Practical Random Access to SLP-Compressed Texts

Travis Gagie, Tomohiro I, Giovanni Manzini,
Gonzalo Navarro, Hiroshi Sakamoto,
Louisa Seelbach Benkner, and Yoshimasa Takabatake

Introduction

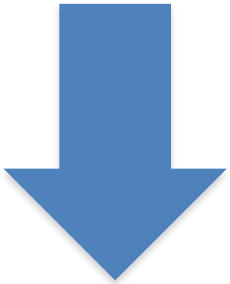
- Grammar compression is a powerful tool for processing repetitive texts in compressed space
  - random access [Bille et al., 2015]
  - index [Claude and Navarro, 2010, 2012]
  - rank/select [Navarro and Ordóñez, 2014]
  - etc.
- Finding the smallest grammar is NP-hard, but there are practical grammar compressors
  - RePair [Larsson and Moffat, 2000]
  - SOLCA [Takabatake et al., 2017]
  - etc.

Introduction

- In the approach with Prefix-Free Parsing (PFP) technique, RePair (and potentially other compressors) becomes really scalable

T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto and Y. Takabatake
Rpair: Rescaling RePair with Rsync, In *Proc. SPIRE 2019*, pp. 35-44, 2019.

59 GB file containing 1000 copies of chromosome 19



Reasonable time and space
- 21 minutes
- 7 GB of workspace

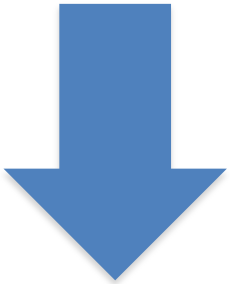
50 MB（0.1%）

Introduction

- In the approach with Prefix-Free Parsing (PFP) technique, RePair (and potentially other compressors) becomes really scalable

T. Gagie, T. I, G. Manzini, G. Navarro, H. Sakamoto and Y. Takabatake
Rpair: Rescaling RePair with Rsync, In *Proc. SPIRE 2019*, pp. 35-44, 2019.

59 GB file containing 1000 copies of chromosome 19



Reasonable time and space
- 21 minutes
- 7 GB of workspace

50 MB（0.1%）

We propose a new practical encoding that supports fast random access on grammar compressed space

Straight-line Program (SLP)

- Consider a CFG that derives a single string, which can be seen as a compressed representation of the string

Rules
$X_1 \rightarrow \texttt{a}\ \ \texttt{a}$
$X_2 \rightarrow \texttt{b}\ \ \texttt{c}$
$X_3 \rightarrow X_1 X_1$
$X_4 \rightarrow X_2\ \ \texttt{b}$
$S \rightarrow X_3 X_4 X_4 X_3$

For simplicity, rules are binary except the starting variable S

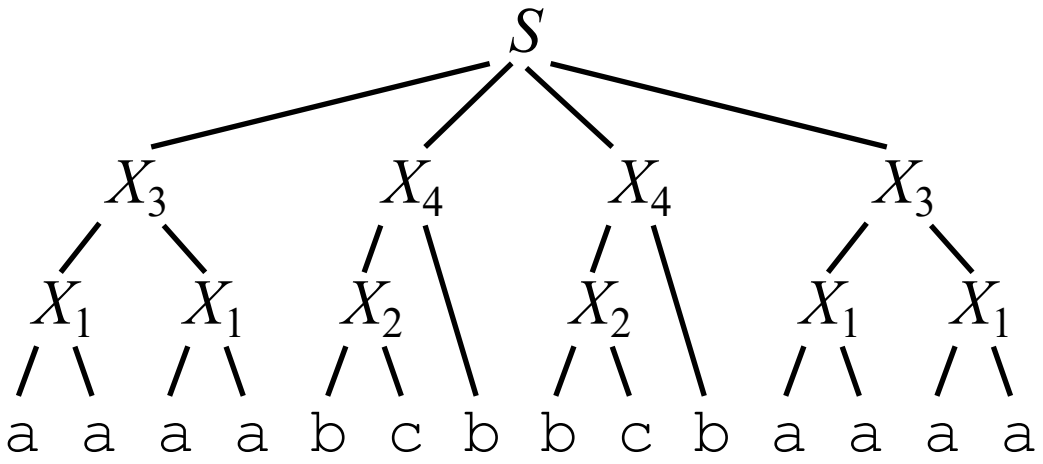
Straight-line Program (SLP)

- Consider a CFG that derives a single string, which can be seen as a compressed representation of the string

Rules
$X_1 \rightarrow \mathtt{a}\ \ \mathtt{a}$
$X_2 \rightarrow \mathtt{b}\ \ \mathtt{c}$
$X_3 \rightarrow X_1 X_1$
$X_4 \rightarrow X_2\ \ \mathtt{b}$
$S \rightarrow X_3 X_4 X_4 X_3$

decomp.

