarz\textsuperscript{1}, Jakub Łącki\textsuperscript{3}, Piotr Sankowski\textsuperscript{1}

\textsuperscript{1}University of Warsaw, Poland
\textsuperscript{2}University of Wrocław, Poland
\textsuperscript{3}Google Research NY, USA

SODA 2018

New Orleans, LA, USA
January 9, 2018
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- $\text{make}(w) = \text{insert } w \in \Sigma^+ \text{ to } \mathcal{W}$;
- $\text{concat}(w_1, w_2) = \text{insert } w_1 w_2 \text{ to } \mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- $\text{split}(w, k) = \text{insert } w[1..k] \text{ and } w[k+1..|w|] \text{ for } w \in \mathcal{W}$;
- $\text{equal}(w_1, w_2) = \text{check whether } w_1 \text{ is the same string as } w_2$;
- $\text{LCP}(w_1, w_2) = \text{return the length of the longest common prefix of } w_1 \text{ and } w_2$;
- $\text{compare}(w_1, w_2) = \text{lexicographically compare } w_1 \text{ and } w_2$. 

$\mathcal{W}: ab$ $1$ $abab$ $2$ $a$ $3$ $bab$ $4$ $ba$ $5$ $b$ $6$ $ba$ $7$ $baab$ $8$
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- **make_string($w$)**: insert $w \in \Sigma^+$ to $\mathcal{W}$;

\[
\mathcal{W}: \quad \begin{array}{l}
ab \\
1
\end{array}
\]

make_string(ab) = 1
Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- $\text{make\_string}(w)$: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- $\text{concat}(w_1, w_2)$: insert $w_1 w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;

$\mathcal{W}$:

```
ab  abab
1   2
```

$\text{concat}(1, 1) = 2$
Dynamic Strings Problem

Maintain a multiset \( \mathcal{W} \) of non-empty strings subject to:

- \texttt{make\_string}(w): insert \( w \in \Sigma^+ \) to \( \mathcal{W} \);
- \texttt{concat}(w_1, w_2): insert \( w_1w_2 \) to \( \mathcal{W} \) for \( w_1, w_2 \in \mathcal{W} \);
- \texttt{split}(w, k): insert \( w[1..k] \) and \( w[k+1..|w|] \) for \( w \in \mathcal{W} \);

\[
\mathcal{W}: \quad \begin{array}{cccc}
  \text{ab} & \text{abab} & \text{a} & \text{bab} \\
  1 & 2 & 3 & 4 \\
\end{array}
\]

\texttt{split}(2, 1) = (3, 4)
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- **make_string(w)**: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- **concat($w_1, w_2$)**: insert $w_1w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- **split($w, k$)**: insert $w[1..k]$ and $w[k + 1..|w|]$ for $w \in \mathcal{W}$;
- **equal($w_1, w_2$)**: check whether $w_1$ is the same string as $w_2$;

$\mathcal{W}$: ab abab a bab

$\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}$

$\text{equal}(1, 3) = \text{false}$
Dynamic Strings Problem

Maintain a multiset \( \mathcal{W} \) of non-empty strings subject to:
- **make_string** \( w \): insert \( w \in \Sigma^+ \) to \( \mathcal{W} \);
- **concat** \( w_1, w_2 \): insert \( w_1 w_2 \) to \( \mathcal{W} \) for \( w_1, w_2 \in \mathcal{W} \);
- **split** \( w, k \): insert \( w[1..k] \) and \( w[k + 1..|w|] \) for \( w \in \mathcal{W} \);
- **equal** \( w_1, w_2 \): check whether \( w_1 \) is the same string as \( w_2 \);

\[
\mathcal{W}: \quad \text{ab} \quad \text{abab} \quad \text{a} \quad \text{bab} \quad \text{ba} \quad \text{b}
\]

1 \hspace{1em} 2 \hspace{1em} 3 \hspace{1em} 4 \hspace{1em} 5 \hspace{1em} 6

\[
\text{split}(4, 2) = (5, 6)
\]
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- $\text{make\_string}(w)$: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- $\text{concat}(w_1, w_2)$: insert $w_1 w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- $\text{split}(w, k)$: insert $w[1..k]$ and $w[k + 1..|w|]$ for $w \in \mathcal{W}$;
- $\text{equal}(w_1, w_2)$: check whether $w_1$ is the same string as $w_2$;

$\mathcal{W}$:

\[
\begin{array}{cccccccc}
\text{\#} & \text{ab} & \text{abab} & \text{a} & \text{bab} & \text{ba} & \text{b} & \text{ba} \\
1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
\]

