Optimal Dynamic Strings

Paweł Gawrychowski[2], **Tomasz Kociumaka**[1], Adam Karczmarz[1], Jakub Łącki[3], Piotr Sankowski[1]

[1]University of Warsaw, Poland
[2]University of Wrocław, Poland
[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:

Dynamic Strings Problem

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

- $\texttt{make\_string}(w)$: insert $w \in \Sigma^+$ to $\mathcal{W}$;

$\mathcal{W}$: $\texttt{ab}$

$\texttt{1}$

$$\texttt{make\_string}(\texttt{ab}) = 1$$

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 $w_1 w_2$ to $\mathcal{W}$ for $w_1, w_2 \in \mathcal{W}$;

$\mathcal{W}$:	ab	abab
	1	2

$$\texttt{concat}(1,1) = 2$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;

$\mathcal{W}$:	ab	abab	a	bab
	1	2	3	4

$$\texttt{split}(2,1) = (3,4)$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: check whether w_1 is the same string as w_2;

$\mathcal{W}$:	ab	abab	a	bab
	1	2	3	4

$$\texttt{equal}(1, 3) = \textbf{false}$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: check whether w_1 is the same string as w_2;

$\mathcal{W}$:	ab	abab	a	bab	ba	b
	1	2	3	4	5	6

$$\texttt{split}(4, 2) = (5, 6)$$

Dynamic Strings Problem

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

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

$$\mathtt{concat}(6, 3) = 7$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: 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

$\texttt{equal}(5, 7) = \textbf{true}$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: check whether w_1 is the same string as w_2;

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

$$\texttt{concat}(7, 1) = 8$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: check whether w_1 is the same string as w_2;
- $\texttt{LCP}(w_1, w_2)$: return the length of the longest common prefix of w_1 and w_2;

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

$$\texttt{LCP}(4, 8) = 2$$

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 $w_1 w_2$ 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 $w \in \mathcal{W}$;
- $\texttt{equal}(w_1, w_2)$: check whether w_1 is the same string as w_2;
- $\texttt{LCP}(w_1, w_2)$: return the length of the longest common prefix of w_1 and w_2;
- $\texttt{compare}(w_1, w_2)$: lexicographically compare w_1 and w_2.

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

$$\texttt{compare}(4, 8) = \text{`>'}$$

Previous & Our Results

Setting

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

Previous & Our Results

Setting

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

	randomization	split, concat	equal	compare, LCP
„folklore"	Monte Carlo	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$

Previous & Our Results

Setting

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

	randomization	`split`, `concat`	`equal`	`compare`, `LCP`
„folklore"	Monte Carlo	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$
Mehlhorn et al. 1994	Las Vegas	$\mathcal{O}(\log^2 n)$ exp.	$\mathcal{O}(1)$	-
Mehlhorn et al. 1994	deterministic	$\mathcal{O}(\log^2 n \log^* n)$	$\mathcal{O}(1)$	-

Previous & Our Results

Setting

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

	randomization	split, concat	equal	compare, LCP
„folklore”	Monte Carlo	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$
Mehlhorn et al. 1994	Las Vegas	$\mathcal{O}(\log^2 n)$ exp.	$\mathcal{O}(1)$	-
Mehlhorn et al. 1994	deterministic	$\mathcal{O}(\log^2 n \log^* n)$	$\mathcal{O}(1)$	-
Alstrup et al. 2000	Las Vegas†	$\mathcal{O}(\log n \log^* n)$ w.h.p.	$\mathcal{O}(1)$	$\mathcal{O}(1)$

†due to dynamic dictionaries (hash tables) only.

Previous & Our Results

Setting

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

	randomization	`split`, `concat`	`equal`	`compare`, `LCP`
„folklore"	Monte Carlo	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$
Mehlhorn et al. 1994	Las Vegas	$\mathcal{O}(\log^2 n)$ exp.	$\mathcal{O}(1)$	-
Mehlhorn et al. 1994	deterministic	$\mathcal{O}(\log^2 n \log^* n)$	$\mathcal{O}(1)$	-
Alstrup et al. 2000	Las Vegas†	$\mathcal{O}(\log n \log^* n)$ w.h.p.	$\mathcal{O}(1)$	$\mathcal{O}(1)$
this work	Las Vegas	$\mathcal{O}(\log n)$ w.h.p.	$\mathcal{O}(1)$	$\mathcal{O}(1)$

†due to dynamic dictionaries (hash tables) only.

Previous & Our Results

Setting

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

	randomization	`split`, `concat`	`equal`	`compare`, `LCP`
„folklore"	Monte Carlo	$\mathcal{O}(\log n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log^2 n)$
Mehlhorn et al. 1994	Las Vegas	$\mathcal{O}(\log^2 n)$ exp.	$\mathcal{O}(1)$	-
Mehlhorn et al. 1994	deterministic	$\mathcal{O}(\log^2 n \log^* n)$	$\mathcal{O}(1)$	-
Alstrup et al. 2000	Las Vegas†	$\mathcal{O}(\log n \log^* n)$ w.h.p.	$\mathcal{O}(1)$	$\mathcal{O}(1)$
this work	Las Vegas	$\mathcal{O}(\log n)$ w.h.p.	$\mathcal{O}(1)$	$\mathcal{O}(1)$

†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.

„Folklore” Solution

Karp–Rabin Fingerprints

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

„Folklore” Solution

Karp–Rabin Fingerprints

- $H(w) = H(w') \quad \Longrightarrow \quad w = w'$ w.h.p.
- $H(u), H(v) \quad \rightsquigarrow \quad H(uv)$ in $\mathcal{O}(1)$ time.

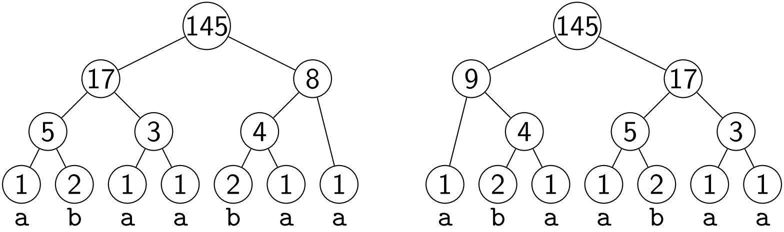
Idea: store each string in a persistent BST supporting split & join.

„Folklore” Solution

Karp–Rabin Fingerprints

- $H(w) = H(w') \quad \Longrightarrow \quad w = w'$ w.h.p.
- $H(u), H(v) \quad \rightsquigarrow \quad H(uv)$ in $\mathcal{O}(1)$ time.

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

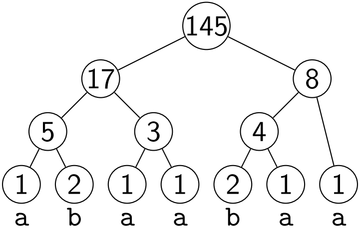


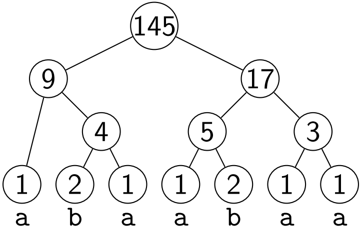
„Folklore" Solution

Karp–Rabin Fingerprints

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

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





`split`(w, k) $\mathcal{O}(\log |w|)$

`concat`(w_1, w_2) $\mathcal{O}(\log |w_1 w_2|)$

`make_string`(w) $\mathcal{O}(|w|)$

`equal`(w_1, w_2) $\mathcal{O}(1)$

„Folklore" Solution: Issues

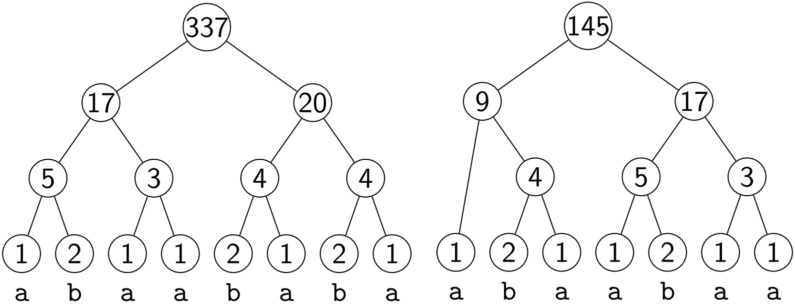
1. Monte Carlo: `equal` is correct w.h.p. only

„Folklore" Solution: Issues

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

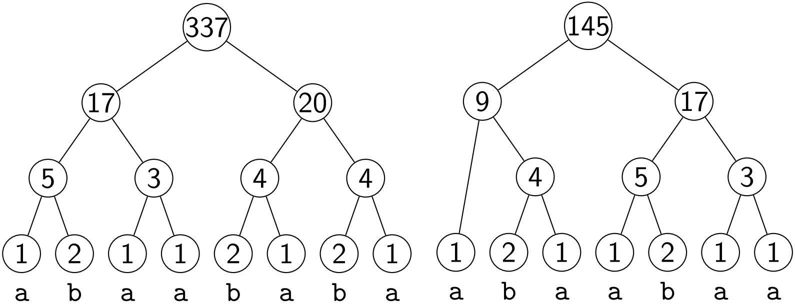
„Folklore” Solution: Issues

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



„Folklore" Solution: Issues

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

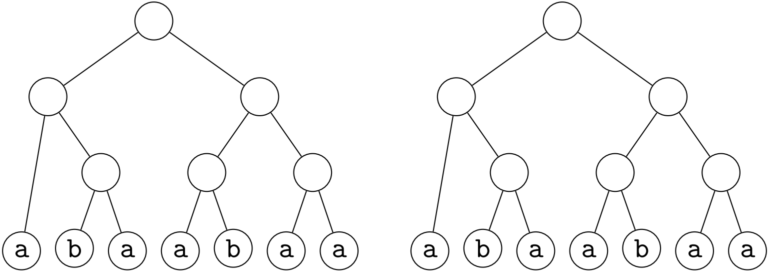
Consistent Parsing

Consistent Parsing (Mehlhorn et al. SODA'94; Sahinalp–Vishkin STOC'94)

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

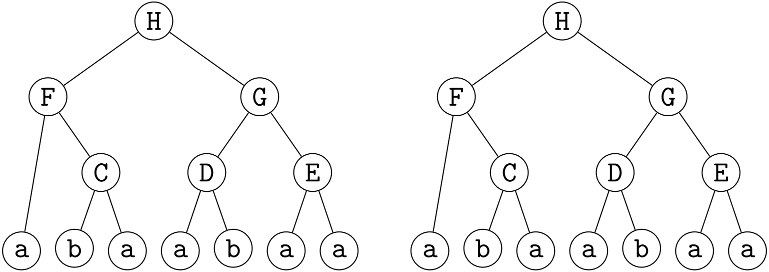
Consistent Parsing (Mehlhorn et al. SODA'94; Sahinalp–Vishkin STOC'94)

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



Consistent Parsing (Mehlhorn et al. SODA'94; Sahinalp–Vishkin STOC'94)

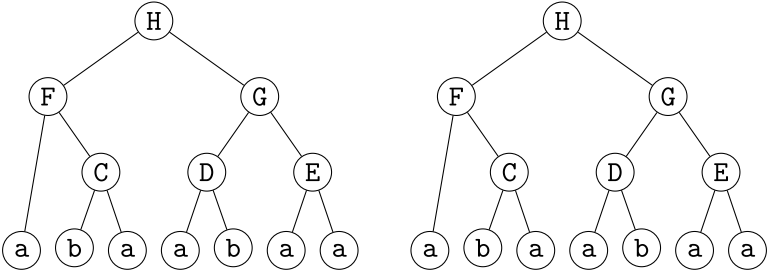
Shape of the parse tree is determined by the underlying string.