Derivation tree



For simplicity, rules are binary except the starting variable S

Random access on SLPs

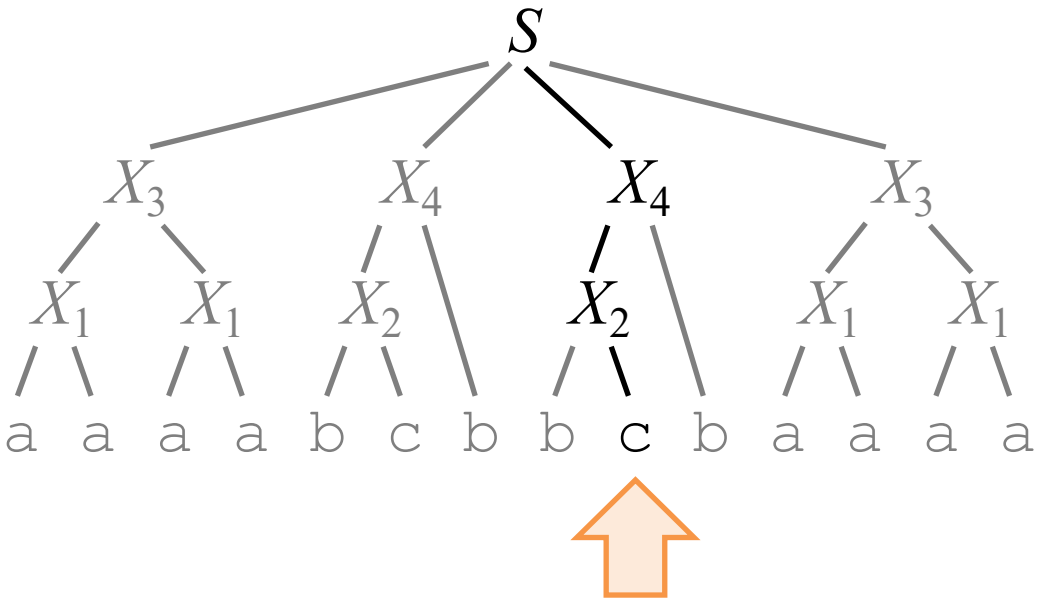
- Store expansion length $L(\cdot)$ for each var, and descend to destination on implicit derivation tree by binary search

Rules	$L(\cdot)$
$X_1 \to$ a a	**2**
$X_2 \to$ b c	**2**
$X_3 \to X_1 X_1$	**4**
$X_4 \to X_2$ b	**3**
$S \to X_3 X_4 X_4 X_3$	

psum: **4 7 10 14**

For S, store prefix sums of $L(\cdot)$

Derivation tree



Contribution: New encoding of SLPs that utilizes $L(\cdot)$ information

- Existing methods encode rules and $L(\cdot)$ separately
- We use $L(\cdot)$ to (hopefully) reduce the bit size of rules
  - Related idea: Using structural information of labeled trees to encode them succinctly [Gańczorz, 2020]

Rules	$L(\cdot)$
$X_1 \to \mathtt{a}\ \ \mathtt{a}$	**2**
$X_2 \to \mathtt{b}\ \ \mathtt{c}$	**2**
$X_3 \to X_1 X_1$	**4**
$X_4 \to X_2\ \ \mathtt{b}$	**3**
$S \to X_3 X_4 X_4 X_3$	