$\text{concat}(6, 3) = 7$
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- $\text{make\_string}(w)$: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- $\text{concat}(w_1, w_2)$: insert $w_1 w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- $\text{split}(w, k)$: insert $w[1..k]$ and $w[k + 1..|w|]$ for $w \in \mathcal{W}$;
- $\text{equal}(w_1, w_2)$: check whether $w_1$ is the same string as $w_2$;

$\mathcal{W}$:  

```
ab  abab  a  bab  ba  b  ba
```

```
1  2  3  4  5  6  7
```

equal(5, 7) = true
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- **make_string($w$)**: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- **concat($w_1, w_2$)**: insert $w_1 w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- **split($w, k$)**: insert $w[1..k]$ and $w[k + 1..|w|]$ for $w \in \mathcal{W}$;
- **equal($w_1, w_2$)**: check whether $w_1$ is the same string as $w_2$;

$\mathcal{W}$: ab abab a bab ba b ba baab

$$\text{concat}(7, 1) = 8$$
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- **make_string($w$):** insert $w \in \Sigma^+$ to $\mathcal{W}$;
- **concat($w_1, w_2$):** insert $w_1w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- **split($w, k$):** insert $w[1..k]$ and $w[k + 1..|w|]$ for $w \in \mathcal{W}$;
- **equal($w_1, w_2$):** check whether $w_1$ is the same string as $w_2$;
- **LCP($w_1, w_2$):** return the length of the longest common prefix of $w_1$ and $w_2$;

$\mathcal{W}$:

$$\begin{align*}
\mathcal{W}: & \quad \text{ab} \quad \text{abab} \quad \text{a} \quad \text{bab} \quad \text{ba} \quad \text{b} \quad \text{ba} \quad \text{baab} \\
& \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8
\end{align*}$$

$LCP(4, 8) = 2$
Dynamic Strings Problem

Maintain a multiset $\mathcal{W}$ of non-empty strings subject to:

- $\text{make\_string}(w)$: insert $w \in \Sigma^+$ to $\mathcal{W}$;
- $\text{concat}(w_1, w_2)$: insert $w_1w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;
- $\text{split}(w, k)$: insert $w[1..k]$ and $w[k+1..|w|]$ for $w \in \mathcal{W}$;
- $\text{equal}(w_1, w_2)$: check whether $w_1$ is the same string as $w_2$;
- $\text{LCP}(w_1, w_2)$: return the length of the longest common prefix of $w_1$ and $w_2$;
- $\text{compare}(w_1, w_2)$: lexicographically compare $w_1$ and $w_2$.

$\mathcal{W}$:

$\begin{array}{cccccccc}
\text{ab} & \text{abab} & \text{a} & \text{bab} & \text{ba} & \text{b} & \text{ba} & \text{baab} \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}$

$\text{compare}(4, 8) = '>'$
Previous & Our Results

Setting

$n =$ total length of strings in the collection
word RAM machine with word size $\Omega(\log n)$
Setting

\( n = \) total length of strings in the collection

word RAM machine with word size \( \Omega(\log n) \)

<table>
<thead>
<tr>
<th></th>
<th>randomization</th>
<th>split, concat</th>
<th>equal</th>
<th>compare, LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>,,folklore”</td>
<td>Monte Carlo</td>
<td>( \mathcal{O}(\log n) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(\log^2 n) )</td>
</tr>
</tbody>
</table>
**Setting**

\[ n = \text{total length of strings in the collection} \]

word RAM machine with word size \( \Omega(\log n) \)

<table>
<thead>
<tr>
<th></th>
<th>randomization</th>
<th>split, concat</th>
<th>equal</th>
<th>compare, LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>„folklore“</td>
<td>Monte Carlo</td>
<td>( \mathcal{O}(\log n) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(\log^2 n) )</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>Las Vegas</td>
<td>( \mathcal{O}(\log^2 n) ) exp.</td>
<td>( \mathcal{O}(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>deterministic</td>
<td>( \mathcal{O}(\log^2 n \log^* n) )</td>
<td>( \mathcal{O}(1) )</td>
<td>-</td>
</tr>
</tbody>
</table>

† due to dynamic dictionaries (hash tables) only.
### Setting

\( n = \) total length of strings in the collection

word RAM machine with word size \( \Omega(\log n) \)

<table>
<thead>
<tr>
<th></th>
<th>randomization</th>
<th>split, concat</th>
<th>equal</th>
<th>compare, LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>„folklore“</td>
<td>Monte Carlo</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(\log^2 n) )</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>Las Vegas</td>
<td>( O(\log^2 n) ) exp.</td>
<td>( O(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>deterministic</td>
<td>( O(\log^2 n \log^* n) )</td>
<td>( O(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Alstrup et al. 2000</td>
<td>Las Vegas(^\dagger)</td>
<td>( O(\log n \log^* n) ) w.h.p.</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

\(^\dagger\) due to dynamic dictionaries (hash tables) only.
## Previous & Our Results