- Name each node based on the names of its children.

Consistent Parsing (Mehlhorn et al. SODA'94; Sahinalp–Vishkin STOC'94)

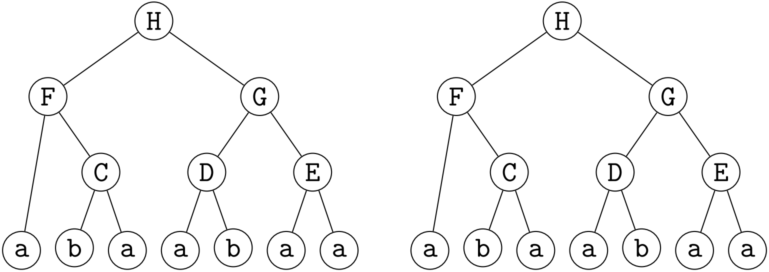
Shape of the parse tree is determined by the underlying string.



- 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.

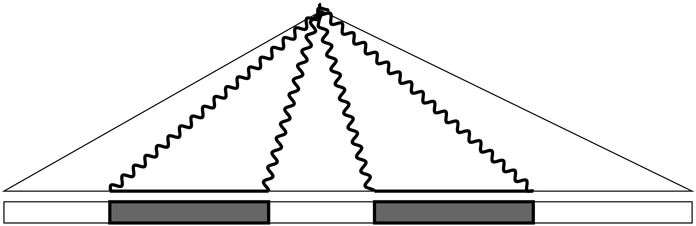
Locally Consistent Parsing

Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

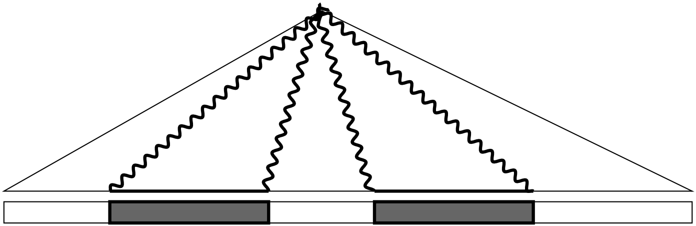
Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.



Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

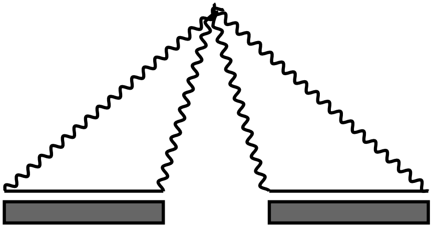
Equal fragments are parsed almost in the same way.



Why is local consistence useful?

Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

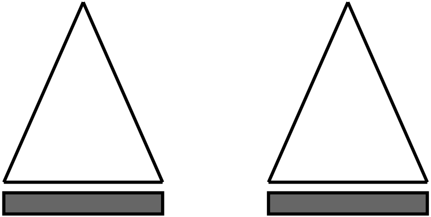


Why is local consistence useful?

- allows to maintain consistent parsing,

Locally Consistent Parsing (Mehlhorn et al.; Sahinalp–Vishkin)

Equal fragments are parsed almost in the same way.

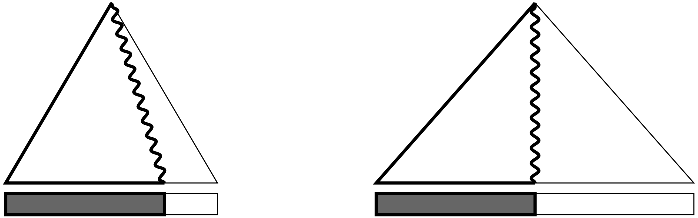


Why is local consistence useful?

- 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?

Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

a a a a a a a a

Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

a a a a a a a a

Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

a a a a a a a a

Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

a a a $\boxed{\text{a a a a}}$ a

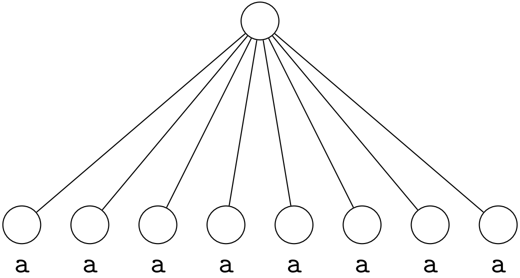
Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

a a a a a a a a

Parse Tree Construction: RLE

How to parse highly-repetitive fragments?

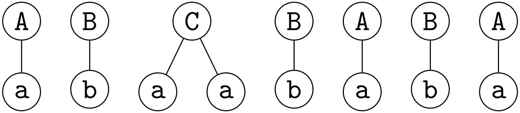


Run-Length Encoding (RLE)

Replace each run with a new symbol: $\underbrace{a \cdots a}_{k \text{ times}} \mapsto (a, k)$.

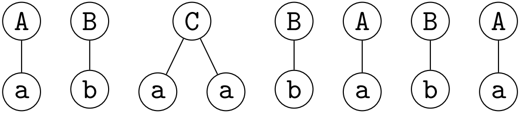
$$\mathtt{aaabbabaabbbaa} \mapsto (\mathtt{a}, 3)(\mathtt{b}, 2)(\mathtt{a}, 1)(\mathtt{b}, 1)(\mathtt{a}, 2)(\mathtt{b}, 3)(\mathtt{a}, 2)$$

Parse Tree Construction: Compress



RLE

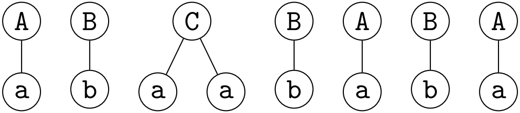
$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

How to consistently partition a string without non-trivial runs?

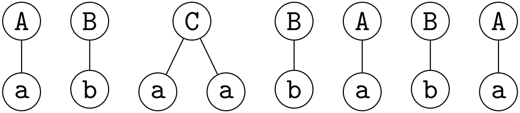


RLE
$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

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: $\mathcal{O}(1)$.

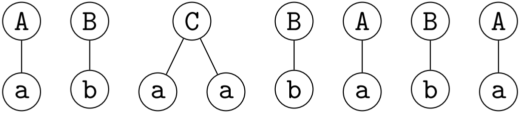


RLE
$\mathtt{A} = (\mathtt{a}, 1),\ \mathtt{B} = (\mathtt{b}, 1),\ \mathtt{C} = (\mathtt{a}, 2)$

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: $\mathcal{O}(1)$.



RLE
$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

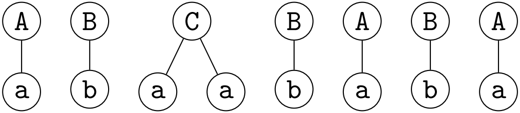
How to consistently partition a string without non-trivial runs?

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: $\mathcal{O}(\log^* n)$.



RLE

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

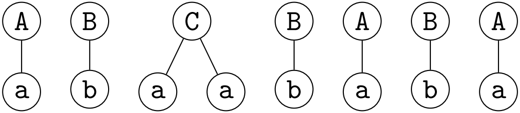
How to consistently partition a string without non-trivial runs?

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

- Block length: 2–$\mathcal{O}(\log n)$ 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: $\mathcal{O}(\log^* n)$.



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

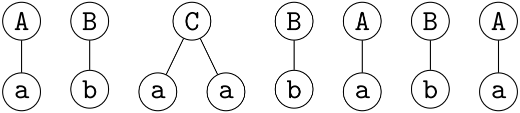
How to consistently partition a string without non-trivial runs?

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.



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

How to consistently partition a string without non-trivial runs?

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

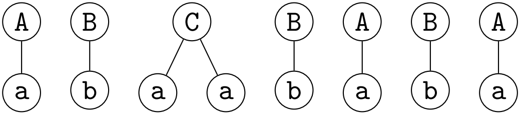
- Block length: 2–$\mathcal{O}(\log n)$ w.h.p., $\mathcal{O}(1)$ in expectation.
- Context size: $\mathcal{O}(1)$.

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.



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

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: $\mathcal{O}(1)$.

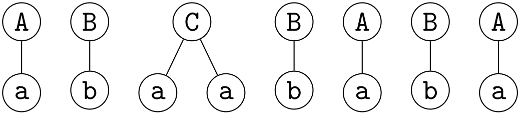
Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):

- 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.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.

Parse Tree Construction: Compress



RLE
$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

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: $\mathcal{O}(1)$.

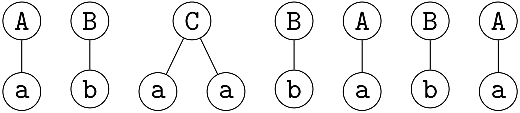
Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):

- 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.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
- Context size: $\mathcal{O}(1)$.

Parse Tree Construction: Compress



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

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: $\mathcal{O}(1)$.

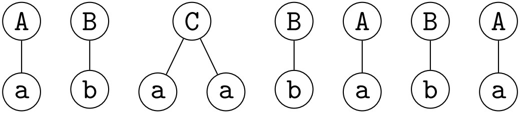
Mehlhorn et al. Deterministic coin tossing (by Cole & Vishkin):

- 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.
- A block of length 2 when a right symbol follows a left symbol.
- Remaining characters in blocks of length 1.
- Context size: $\mathcal{O}(1)$.

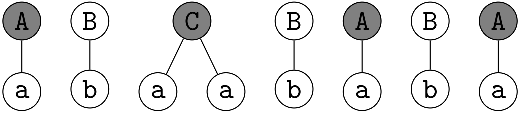
Parse Tree Construction: Example



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

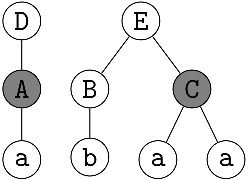
Parse Tree Construction: Example



RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

Parse Tree Construction: Example



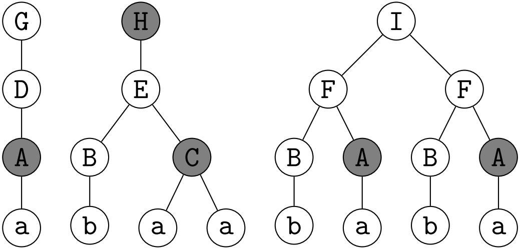
COMPRESS

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

Parse Tree Construction: Example



RLE

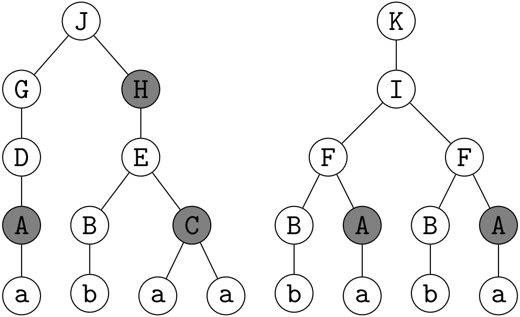
$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

Compress

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$



COMPRESS

$\mathtt{J} = (\mathtt{G}, \mathtt{H})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