psum: **4 7 10 14**

chr19x1000: 59 GB

	Size [MB]	access [μs]
Naive	217	2.7
Existing	86	27.0
Proposed	80	7.5

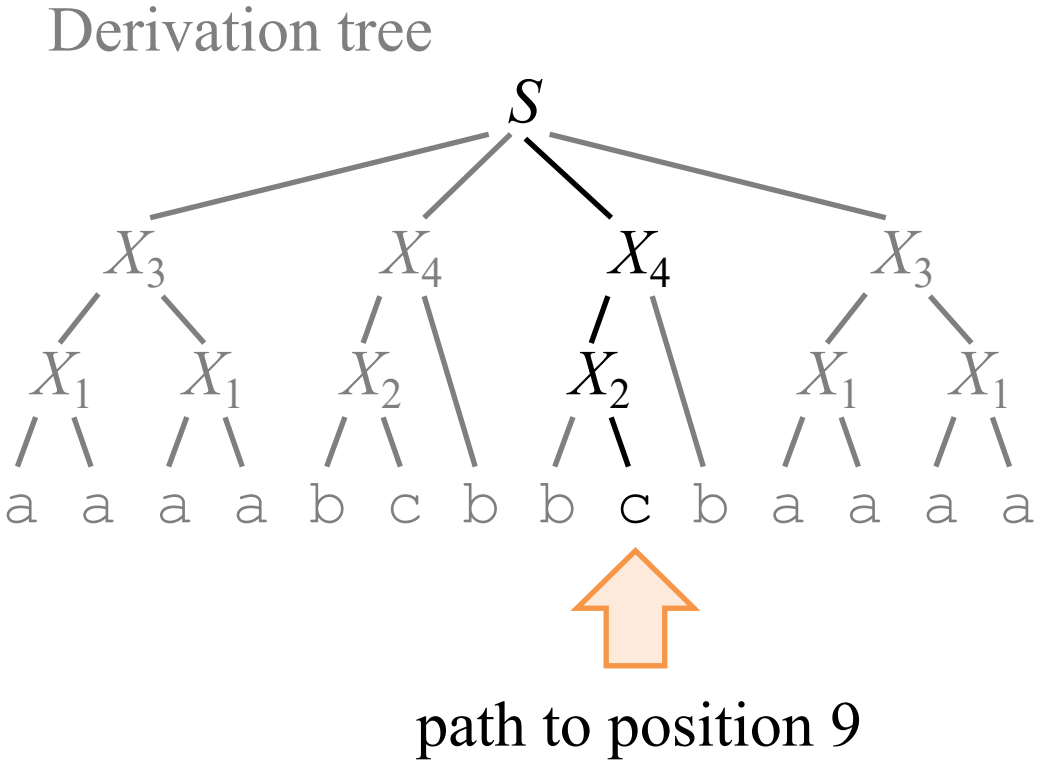
New encoding of SLPs

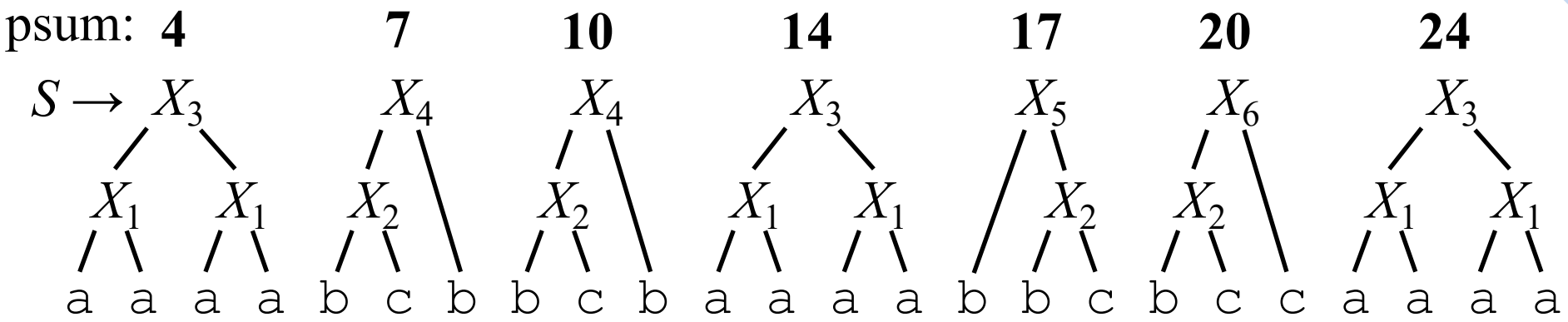
Key observation and idea

- When descending the tree, we first access $L(\cdot)$ and then var
- Partition vars into groups having the same $L(\cdot)$ and reduce bits to represent vars by **storing offsets** in a group

Rules	$L(\cdot)$
$X_1 \rightarrow$ a a	**2**
$X_2 \rightarrow$ b c	**2**
$X_3 \rightarrow X_1 X_1$	**4**
$X_4 \rightarrow X_2$ b	**3**
$S \rightarrow X_3 X_4 X_4 X_3$	