### Setting

\[ n = \text{total length of strings in the collection} \]

word RAM machine with word size \( \Omega(\log n) \)

<table>
<thead>
<tr>
<th>Setting</th>
<th>randomization</th>
<th>split, concat</th>
<th>equal</th>
<th>compare, LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>„folklore“</td>
<td>Monte Carlo</td>
<td>( O(\log n) )</td>
<td>( O(1) )</td>
<td>( O(\log^2 n) )</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>Las Vegas</td>
<td>( O(\log^2 n) ) exp.</td>
<td>( O(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>deterministic</td>
<td>( O(\log^2 n \log^* n) )</td>
<td>( O(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Alstrup et al. 2000</td>
<td>Las Vegas(^\dagger)</td>
<td>( O(\log n \log^* n) ) w.h.p.</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>this work</td>
<td>Las Vegas</td>
<td>( O(\log n) ) w.h.p.</td>
<td>( O(1) )</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

\(^\dagger\)due to dynamic dictionaries (hash tables) only.
**Setting**

\( n = \) total length of strings in the collection

word RAM machine with word size \( \Omega(\log n) \)

<table>
<thead>
<tr>
<th></th>
<th>randomization</th>
<th>split, concat</th>
<th>equal</th>
<th>compare, LCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>„folklore“</td>
<td>Monte Carlo</td>
<td>( \mathcal{O}(\log n) )</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(\log^2 n) )</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>Las Vegas</td>
<td>( \mathcal{O}(\log^2 n) ) exp.</td>
<td>( \mathcal{O}(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Mehlhorn et al. 1994</td>
<td>deterministic</td>
<td>( \mathcal{O}(\log^2 n \log^* n) )</td>
<td>( \mathcal{O}(1) )</td>
<td>-</td>
</tr>
<tr>
<td>Alstrup et al. 2000</td>
<td>Las Vegas(^\dagger)</td>
<td>( \mathcal{O}(\log n \log^* n) ) w.h.p.</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
<tr>
<td>this work</td>
<td>Las Vegas</td>
<td>( \mathcal{O}(\log n) ) w.h.p.</td>
<td>( \mathcal{O}(1) )</td>
<td>( \mathcal{O}(1) )</td>
</tr>
</tbody>
</table>

\(^\dagger\) due to dynamic dictionaries (hash tables) only.

---

**Unconditional Lower Bound (this work)**

No Monte Carlo algorithm (correct with high probability) supports split, concat, and equal operations in \( o(\log n) \) amortized time. This is true even if split and concat invalidate their arguments.
Karp–Rabin Fingerprints

- \( H(w) = H(w') \implies w = w' \) w.h.p.
- \( H(u), H(v) \rightsquigarrow H(uv) \) in \( O(1) \) time.
Karp–Rabin Fingerprints

- $H(w) = H(w') \implies w = w'$ w.h.p.
- $H(u), H(v) \rightsquigarrow H(uv)$ in $O(1)$ time.

Idea: store each string in a persistent BST supporting split & join.
Folklore” Solution

Karp–Rabin Fingerprints

- $H(w) = H(w') \implies w = w' \text{ w.h.p.}$
- $H(u), H(v) \leadsto H(uv) \text{ in } \mathcal{O}(1) \text{ time.}$

Idea: store each string in a persistent BST supporting split & join.
Karp–Rabin Fingerprints

- $H(w) = H(w') \implies w = w' \text{ w.h.p.}$
- $H(u), H(v) \leadsto H(uv) \text{ in } O(1) \text{ time.}$

Idea: store each string in a persistent BST supporting split & join.

\[
\begin{array}{c}
\text{split}(w, k) \quad O(\log |w|) \\
\text{concat}(w_1, w_2) \quad O(\log |w_1w_2|)
\end{array}
\]

\[
\begin{array}{c}
\text{make\_string}(w) \quad O(|w|) \\
\text{equal}(w_1, w_2) \quad O(1)
\end{array}
\]
Monte Carlo: equal is correct w.h.p. only
“Folklore” Solution: Issues

1. Monte Carlo: equal is correct w.h.p. only
2. Answering $\text{LCP}(w_1, w_2)$ and $\text{compare}(w_1, w_2)$ is slow.
1. Monte Carlo: `equal` is correct w.h.p. only
2. Answering $\text{LCP}(w_1, w_2)$ and $\text{compare}(w_1, w_2)$ is slow.
1. Monte Carlo: equal is correct w.h.p. only
2. Answering \( \text{LCP}(w_1, w_2) \) and \( \text{compare}(w_1, w_2) \) is slow.