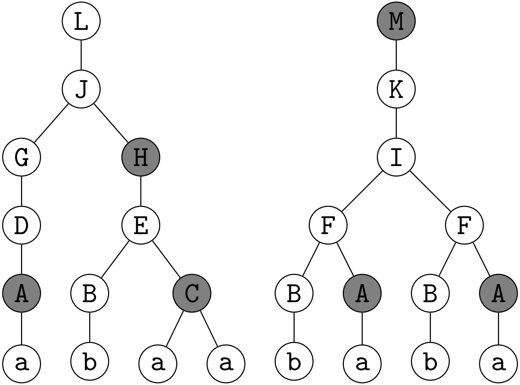
COMPRESS

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

Parse Tree Construction: Example



RLE

$\mathtt{L} = (\mathtt{J}, 1)$, $\mathtt{M} = (\mathtt{K}, 1)$

COMPRESS

$\mathtt{J} = (\mathtt{G}, \mathtt{H})$, $\mathtt{K} = (\mathtt{I})$

RLE

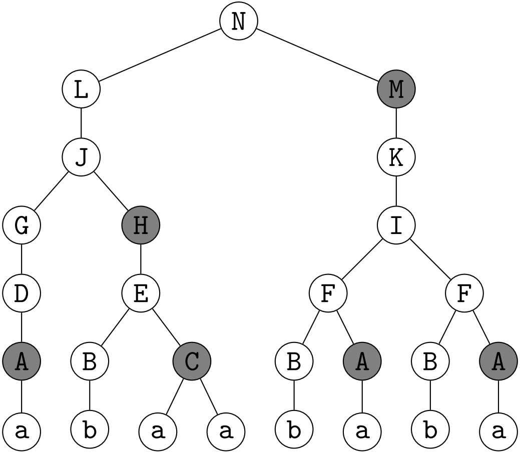
$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

COMPRESS

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$



COMPRESS

$\mathtt{N} = (\mathtt{L}, \mathtt{M})$

RLE

$\mathtt{L} = (\mathtt{J}, 1)$, $\mathtt{M} = (\mathtt{K}, 1)$

COMPRESS

$\mathtt{J} = (\mathtt{G}, \mathtt{H})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

COMPRESS

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

Parse Tree Construction: Alphabet Partitioning

How to partition the alphabet at each level?

Parse Tree Construction: Alphabet Partitioning

How to partition the alphabet at each level?

Jeż To guarantee compression ratio $r < 1$

Parse Tree Construction: Alphabet Partitioning

How to partition the alphabet at each level?

Jeż To guarantee compression ratio $r < 1$

- Works well in the static setting!

Parse Tree Construction: Alphabet Partitioning

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 $\frac{1}{4}$.

Parse Tree Construction: Alphabet Partitioning

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: $\frac{3}{4}$.

Parse Tree Construction: Alphabet Partitioning

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: $\frac{3}{4}$.

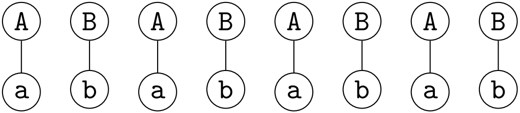
Lemma

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

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

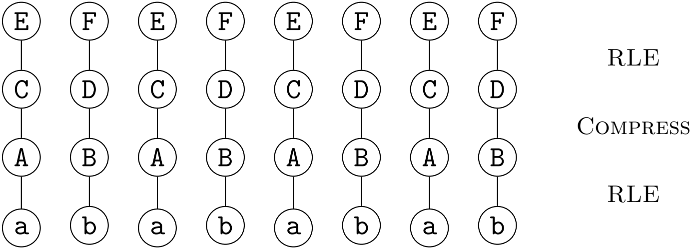
In short: depth $\mathcal{O}(\log n)$ with high probability.

Difficulties

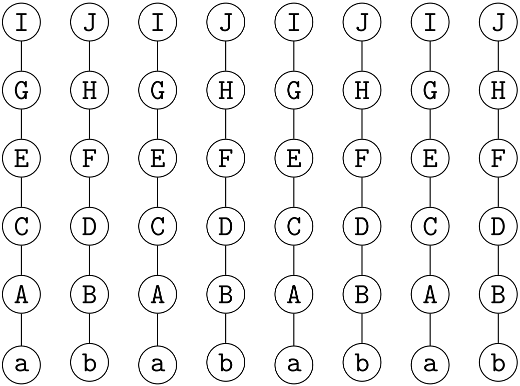


RLE

Difficulties



Difficulties



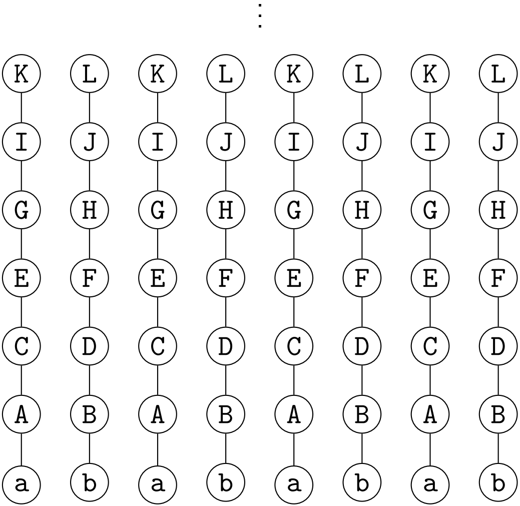
COMPRESS

RLE

COMPRESS

RLE

Difficulties



COMPRESS

RLE

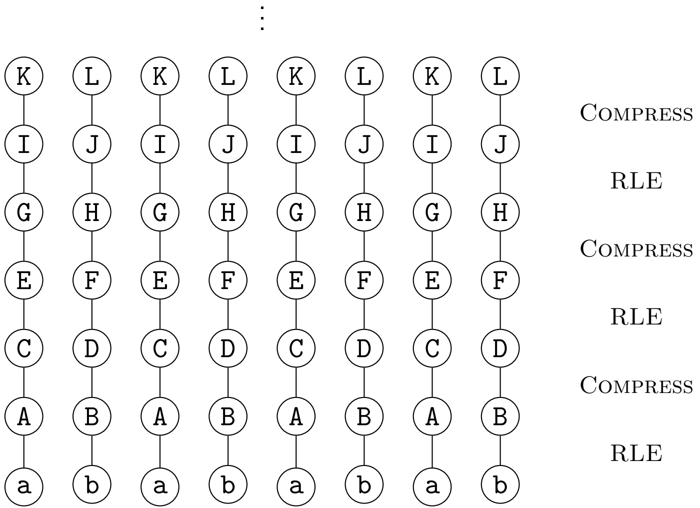
COMPRESS

RLE

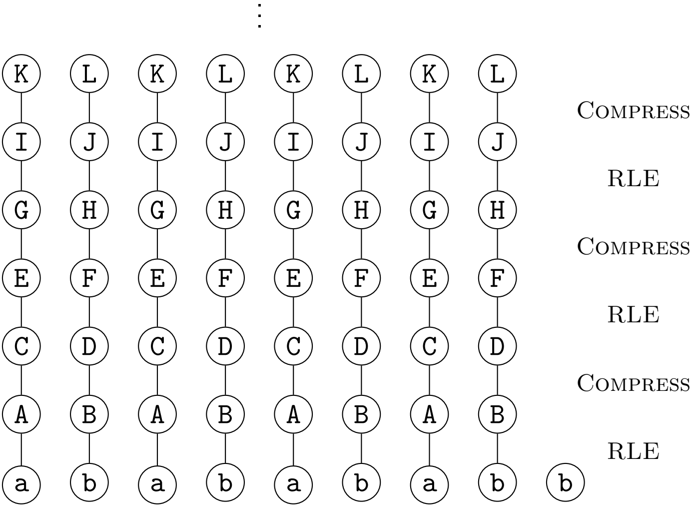
COMPRESS

RLE

Difficulties

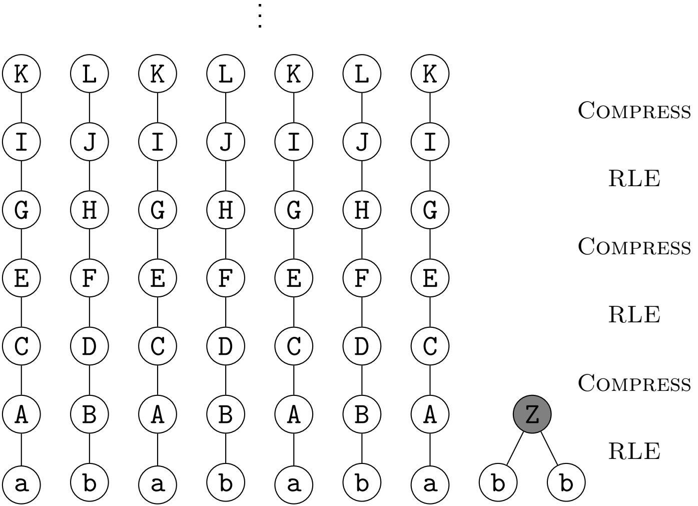


- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.



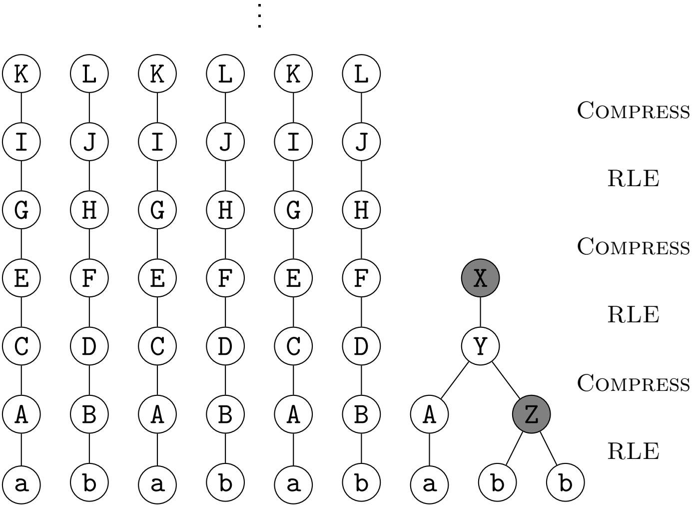
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.

Difficulties



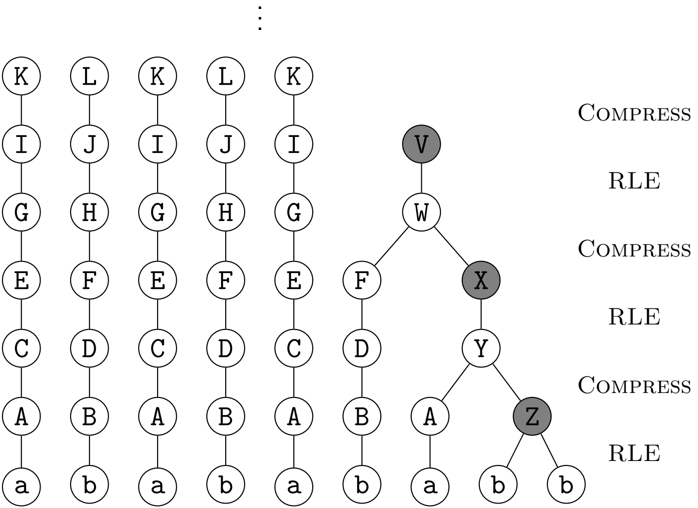
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.