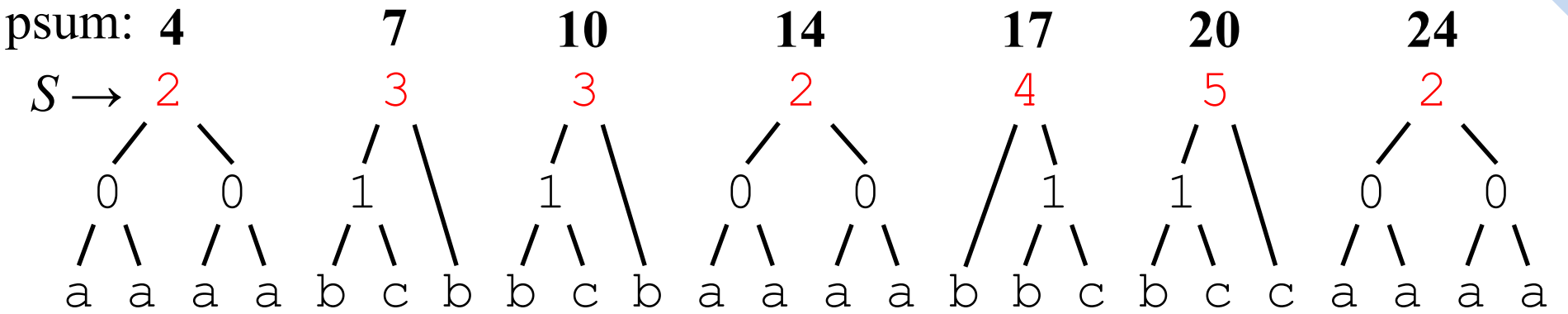
psum: **4 7 10 14**





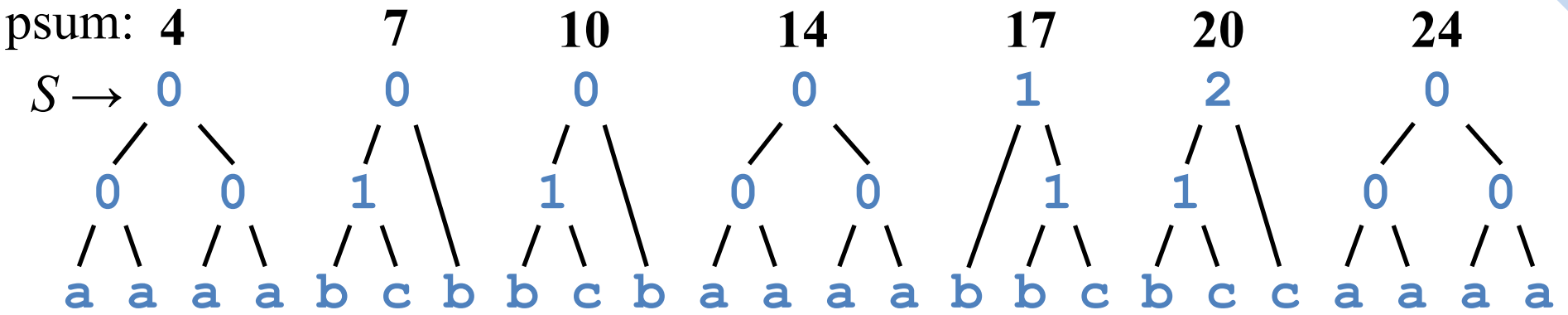
Rules
$X_1 \rightarrow$ a a
$X_2 \rightarrow$ b c
$X_3 \rightarrow X_1 X_1$
$X_4 \rightarrow X_2$ b
$X_5 \rightarrow$ b X_2
$X_6 \rightarrow X_2$ c

Bigger example



- Assume each var is represented by integer ID
- If integers in rules are small, variable-length codes (e.g. gamma codes) will encode them well

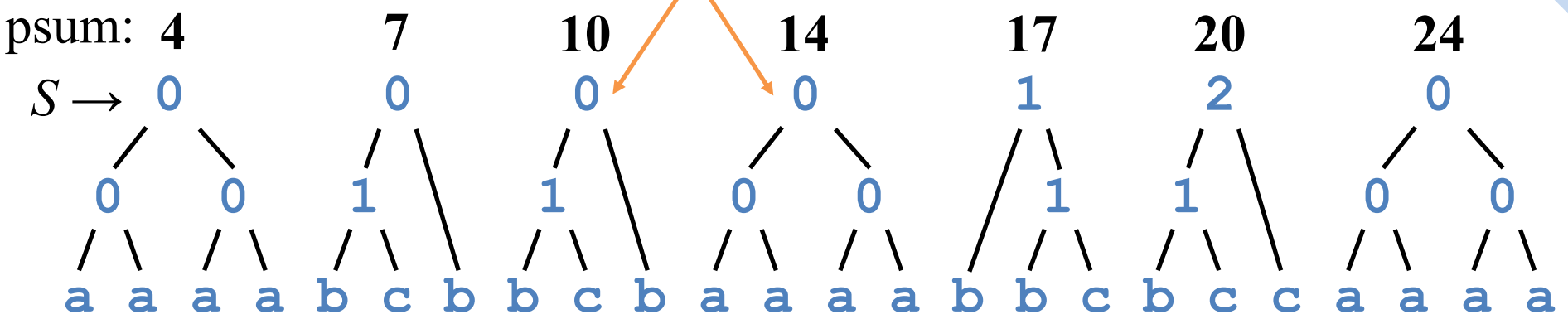
ID	Rules
0	a a
1	b c
2	0 0
3	1 b
4	b 1
5	1 c