- The BST structure is not particularly helpful...
- Fall back to the naive solution:
  - LCP: binary search using split and equal; \( \mathcal{O}(\log^2 |w|) \) time;
  - compare: compare the characters following LCP.
Consistent Parsing (Mehlhorn et al. SODA’94; Sahinalp–Vishkin STOC’94)

Shape of the parse tree is determined by the underlying string.
Consistent Parsing (Mehlhorn et al. SODA’94; Sahinalp-Vishkin STOC’94)

Shape of the parse tree is determined by the underlying string.
Consistent Parsing (Mehlhorn et al. SODA’94; Sahinalp–Vishkin STOC’94)

Shape of the parse tree is determined by the underlying string.

- Name each node based on the names of its children.
Consistent Parsing (Mehlhorn et al. SODA’94; Sahinalp–Vishkin STOC’94)

Shape of the parse tree is determined by the underlying string.

- Name each node based on the names of its children.
- Compare the roots’ names for equal with no errors.
Consistent Parsing (Mehlhorn et al. SODA’94; Sahinalp–Vishkin STOC’94)

Shape of the parse tree is determined by the underlying string.

- Name each node based on the names of its children.
- Compare the roots’ names for equal with no errors.
- Dictionaries needed for the naming function.
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

Why is local constinance useful?
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

Why is local consistence useful?
- allows to maintain consistent parsing,
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

Why is local consistence useful?

- allows to maintain consistent parsing,
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

Why is local consistence useful?

- allows to maintain consistent parsing,
- enables efficient LCP (and compare) queries.
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

Run-Length Encoding (RLE)

Replace each run with a new symbol:

\[ \text{a} \cdots \text{a} \mapsto (\text{a}, k). \]

\[
\text{aaabbabaabbbaa} \mapsto (\text{a}, 3) (\text{b}, 2) (\text{a}, 1) (\text{b}, 1) (\text{a}, 2) (\text{b}, 3) (\text{a}, 2).
\]
How to parse highly-repetitive fragments?

\[
\text{aaaababaabbaa}
\]

\[
\quad \rightarrow \quad \langle a, 3 \rangle \langle b, 2 \rangle \langle a, 1 \rangle \langle b, 1 \rangle \langle a, 2 \rangle \langle b, 3 \rangle \langle a, 2 \rangle
\]
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

\[ \text{aaabbabaabbbaa} \mapsto (a, 3)(b, 2)(a, 1)(b, 1)(a, 2)(b, 3)(a, 2) \]
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

```
  a  a  a  a  a  a  a  a  a  a  a
```

Run-Length Encoding (RLE)
Replace each run with a new symbol:

```
(\text{a}, 3) (\text{b}, 2) (\text{a}, 1) (\text{b}, 1) (\text{a}, 2) (\text{b}, 3) (\text{a}, 2)
```
How to parse highly-repetitive fragments?

Run-Length Encoding (RLE)

Replace each run with a new symbol:

\[ a \cdots a k \times \mapsto (a, k), (b, 2), (a, 1), (b, 1), (a, 2), (b, 3), (a, 2) \]

a a a a a a a a
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

\[
\begin{array}{cccccccc}
\text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} & \text{a} \\
\end{array}
\]
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

Run-Length Encoding (RLE)

Replace each run with a new symbol: \( a \cdots a \mapsto (a, k). \)

\[ aaabbabaabbbbaa \mapsto (a, 3)(b, 2)(a, 1)(b, 1)(a, 2)(b, 3)(a, 2) \]
Parse Tree Construction: COMPRESS

\[
A = (a, 1), \quad B = (b, 1), \quad C = (a, 2)
\]

RLE

Mehlhorn et al. Local minima wrt. a random order:

- Block length: \(2 - O(\log n)\) w.h.p., \(O(1)\) in expectation.
- Context size: \(O(1)\).

Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):

- Blocks length: 2–4; context size: \(O(\log^* n)\).

Reused by Alstrup et al.

Jeż (Recompression) Alphabet partitioning:

- Partition the alphabet into left and right symbols.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
- Context size: \(O(1)\).
How to **consistently** partition a string without non-trivial runs?

RLE

\[ A = (a, 1), \ B = (b, 1), \ C = (a, 2) \]
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2 - O(\log n)$ w.h.p., $O(1)$ in expectation.
- Context size: $O(1)$.
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2 - O(\log n)$ w.h.p., $O(1)$ in expectation.
- Context size: $O(1)$.
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2 - \mathcal{O}(\log n)$ w.h.p., $\mathcal{O}(1)$ in expectation.
- Context size: $\mathcal{O}(1)$.

Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: 2–4; context size: $\mathcal{O}(\log^* n)$. 

A = (a, 1), B = (b, 1), C = (a, 2)
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2-O(\log n)$ w.h.p., $O(1)$ in expectation.
- Context size: $O(1)$.

Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: $2-4$; context size: $O(\log^* n)$.
How to **consistently** partition a string without non-trivial runs?