Difficulties



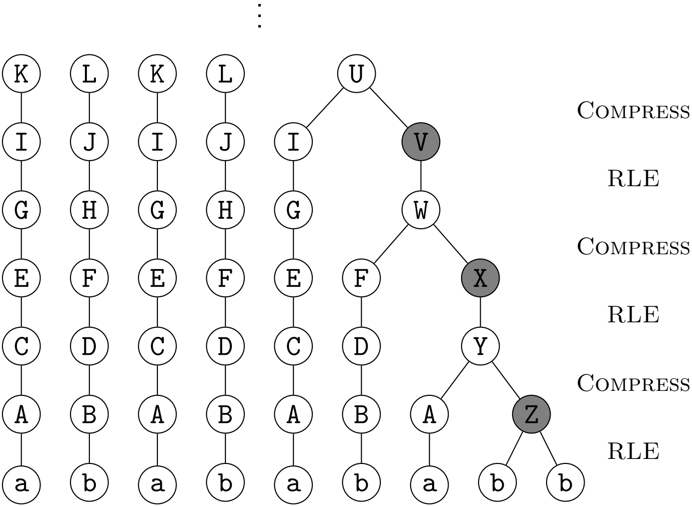
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.

Difficulties



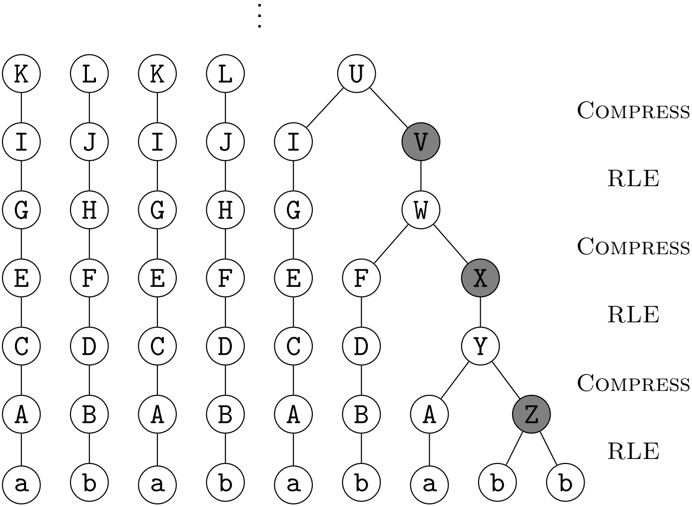
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.

Difficulties



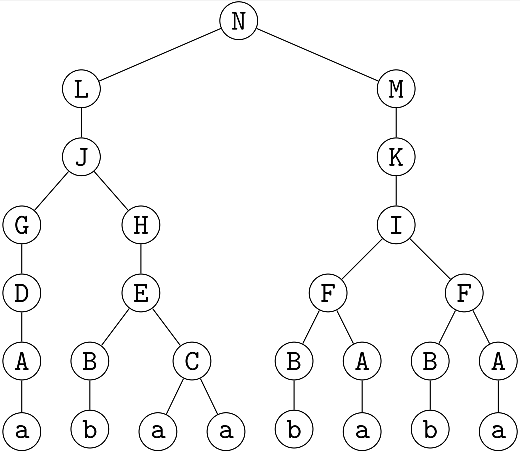
- $\Theta(n \log n)$ nodes with non-negligible probability $\frac{1}{n^{\varepsilon}}$.

Difficulties



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

Workaround: Compress the Parse Tree



COMPRESS

$\mathtt{N} = (\mathtt{L}, \mathtt{M})$

RLE

$\mathtt{L} = (\mathtt{J}, 1)$, $\mathtt{M} = (\mathtt{K}, 1)$

COMPRESS

$\mathtt{J} = (\mathtt{G}, \mathtt{H})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

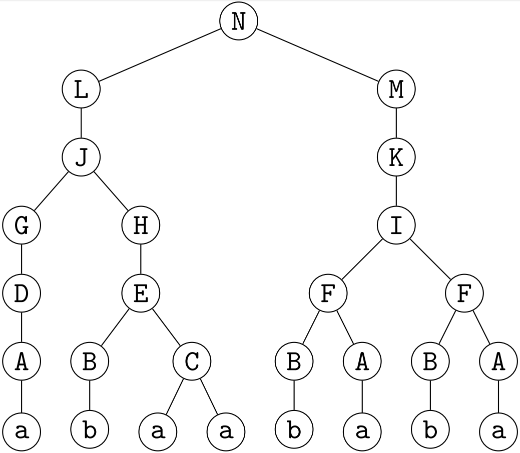
COMPRESS

$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

Workaround: Compress the Parse Tree



COMPRESS

$\mathtt{N} = (\mathtt{L}, \mathtt{M})$

RLE

$\mathtt{L} = (\mathtt{J}, 1)$, $\mathtt{M} = (\mathtt{K}, 1)$

COMPRESS

$\mathtt{J} = (\mathtt{G}, \mathtt{H})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{G} = (\mathtt{D}, 1)$, $\mathtt{H} = (\mathtt{E}, 1)$, $\mathtt{I} = (\mathtt{F}, 2)$

COMPRESS

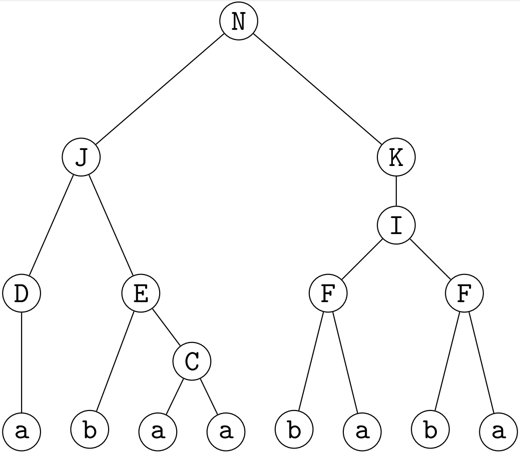
$\mathtt{D} = (\mathtt{A})$, $\mathtt{E} = (\mathtt{B}, \mathtt{C})$, $\mathtt{F} = (\mathtt{B}, \mathtt{A})$

RLE

$\mathtt{A} = (\mathtt{a}, 1)$, $\mathtt{B} = (\mathtt{b}, 1)$, $\mathtt{C} = (\mathtt{a}, 2)$

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

Workaround: Compress the Parse Tree



COMPRESS

$\mathtt{N} = (\mathtt{J}, \mathtt{K})$

RLE

COMPRESS

$\mathtt{J} = (\mathtt{D}, \mathtt{E})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{I} = (\mathtt{F}, 2)$

COMPRESS

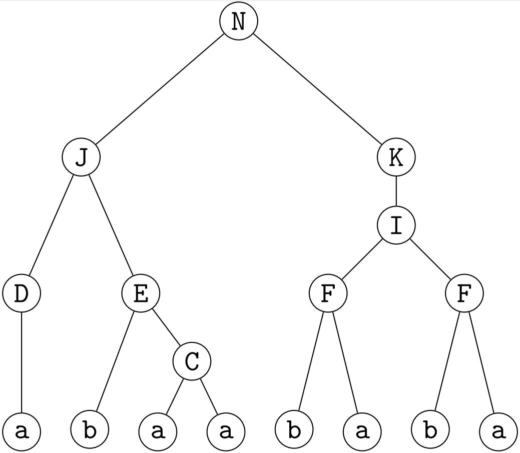
$\mathtt{D} = (\mathtt{a})$, $\mathtt{E} = (\mathtt{b}, \mathtt{C})$, $\mathtt{F} = (\mathtt{b}, \mathtt{a})$

RLE

$\mathtt{C} = (\mathtt{a}, 2)$

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

Workaround: Compress the Parse Tree



COMPRESS

$\mathtt{N} = (\mathtt{J}, \mathtt{K})$

RLE

COMPRESS

$\mathtt{J} = (\mathtt{D}, \mathtt{E})$, $\mathtt{K} = (\mathtt{I})$

RLE

$\mathtt{I} = (\mathtt{F}, 2)$

COMPRESS

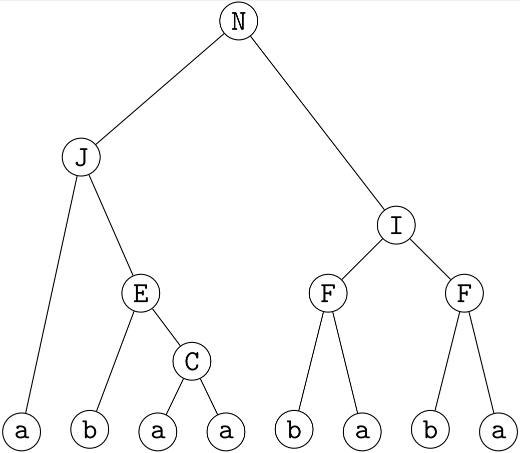
$\mathtt{D} = (\mathtt{a})$, $\mathtt{E} = (\mathtt{b}, \mathtt{C})$, $\mathtt{F} = (\mathtt{b}, \mathtt{a})$

RLE

$\mathtt{C} = (\mathtt{a}, 2)$

- Do not replace S with $(S, 1)$ in RLE.
- Do not introduce new symbols for unary blocks in COMPRESS.

Workaround: Compress the Parse Tree



COMPRESS

$\mathtt{N} = (\mathtt{J}, \mathtt{I})$

RLE

COMPRESS

$\mathtt{J} = (\mathtt{a}, \mathtt{E})$

RLE

$\mathtt{I} = (\mathtt{F}, 2)$

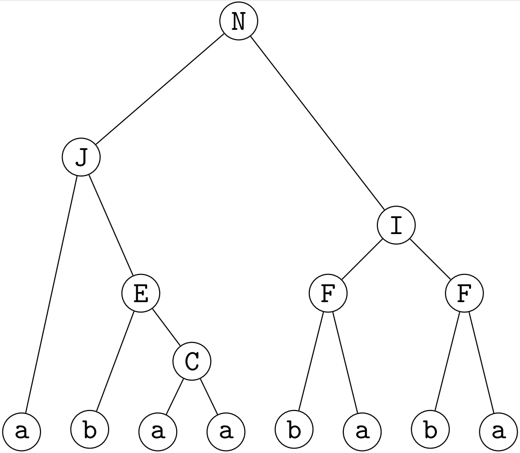
COMPRESS

$\mathtt{E} = (\mathtt{b}, \mathtt{C})$, $\mathtt{F} = (\mathtt{b}, \mathtt{a})$

RLE

$\mathtt{C} = (\mathtt{a}, 2)$

- Do not replace S with $(S, 1)$ in RLE.
- Do not introduce new symbols for unary blocks in COMPRESS.



6 COMPRESS

$N = (J, I)$

5 RLE

4 COMPRESS

$J = (a, E)$

3 RLE

$I = (F, 2)$

2 COMPRESS

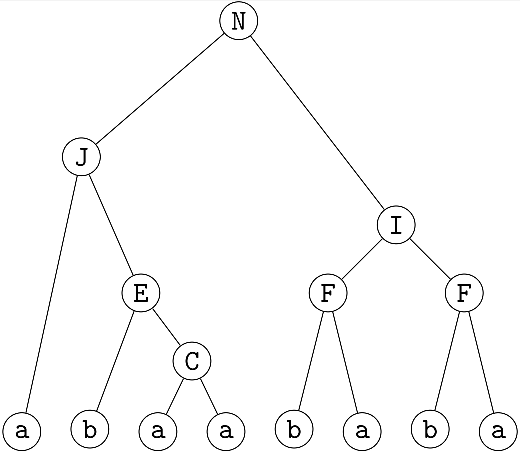
$E = (b, C)$, $F = (b, a)$

1 RLE

$C = (a, 2)$

0

- 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 ...