- Replace ID with **offset** in a group having the same $L(\cdot)$

	offset	ID	Rules
$L(\cdot) = 2$	0	0	a a
	1	1	b c
$L(\cdot) = 4$	0	2	0 0
$L(\cdot) = 3$	0	3	1 b
	1	4	b 1
	2	5	1 c

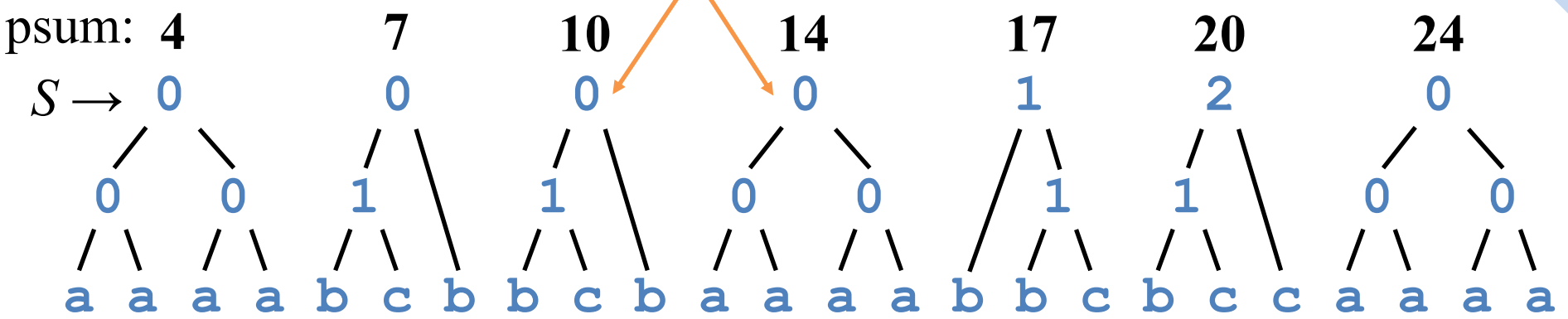
These 0s can be distinguished by expansion lengths



- Replace ID with **offset** in a group having the same $L(\cdot)$

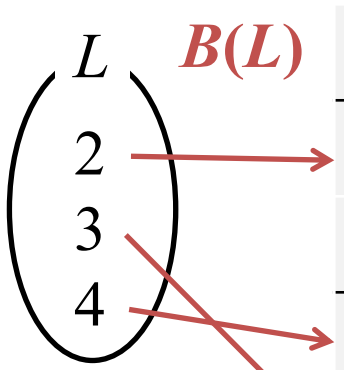
	offset	ID	Rules
$L(\cdot) = 2$	0	0	a a
	1	1	b c
$L(\cdot) = 4$	0	2	0 0
$L(\cdot) = 3$	0	3	1 b
	1	4	b 1
	2	5	1 c

These 0s can be distinguished by expansion lengths



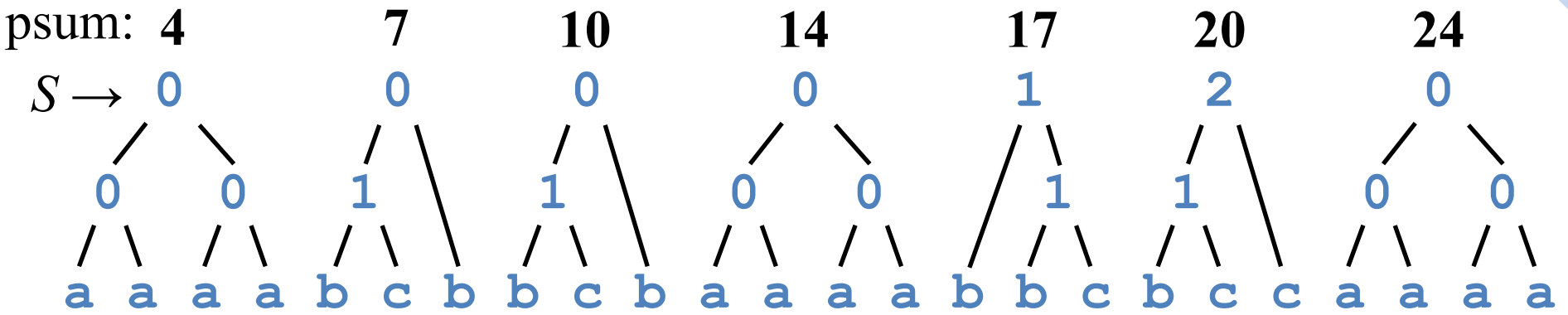
- Replace ID with **offset** in a group having the same $L(\cdot)$