**Mehlhorn et al.** Local minima wrt. a random order:
- Block length: $2–\mathcal{O}(\log n)$ w.h.p., $\mathcal{O}(1)$ in expectation.
- Context size: $\mathcal{O}(1)$.

**Mehlhorn et al.** Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: $2–4$; context size: $\mathcal{O}(\log^* n)$.
- Reused by Alstrup et al.
How to **consistently** partition a string without non-trivial runs?

**Mehlhorn et al.** Local minima wrt. a **random order**:
- Block length: $2–\mathcal{O}(\log n)$ w.h.p., $\mathcal{O}(1)$ in expectation.
- Context size: $\mathcal{O}(1)$.

**Mehlhorn et al.** **Deterministic coin tossing** (by Cole & Vishkin):
- Blocks length: $2–4$; context size: $\mathcal{O}(\log^* n)$.
- Reused by Alstrup et al.

**Jeż (Recompression)** Alphabet partitioning:
- Partition the alphabet into **left** and **right** symbols.
Parse Tree Construction: COMPRESS

RLE

\[ A = (a, 1), \ B = (b, 1), \ C = (a, 2) \]

How to \textit{consistently} partition a string without non-trivial runs?

\textbf{Mehlhorn et al.} Local minima wrt. a random order:
- Block length: \( 2-O(\log n) \) w.h.p., \( O(1) \) in expectation.
- Context size: \( O(1) \).

\textbf{Mehlhorn et al.} Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: 2–4; context size: \( O(\log^* n) \).
- Reused by Alstrup et al.

\textbf{Jeż (Recompression)} Alphabet partitioning:
- Partition the alphabet into left and right symbols.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2 - O(\log n)$ w.h.p., $O(1)$ in expectation.
- Context size: $O(1)$.

Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: 2–4; context size: $O(\log^* n)$.
- Reused by Alstrup et al.

Jeż (Recompression) Alphabet partitioning:
- Partition the alphabet into left and right symbols.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
- Context size: $O(1)$.
How to consistently partition a string without non-trivial runs?

Mehlhorn et al. Local minima wrt. a random order:
- Block length: $2–O(\log n)$ w.h.p., $O(1)$ in expectation.
- Context size: $O(1)$.

Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):
- Blocks length: $2–4$; context size: $O(\log^* n)$.
- Reused by Alstrup et al.

Jeż (Recompression) Alphabet partitioning:
- Partition the alphabet into left and right symbols.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
- Context size: $O(1)$.
Parse Tree Construction: Example

A = (a, 1), B = (b, 1), C = (a, 2)

RLE

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski
Optimal Dynamic Strings
Parse Tree Construction: Example

RLE
A = (a, 1), B = (b, 1), C = (a, 2)
Parse Tree Construction: Example

\[ A = (a, 1), B = (b, 1), C = (a, 2) \]
P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski
Optimal Dynamic Strings

Parse Tree Construction: Example

RLE
\[ G = (D, 1), H = (E, 1), I = (F, 2) \]

COMPRESS
\[ D = (A), E = (B, C), F = (B, A) \]

RLE
\[ A = (a, 1), B = (b, 1), C = (a, 2) \]
Parse Tree Construction: Example

\[
\begin{align*}
\text{RLE} & \quad \text{A} = (a, 1), \text{B} = (b, 1), \text{C} = (a, 2) \\
\text{RLE} & \quad \text{D} = (A), \text{E} = (B, C), \text{F} = (B, A) \\
\text{COMPRESS} & \quad \text{J} = (\text{G}, \text{H}), \text{K} = (\text{I}) \\
\text{RLE} & \quad \text{G} = (\text{D}, 1), \text{H} = (\text{E}, 1), \text{I} = (\text{F}, 2)
\end{align*}
\]
Parse Tree Construction: Example

\[
\begin{align*}
    A &= (a, 1) \\
    B &= (b, 1) \\
    C &= (a, 2)
\end{align*}
\]

\[
\begin{align*}
    D &= (A) \\
    E &= (B, C) \\
    F &= (B, A)
\end{align*}
\]

\[
\begin{align*}
    G &= (D, 1) \\
    H &= (E, 1) \\
    I &= (F, 2)
\end{align*}
\]

\[
\begin{align*}
    J &= (G, H) \\
    K &= (I)
\end{align*}
\]

\[
\begin{align*}
    L &= (J, 1) \\
    M &= (K, 1)
\end{align*}
\]

Compress

RLE

\[
\begin{align*}
    L &= (J, 1), M = (K, 1) \\
    J &= (G, H), K &= (I) \\
    G &= (D, 1), H &= (E, 1), I &= (F, 2) \\
    D &= (A), E &= (B, C), F &= (B, A) \\
    A &= (a, 1), B &= (b, 1), C &= (a, 2)
\end{align*}
\]
Parse Tree Construction: Example

\[
\begin{align*}
N &= (L, M) \\
RLE \\
L &= (J, 1), M = (K, 1) \\
COMPRESS \\
J &= (G, H), K = (I) \\
RLE \\
G &= (D, 1), H = (E, 1), I = (F, 2) \\
COMPRESS \\
D &= (A), E = (B, C), F = (B, A) \\
RLE \\
A &= (a, 1), B = (b, 1), C = (a, 2)
\end{align*}
\]
How to partition the alphabet at each level?
How to partition the alphabet at each level?