6 COMPRESS

$\mathtt{N} = (\mathtt{J}, \mathtt{I})$

5 RLE

4 COMPRESS

$\mathtt{J} = (\mathtt{a}, \mathtt{E})$

3 RLE

$\mathtt{I} = (\mathtt{F}, 2)$

2 COMPRESS

$\mathtt{E} = (\mathtt{b}, \mathtt{C})$, $\mathtt{F} = (\mathtt{b}, \mathtt{a})$

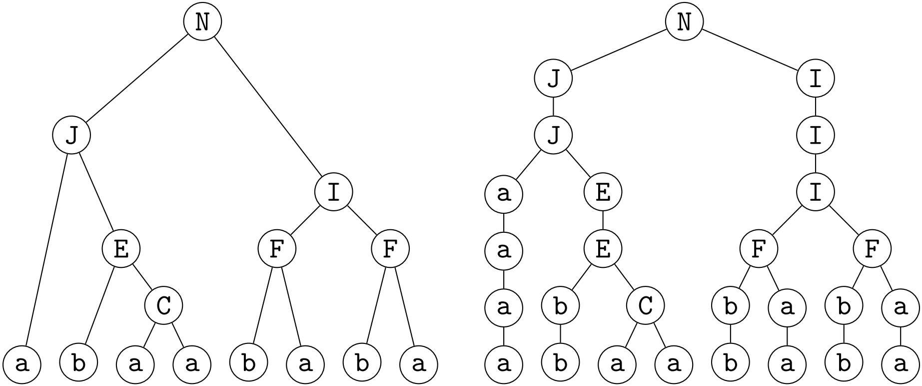
1 RLE

$\mathtt{C} = (\mathtt{a}, 2)$

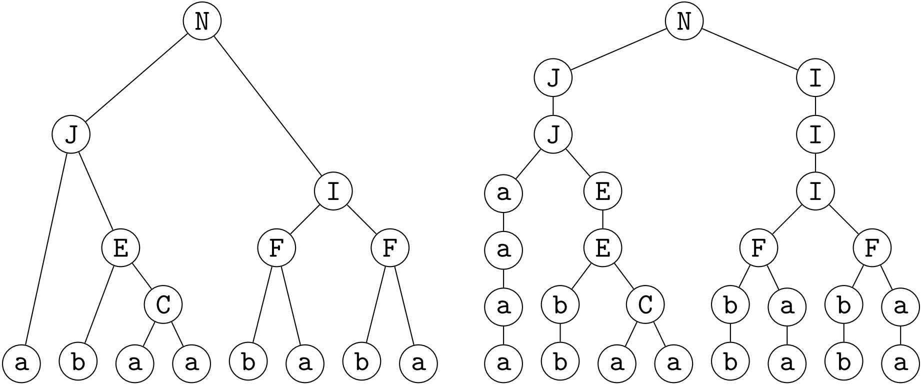
0

- 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 ...
- and a random bit for each larger even level ($\mathcal{O}(\log n)$ w.h.p.).

Navigating Uncompressed Parse Trees



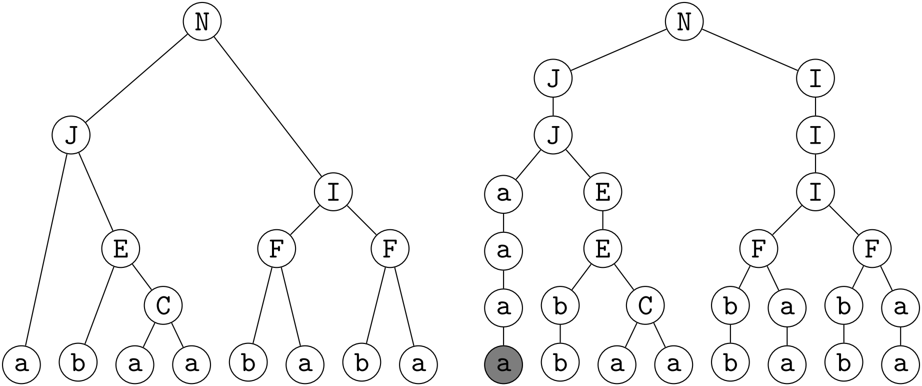
Navigating Uncompressed Parse Trees



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

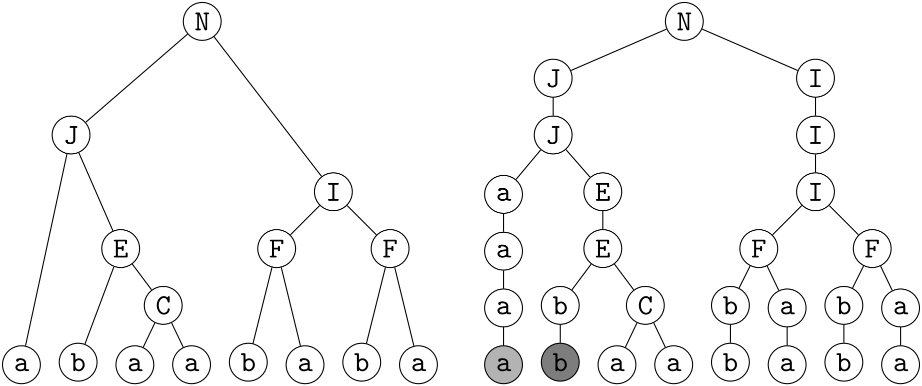
Navigating Uncompressed Parse Trees



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

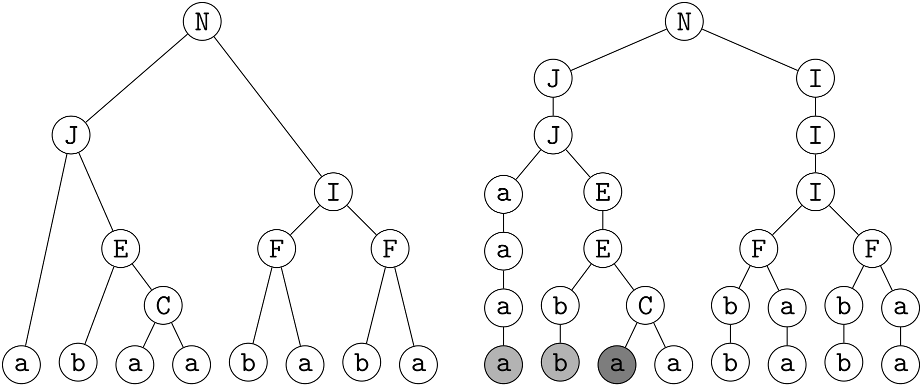
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs

Navigating Uncompressed Parse Trees



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

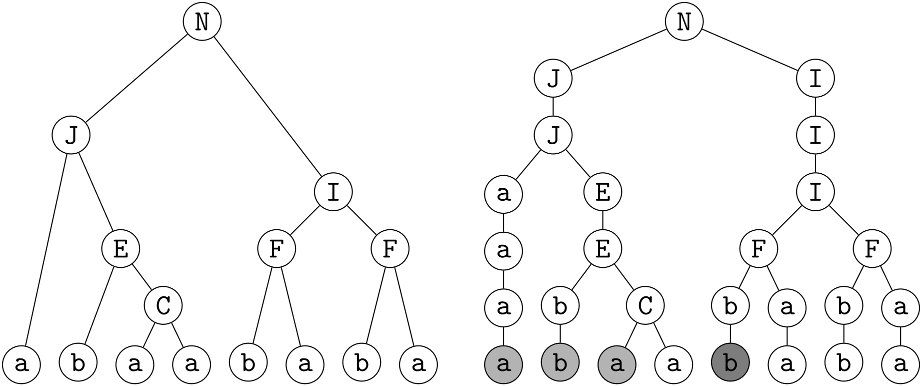
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

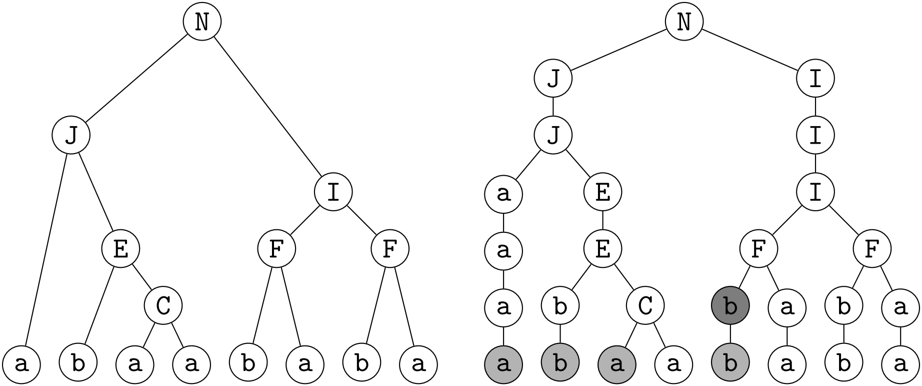
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs

Navigating Uncompressed Parse Trees



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

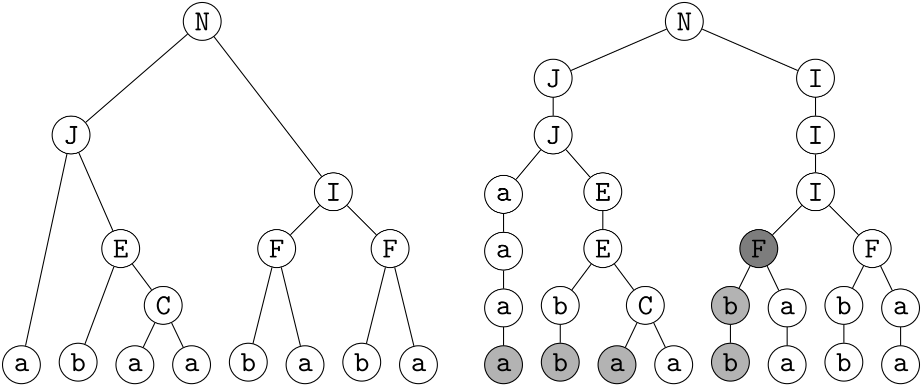
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

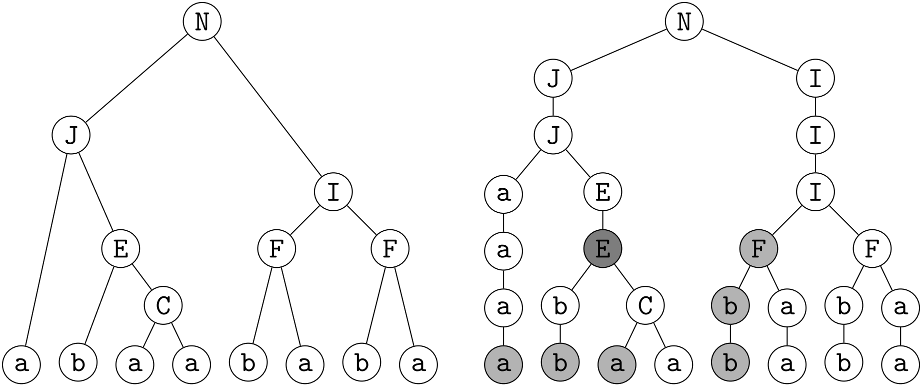
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs

Navigating Uncompressed Parse Trees



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

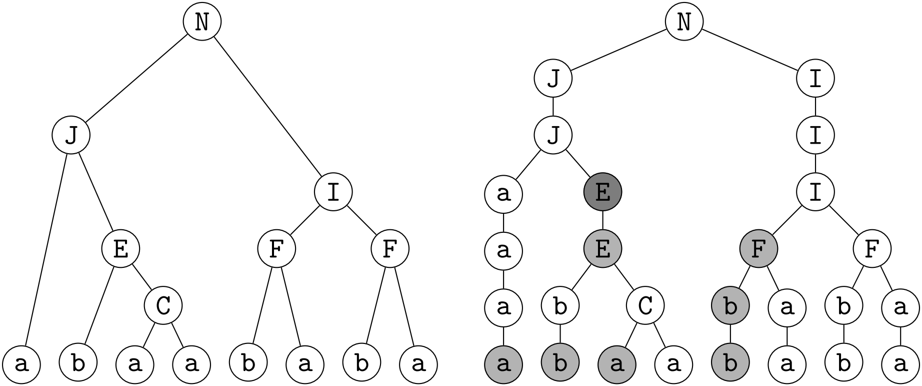
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

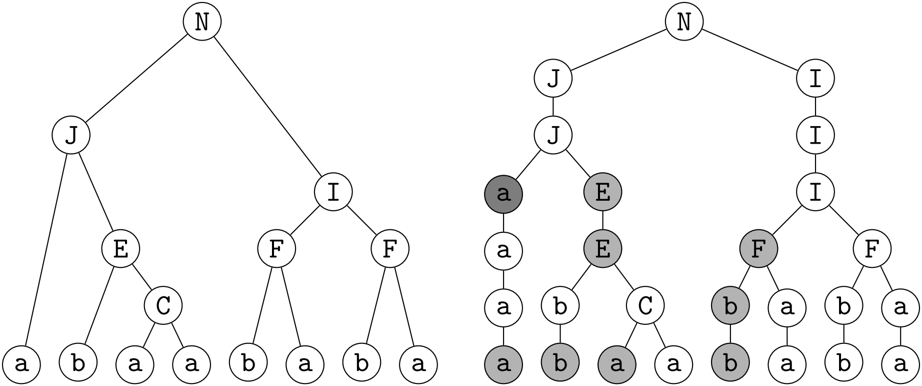
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs

Navigating Uncompressed Parse Trees



Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

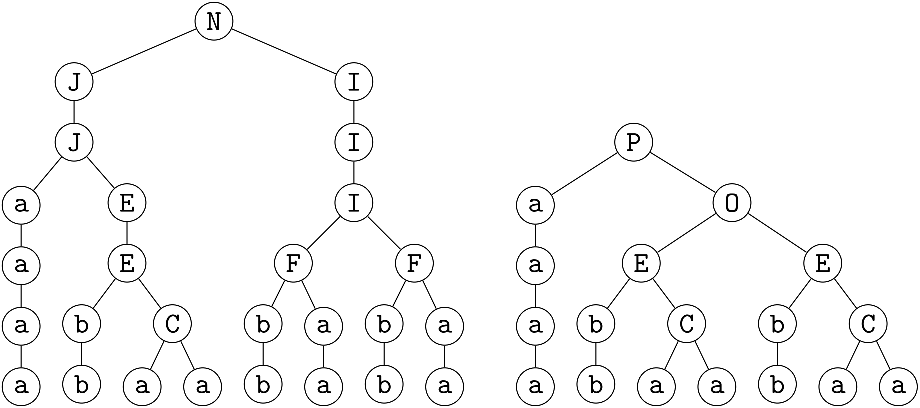
- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs



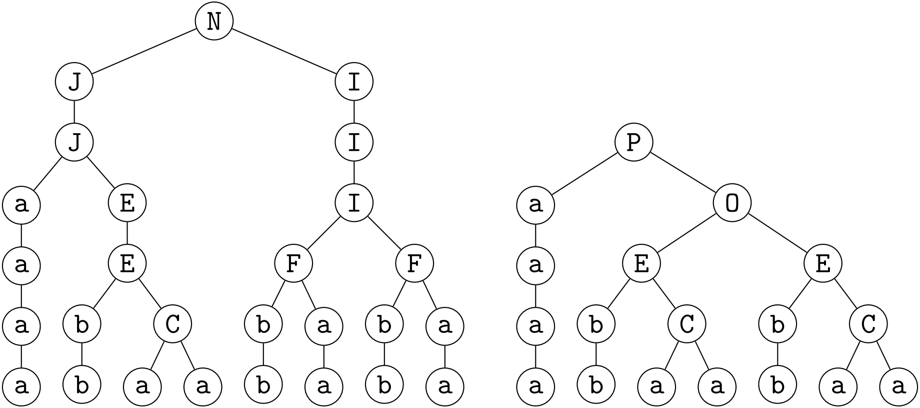
Navigate the uncompressed parse trees in $\mathcal{O}(1)$ time:

- traverse edges: go to the parent or to the k-th child
- traverse levels: go left or right, perhaps skipping runs

Simple LCP and compare Implementation

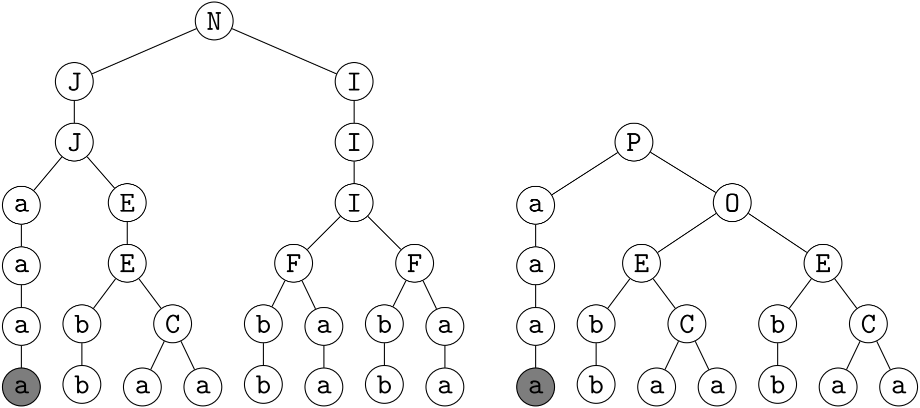


Simple LCP and compare Implementation



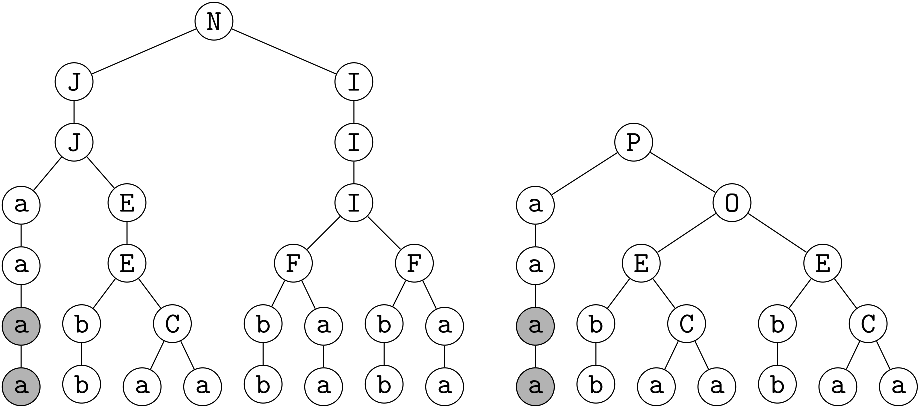
1. Start at the leftmost leaves and go up as far as possible.

Simple LCP and compare Implementation



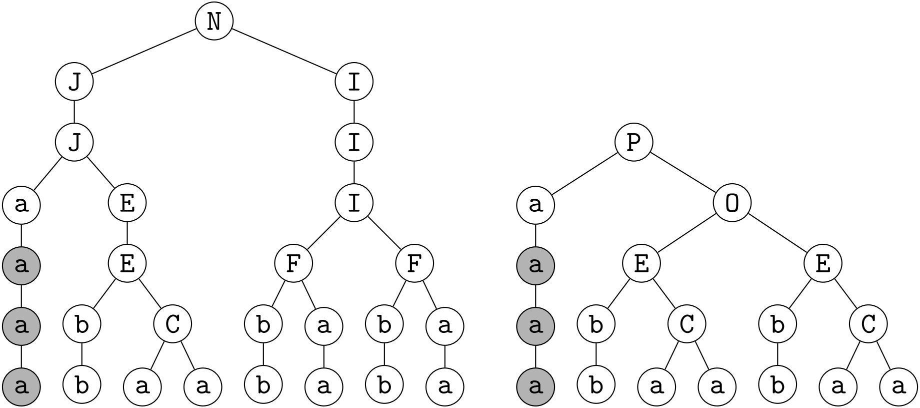
1. Start at the leftmost leaves and go up as far as possible.