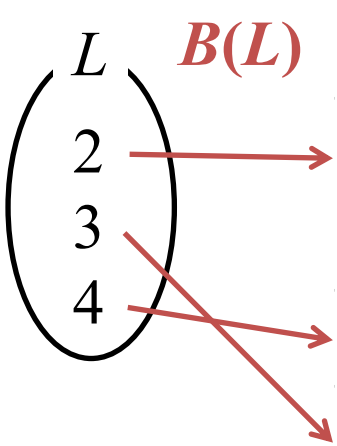
$$\text{ID} = B(L) + \text{offset}$$



offset	ID	Rules
0	0	a a
1	1	b c
0	2	0 0
0	3	1 b
1	4	b 1
2	5	1 c

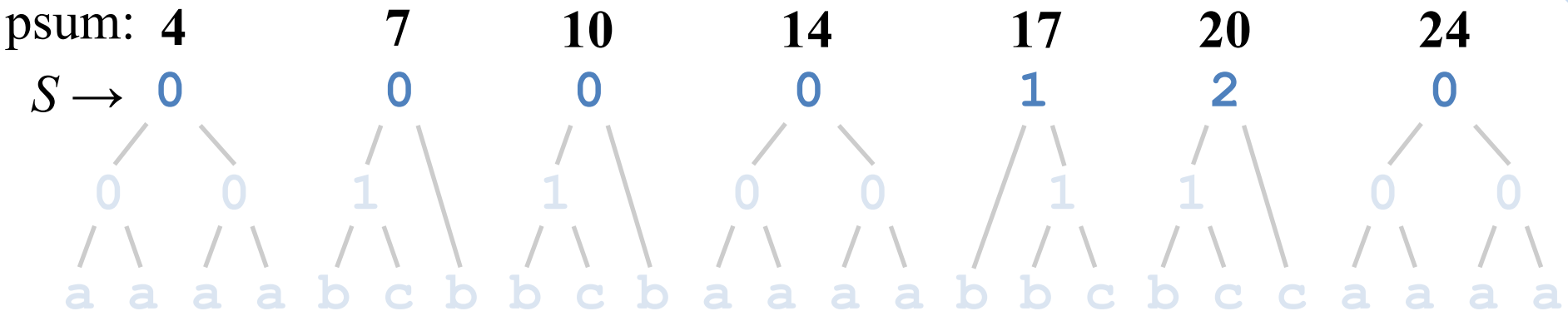
- Function $B(L)$ maps expansion length L to the base of group with $L(\cdot) = L$
- $B(L)$ can be implemented with Perfect Minimal Hash (PMH) and rank data structure

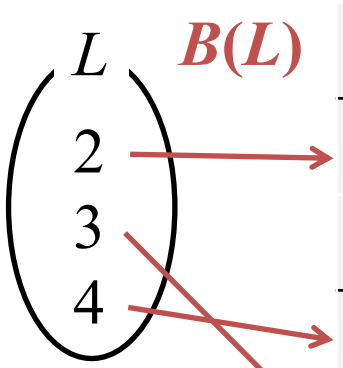




offset	ID	Rules	$LL(\cdot)$
0	0	a a	1
1	1	b c	1
0	2	0 0	2
0	3	1 b	2
1	4	b 1	1
2	5	1 c	2