Jeż To guarantee compression ratio $r < 1$
How to partition the alphabet at each level?

Jeż To guarantee compression ratio $r < 1$
  - Works well in the static setting!
How to partition the alphabet at each level?

**Jeż** To guarantee compression ratio $r < 1$
- Works well in the static setting!
- Permanent decisions prone to be exploited by an adversary.
How to partition the alphabet at each level?

**Jeż** To guarantee compression ratio $r < 1$
- Works well in the static setting!
- Permanent decisions prone to be exploited by an adversary.

**This work** Uniformly at random!
- Adjacent symbols form a block with probability $\frac{1}{4}$.
How to partition the alphabet at each level?

**Jeż** To guarantee compression ratio $r < 1$
- Works well in the static setting!
- Permanent decisions prone to be exploited by an adversary.

**This work** Uniformly at random!
- Adjacent symbols form a block with probability $\frac{1}{4}$.
- **Expected** compression ratio: $\frac{3}{4}$.
How to partition the alphabet at each level?

**Jeż** To guarantee compression ratio $r < 1$

- Works well in the static setting!
- Permanent decisions prone to be exploited by an adversary.

**This work** Uniformly at random!

- Adjacent symbols form a block with probability $\frac{1}{4}$.
- **Expected** compression ratio: $\frac{3}{4}$.

---

**Lemma**

*For any string $w$ and any $d \in \mathbb{R}_{\geq 0}$:*

$$\mathbb{P}[\text{DEPTH}(w) \leq 8(d + \ln |w|)] \geq 1 - e^{-d}$$

In short: depth $\mathcal{O}(\log n)$ with high probability.
Difficulties

$\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^\epsilon}$. Adding a letter might affect $\Theta(\log^2 n)$ nodes.

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski

Optimal Dynamic Strings
Difficulties

$\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n \epsilon}$.

Appending a letter might affect $\Theta(\log^2 n)$ nodes.
Difficulties

$\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^\varepsilon}$.

Appending a letter might affect $\Theta(\log^2 n)$ nodes.

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski

Optimal Dynamic Strings

12/19
Difficulties

\[ \Theta(n \log n) \text{ nodes with non-negligible probability} \]

Appending a letter might affect \( \Theta(\log^2 n) \) nodes.

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski
Optimal Dynamic Strings
\[\Theta(n \log n)\] nodes with non-negligible probability \(\frac{1}{n^\varepsilon}\).
\[ \Theta(n \log n) \text{ nodes with non-negligible probability } \frac{1}{n^\varepsilon}. \]
$\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^\varepsilon}$.
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^\varepsilon}$.
Difficulties

\[ \Theta(n \log n) \text{ nodes with non-negligible probability } \frac{1}{n^{\varepsilon}}. \]
\( \Theta(n \log n) \) nodes with non-negligible probability \( \frac{1}{n^\varepsilon} \).
\( \Theta(n \log n) \) nodes with non-negligible probability \( \frac{1}{n^\varepsilon} \).

- Appending a letter might affect \( \Theta(\log^2 n) \) nodes.
Workaround: Compress the Parse Tree

\[
\text{Compress} \\
N = (L, M) \\
\text{RLE} \\
L = (J, 1), M = (K, 1) \\
\text{Compress} \\
J = (G, H), K = (I) \\
\text{RLE} \\
G = (D, 1), H = (E, 1), I = (F, 2) \\
\text{Compress} \\
D = (A), E = (B, C), F = (B, A) \\
\text{RLE} \\
A = (a, 1), B = (b, 1), C = (a, 2)
\]
Workaround: Compress the Parse Tree

- Do not replace $S$ with $(S, 1)$ in RLE.
Workaround: Compress the Parse Tree

- N
  - J
    - D
      - a
      - b
    - E
      - a
    - C
      - a
  - K
    - F
      - F
      - b
      - a

Do not replace S with (S, 1) in RLE.