Simple LCP and compare Implementation

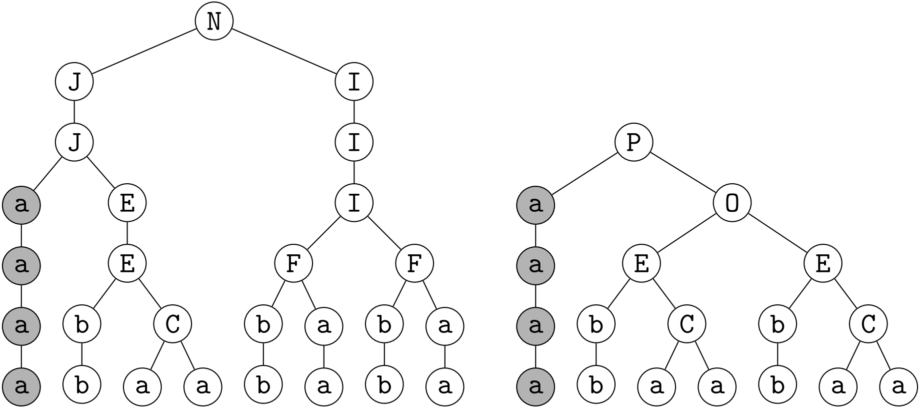


1. Start at the leftmost leaves and go up as far as possible.

Simple LCP and compare Implementation

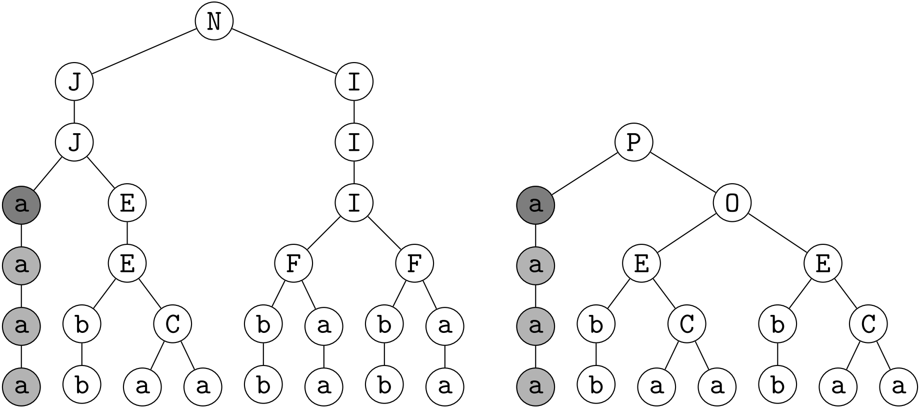


1. Start at the leftmost leaves and go up as far as possible.



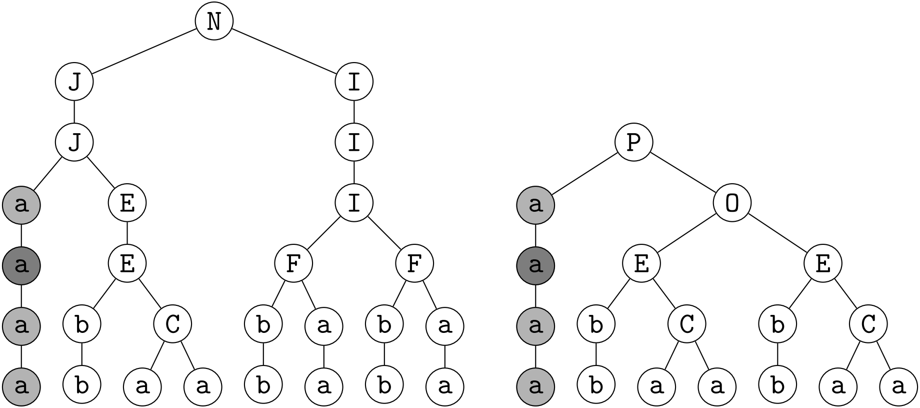
1. Start at the leftmost leaves and go up as far as possible.

Simple LCP and compare Implementation



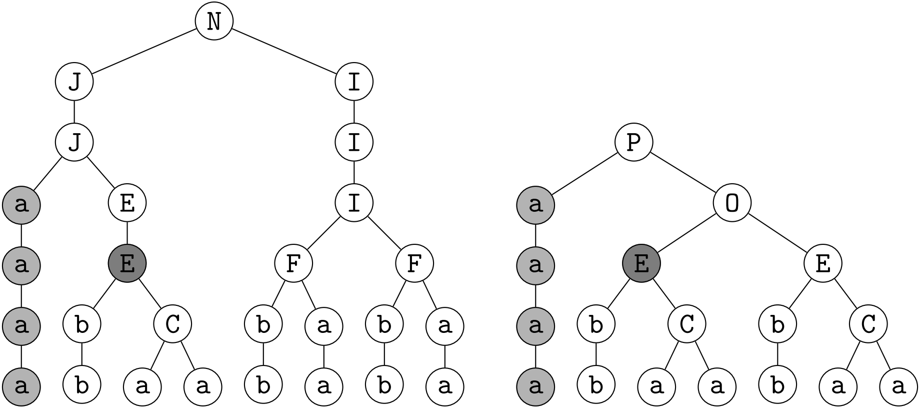
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

Simple LCP and compare Implementation



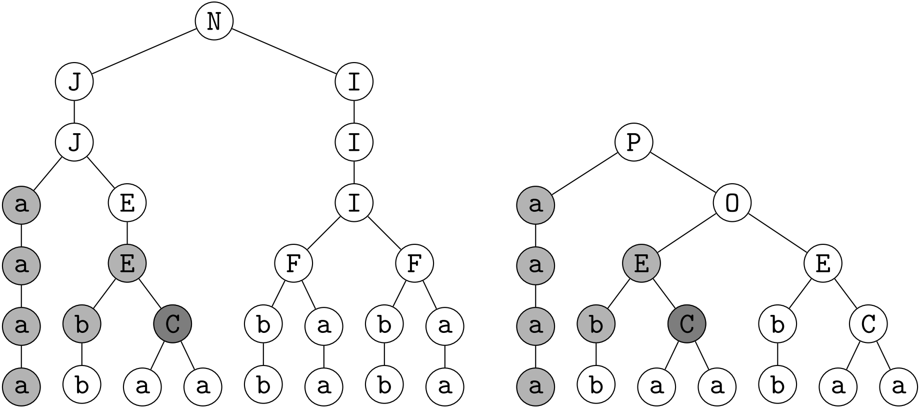
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

Simple LCP and compare Implementation

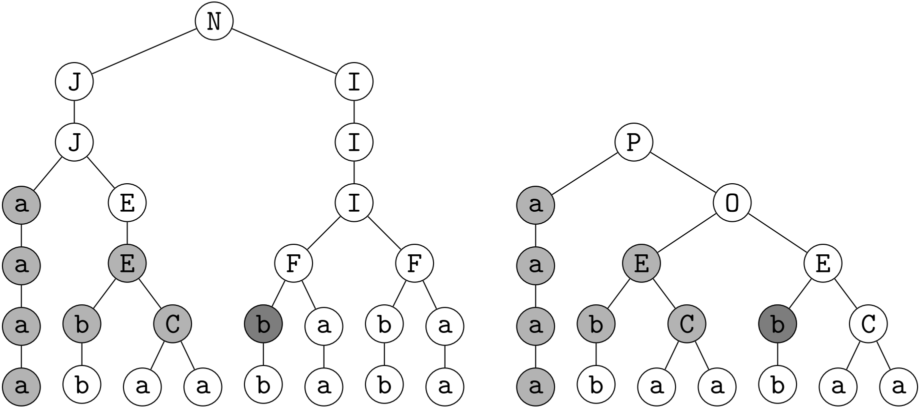


1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

Simple LCP and compare Implementation

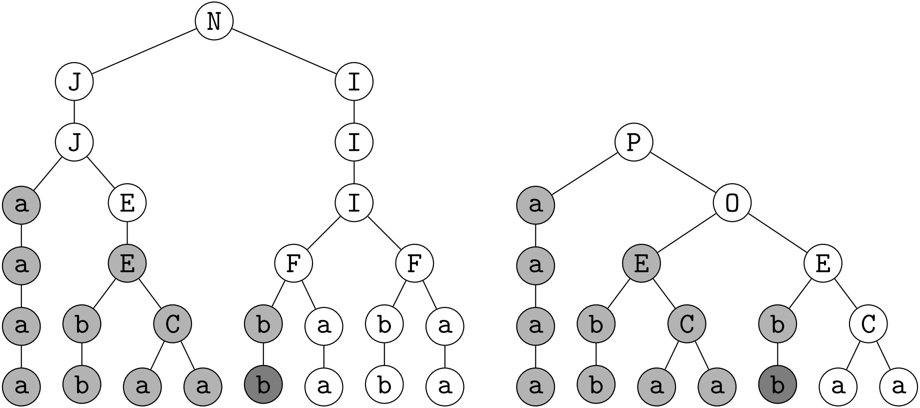


1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.



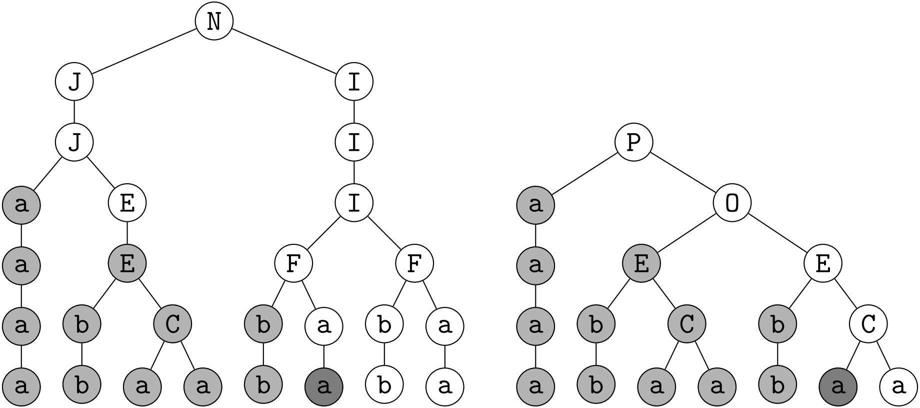
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

Simple LCP and compare Implementation



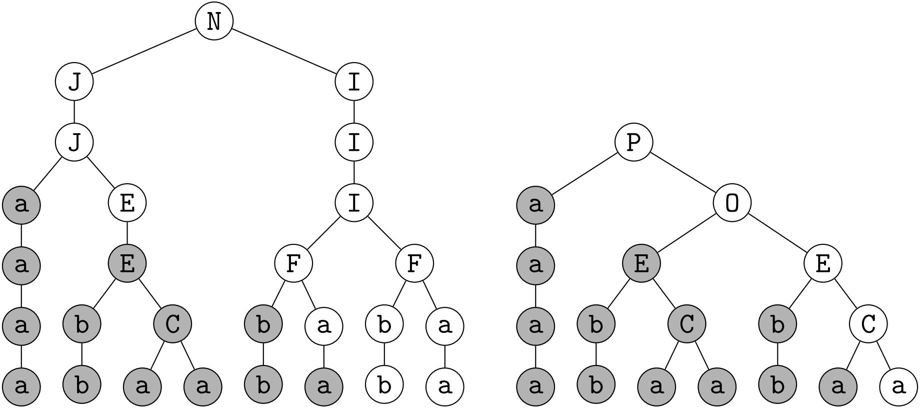
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

Simple LCP and compare Implementation



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.

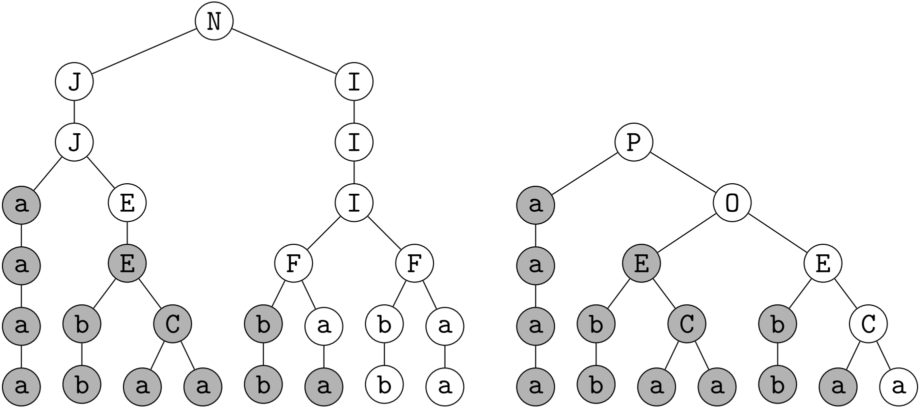
Simple LCP and compare Implementation



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.

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

Simple LCP and compare Implementation



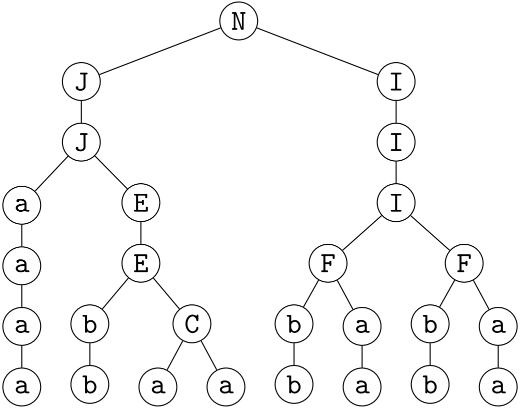
1. Start at the leftmost leaves and go up as far as possible.
2. Go right as far as possible, to the last child, and repeat.