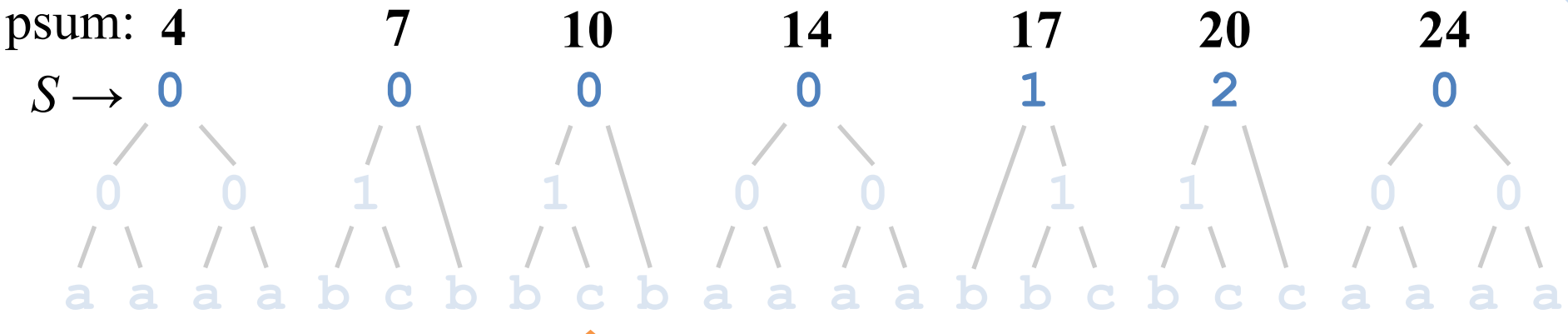
- For each var, store expansion length $LL(\cdot)$ of left child



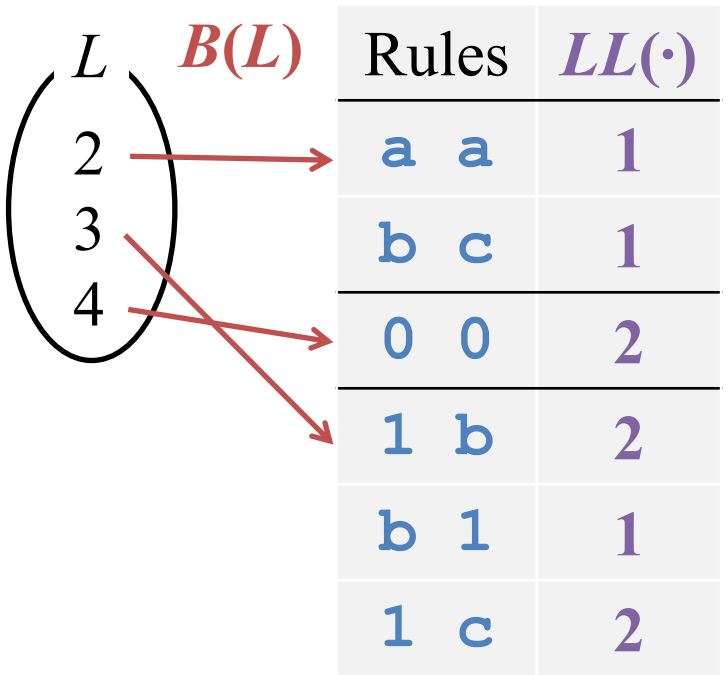


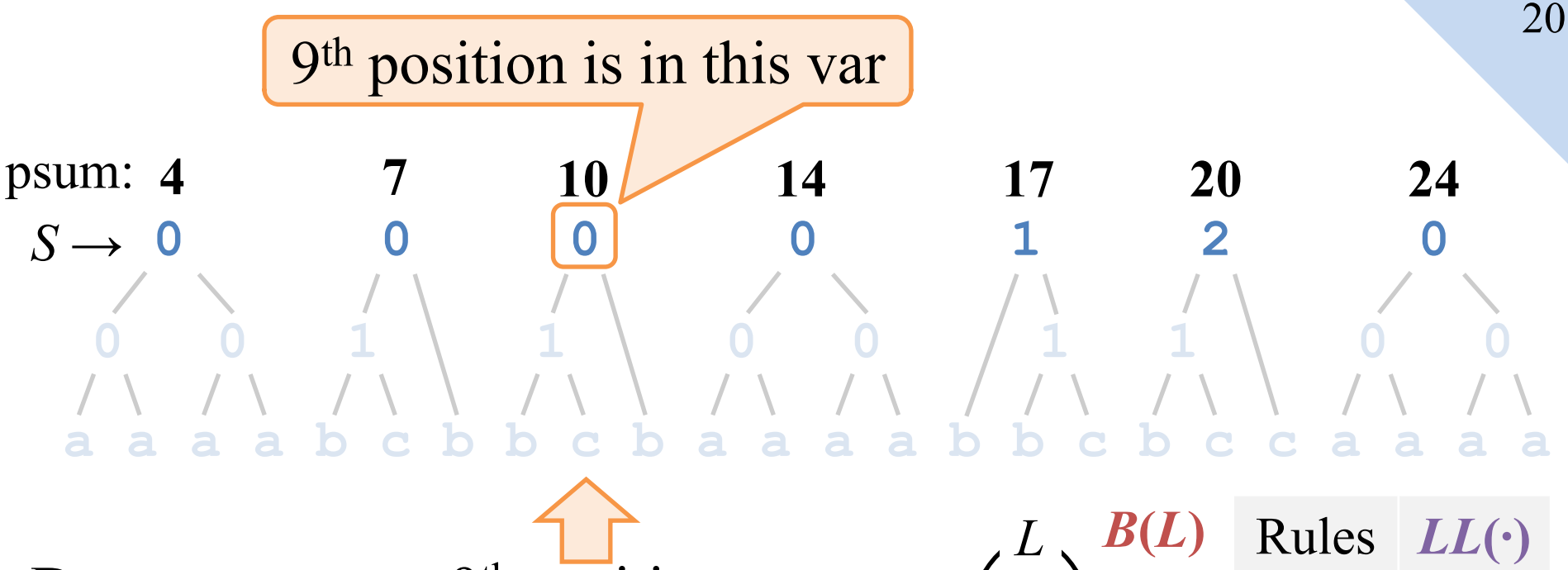
Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