Compress
N = (J, K)
RLE

Compress
J = (D, E), K = (I)
RLE
I = (F, 2)
Compress
D = (a), E = (b, C), F = (b, a)
RLE
C = (a, 2)

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski
Optimal Dynamic Strings
Do not replace $S$ with $(S, 1)$ in RLE.
Do not introduce new symbols for unary blocks in COMPRESS.
Workaround: Compress the Parse Tree

- Do not replace $S$ with $(S, 1)$ in RLE.
- Do not introduce new symbols for unary blocks in COMPRESS.
Workaround: Compress the Parse Tree

- Do not replace $S$ with $(S, 1)$ in RLE.
- Do not introduce new symbols for unary blocks in Compress.
- Each symbol has a level where it appears . . .
Do not replace $S$ with $(S, 1)$ in RLE.
Do not introduce new symbols for unary blocks in \texttt{Compress}.
Each symbol has a level where it appears . . .
and a random bit for each larger even level ($\mathcal{O}(\log n)$ w.h.p.).
Navigating Uncompressed Parse Trees

Navigate the uncompressed parse trees in $O(1)$ time:
- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigating the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the _uncompressed parse trees_ in $\mathcal{O}(1)$ time:

- Traverse edges: go to the parent or to the $k$-th child
- Traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the **uncompressed parse trees** in $\mathcal{O}(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the **uncompressed parse trees** in $\mathcal{O}(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- traverse edges: go to the parent or to the $k$-th child
- traverse levels: go left or right, perhaps skipping runs
Navigate the uncompressed parse trees in $O(1)$ time:

- Traverse edges: go to the parent or to the $k$-th child
- Traverse levels: go left or right, perhaps skipping runs
Simple LCP and compare Implementation

Start at the leftmost leaves and go up as far as possible.

Go right as far as possible, to the last child, and repeat.

Fact. While going to the right, at most run is traversed.

Total running time: $O(\min(Depth(w_1), Depth(w_2)))$. 

P. Gawrychowski, T. Kociumaka, A. Karczmarz, J. Łącki, P. Sankowski

Optimal Dynamic Strings
Start at the leftmost leaves and go up as far as possible.
Start at the leftmost leaves and go up as far as possible.
Start at the leftmost leaves and go up as far as possible.
Simple LCP and compare Implementation

1. Start at the leftmost leaves and go up as far as possible.
Start at the leftmost leaves and go up as far as possible.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
Start at the leftmost leaves and go up as far as possible.

Go right as far as possible, to the last child, and repeat.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.
Start at the leftmost leaves and go up as far as possible.

Go right as far as possible, to the last child, and repeat.

Fact. While going to the right, at most run is traversed.
Start at the leftmost leaves and go up as far as possible.

Go right as far as possible, to the last child, and repeat.

**Fact.** While going to the right, at most run is traversed.

Total running time: $O(\min(\text{DEPTH}(w_1), \text{DEPTH}(w_2)))$. 
Some nodes in the parse tree are preserved in any context.
Some nodes in the parse tree are preserved in any context.
Some nodes in the parse tree are preserved in any context.
Some nodes in the parse tree are \textit{preserved} in any \textit{context}.
Some nodes in the parse tree are preserved in any context.

Context-insensitive decomposition $D = aEFba$
Some nodes in the parse tree are preserved in any context.

Context-insensitive decomposition $D = aEFba$

In general, $|\text{RLE}(D)| \leq 2\text{DEPTH}(w)$. $\mathcal{O}(\text{DEPTH}(w))$ branching context-sensitive nodes.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $W$ in $\mathcal{O}(|\text{RLE}(D)| + \text{Depth}(w))$ time.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{Depth}(w))$ time.

- **make_string**: Build over the decomposition into letters.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $W$ in $O(|\text{RLE}(D)| + \text{DEPTH}(w))$ time.

- **make_string**: Build over the decomposition into letters.
- **concat, split**: Extract the context-insensitive decompositions and build over them.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{DEPTH}(w))$ time.

- **make_string**: Build over the decomposition into letters.
- **concat, split**: Extract the context-insensitive decompositions and build over them.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{Depth}(w))$ time.

- **make_string**: Build over the decomposition into letters.
- **concat, split**: Extract the context-insensitive decompositions and build over them.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{DEPTH}(w))$ time.

- **make_string**: Build over the decomposition into letters.
- **concat, split**: Extract the context-insensitive decompositions and build over them.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{DEPTH}(w))$ time.

- `make_string`: Build over the decomposition into letters.
- `concat, split`: Extract the context-insensitive decompositions and build over them.
Lemma (Building over a decomposition)

Given a run-length encoded decomposition $D$ of $w$, we can add $w$ to $\mathcal{W}$ in $O(|\text{RLE}(D)| + \text{DEPTH}(w))$ time.

- **make_string**: Build over the decomposition into letters.
- **concat, split**: Extract the context-insensitive decompositions and build over them.
Indexing dynamic strings:
- find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;
Indexing dynamic strings:

- find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;
- improves upon Alstrup et al;
- described in the full ArXiv version.
Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;
   - improves upon Alstrup et al;
   - described in the full ArXiv version.