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

Total running time: $\mathcal{O}(\min(\textsc{Depth}(w_1), \textsc{Depth}(w_2)))$.

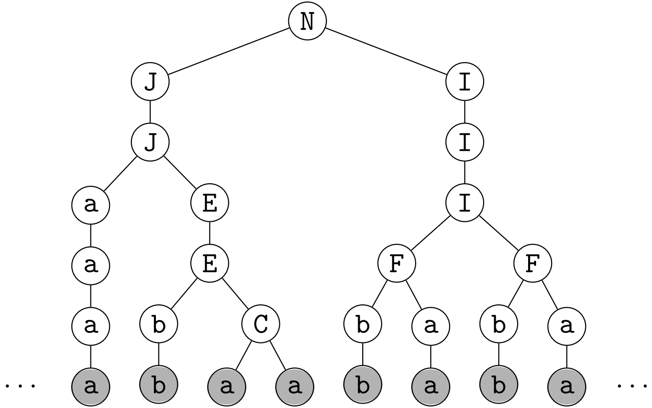
Context-Insensitive Nodes

Some nodes in the parse tree are preserved in any context.



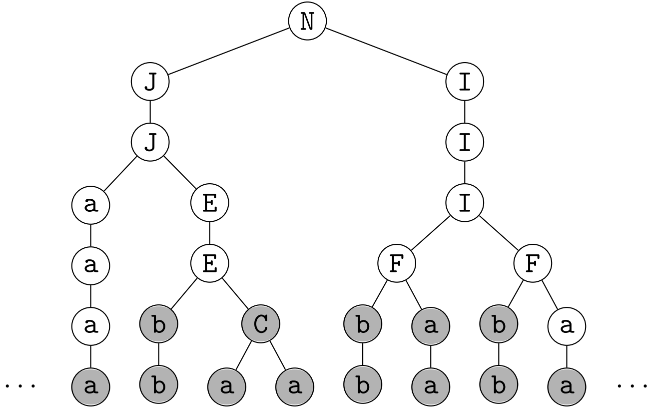
Context-Insensitive Nodes

Some nodes in the parse tree are preserved in any context.



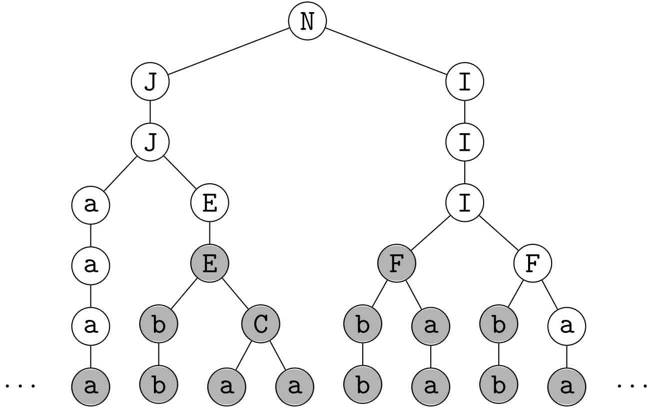
Context-Insensitive Nodes

Some nodes in the parse tree are preserved in any context.



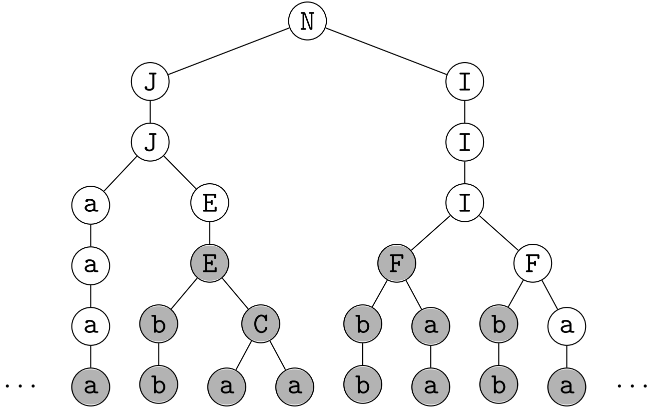
Context-Insensitive Nodes

Some nodes in the parse tree are preserved in any context.



Context-Insensitive Nodes

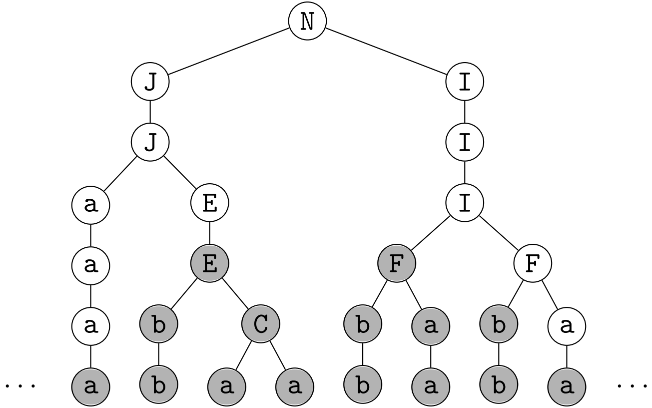
Some nodes in the parse tree are preserved in any context.



Context-insensitive decomposition $D = \mathtt{aEFba}$

Context-Insensitive Nodes

Some nodes in the parse tree are preserved in any context.



Context-insensitive decomposition $D = \mathtt{aEFba}$

In general, $|\mathrm{RLE}(D)| \le 2\textsc{Depth}(w)$.
$\mathcal{O}(\textsc{Depth}(w))$ branching context-sensitive nodes.

Update Implementation

Lemma (Building over a decomposition)

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

Update Implementation

Lemma (Building over a decomposition)

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

- `make_string`: Build over the decomposition into letters.

Update Implementation

Lemma (Building over a decomposition)

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

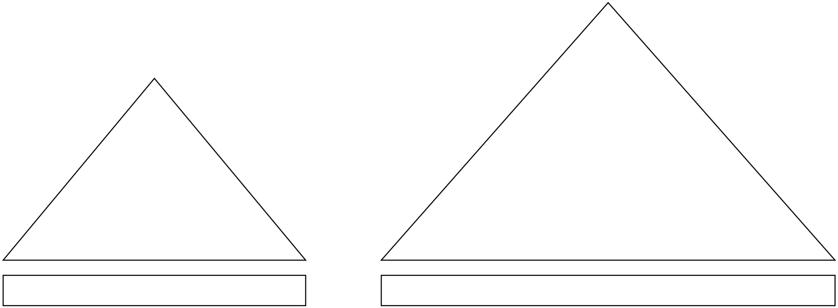
- `make_string`: Build over the decomposition into letters.
- `concat`, `split`: Extract the context-insensitive decompositions and build over them.

Update Implementation

Lemma (Building over a decomposition)

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

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

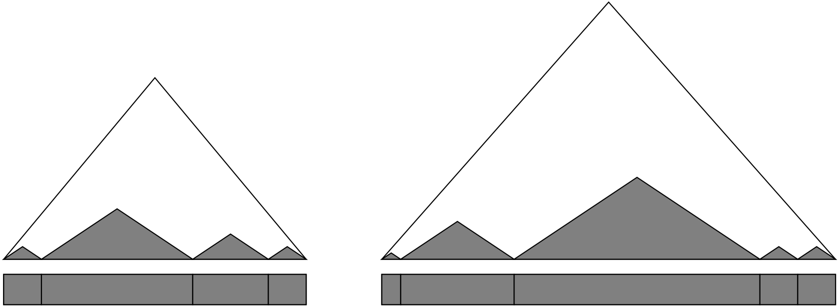


Update Implementation

Lemma (Building over a decomposition)

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

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

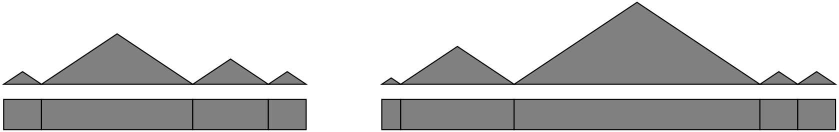


Update Implementation

Lemma (Building over a decomposition)

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

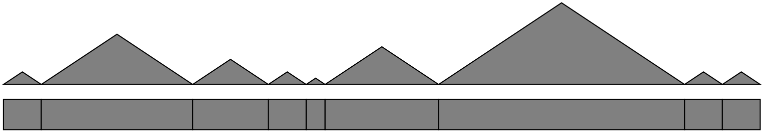
- `make_string`: Build over the decomposition into letters.
- `concat`, `split`: Extract the context-insensitive decompositions and build over them.



Lemma (Building over a decomposition)

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

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

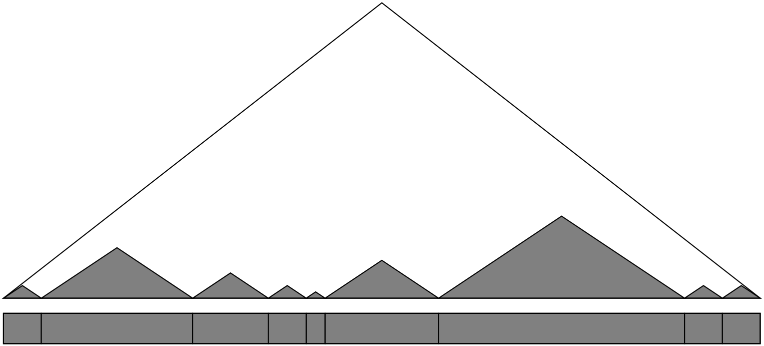


Update Implementation

Lemma (Building over a decomposition)

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

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



Further Extensions

1. Indexing dynamic strings:
   - find all occurrences of a given pattern in the strings $w \in \mathcal{W}$;

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.

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 $\mathcal{W}$;

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.

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:

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 $\texttt{drop\_string}(w)$ 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 $\mathcal{W}$;

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 primitive root and canonize it with respect to cyclic equivalence.

Summary

Data Structure	Ours	Impossible
`make_string`	$\mathcal{O}(\lvert w\rvert + \log n)$	$o(\lvert w\rvert \log n)$
`concat`	$\mathcal{O}(\log n)$	$o(\log n)$
`split`	$\mathcal{O}(\log n)$	$o(\log n)$
`equal`	$\mathcal{O}(1)$	$o(\log n)$
`compare`	$\mathcal{O}(1)$	-
`LCP`	$\mathcal{O}(1)$	-
persistent	YES	NO
	Las Vegas	Monte Carlo
	worst-case	amortized

Summary

Data Structure	Ours	Impossible
`make_string`	$\mathcal{O}(\lvert w\rvert + \log n)$	$o(\lvert w\rvert \log n)$
`concat`	$\mathcal{O}(\log n)$	$o(\log n)$
`split`	$\mathcal{O}(\log n)$	$o(\log n)$
`equal`	$\mathcal{O}(1)$	$o(\log n)$
`compare`	$\mathcal{O}(1)$	-
`LCP`	$\mathcal{O}(1)$	-
persistent	YES	NO
	Las Vegas	Monte Carlo
	worst-case	amortized

Thank you for your attention!