2. Introduce $\text{drop\_string}(w)$ to remove $w \in \mathcal{W}$:
   - garbage collection of unused symbols;
   - $O(z \log n)$ symbols remain w.h.p., where $z$ is the minimum number of operations to generate $\mathcal{W}$.
Indexing dynamic strings:
- find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;
- improves upon Alstrup et al;
- described in the full ArXiv version.

Introduce $\text{drop\_string}(w)$ to remove $w \in \mathcal{W}$:
- garbage collection of unused symbols;
- $O(z \log n)$ symbols remain w.h.p., where $z$ is the minimum number of operations to generate $\mathcal{W}$;
- tweaking hash tables needed to guarantee w.h.p. running time in $O(z \log n)$ space.
Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;
   - improves upon Alstrup et al;
   - described in the full ArXiv version.

2. Introduce $\text{drop\_string}(w)$ to remove $w \in \mathcal{W}$:
   - garbage collection of unused symbols;
   - $O(z \log n)$ symbols remain w.h.p., where $z$ is the minimum number of operations to generate $\mathcal{W}$;
   - tweaking hash tables needed to guarantee w.h.p. running time in $O(z \log n)$ space.

3. Further $O(\log n)$-time operations:
Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings \( w \in \mathcal{W} \);
   - improves upon Alstrup et al;
   - described in the full ArXiv version.

2. Introduce \texttt{drop\_string}(w) to remove \( w \in \mathcal{W} \):
   - garbage collection of unused symbols;
   - \( \mathcal{O}(z \log n) \) symbols remain w.h.p., where \( z \) is the minimum number of operations to generate \( \mathcal{W} \);
   - tweaking hash tables needed to guarantee w.h.p. running time in \( \mathcal{O}(z \log n) \) space.

3. Further \( \mathcal{O}(\log n) \)-time operations:
   - \texttt{power}(w, k): insert the power \( w^k \) to \( \mathcal{W} \);
Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings \( w \in \mathcal{W} \);
   - improves upon Alstrup et al;
   - described in the full ArXiv version.

2. Introduce `drop_string(w)` to remove \( w \in \mathcal{W} \):
   - garbage collection of unused symbols;
   - \( \mathcal{O}(z \log n) \) symbols remain w.h.p., where \( z \) is the minimum number of operations to generate \( \mathcal{W} \);
   - tweaking hash tables needed to guarantee w.h.p. running time in \( \mathcal{O}(z \log n) \) space.

3. Further \( \mathcal{O}(\log n) \)-time operations:
   - `power(w, k)`: insert the power \( w^k \) to \( \mathcal{W} \);
   - `reverse(w)`: insert the reverse \( w^R \) to \( \mathcal{W} \).
Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings \( w \in \mathcal{W} \);
   - improves upon Alstrup et al;
   - described in the full ArXiv version.

2. Introduce \texttt{drop\_string}(w) to remove \( w \in \mathcal{W} \):
   - garbage collection of unused symbols;
   - \( \mathcal{O}(z \log n) \) symbols remain w.h.p., where \( z \) is the minimum number of operations to generate \( \mathcal{W} \);
   - tweaking hash tables needed to guarantee w.h.p. running time in \( \mathcal{O}(z \log n) \) space.

3. Further \( \mathcal{O}(\log n) \)-time operations:
   - \texttt{power}(w, k): insert the power \( w^k \) to \( \mathcal{W} \);
   - \texttt{reverse}(w): insert the reverse \( w^R \) to \( \mathcal{W} \);
   - \texttt{cyclic}(w): compute the \textit{primitive root} and canonize it with respect to cyclic equivalence.
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Ours</th>
<th>Impossible</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_string</td>
<td>$O(</td>
<td>w</td>
</tr>
<tr>
<td>concat</td>
<td>$O(\log n)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>split</td>
<td>$O(\log n)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>equal</td>
<td>$O(1)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>compare</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>LCP</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>persistent</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Las Vegas</td>
<td>amortized</td>
<td>Monte Carlo</td>
</tr>
<tr>
<td>worst-case</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
### Summary

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Ours</th>
<th>Impossible</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_string</td>
<td>$O(</td>
<td>w</td>
</tr>
<tr>
<td>concat</td>
<td>$O(\log n)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>split</td>
<td>$O(\log n)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>equal</td>
<td>$O(1)$</td>
<td>$o(\log n)$</td>
</tr>
<tr>
<td>compare</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>LCP</td>
<td>$O(1)$</td>
<td>-</td>
</tr>
<tr>
<td>persistent</td>
<td>YES</td>
<td>NO</td>
</tr>
</tbody>
</table>

- Las Vegas
- Monte Carlo
- worst-case
- amortized

---

Thank you for your attention!