What we store



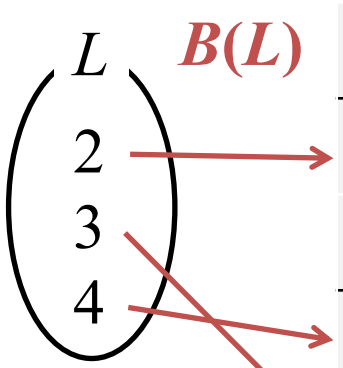
Demo: access to 9th position



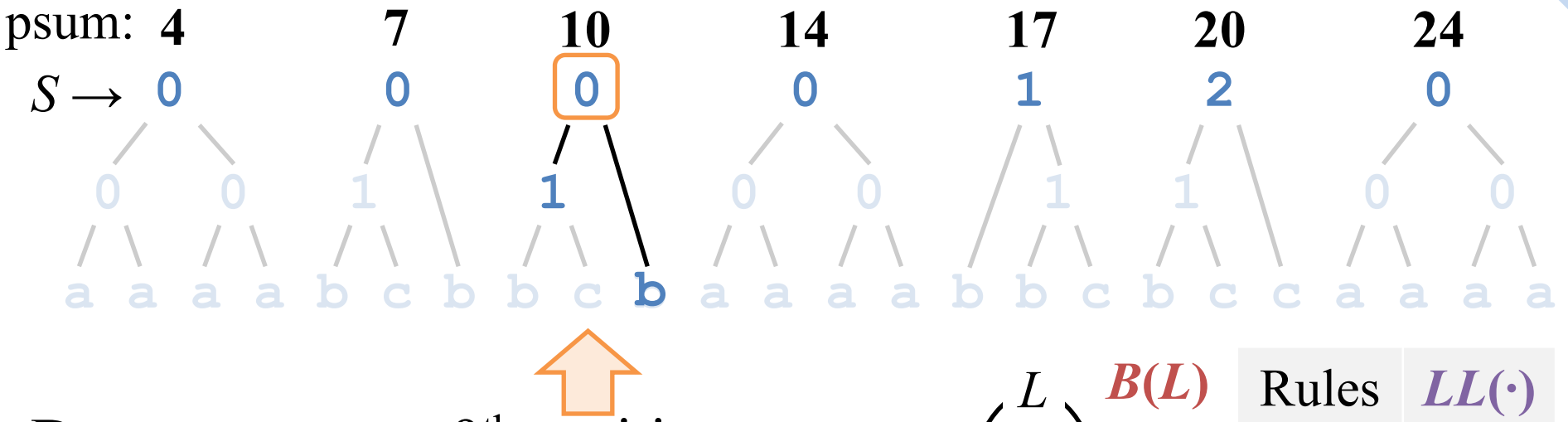


Demo: access to 9th position

- Predecessor query on psum gives $L = 3$, offset = 0

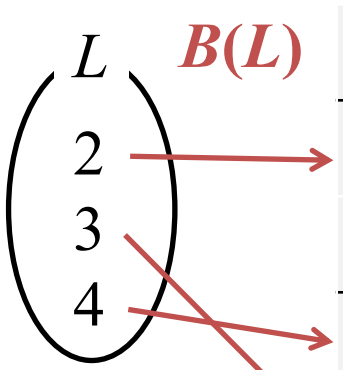


Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

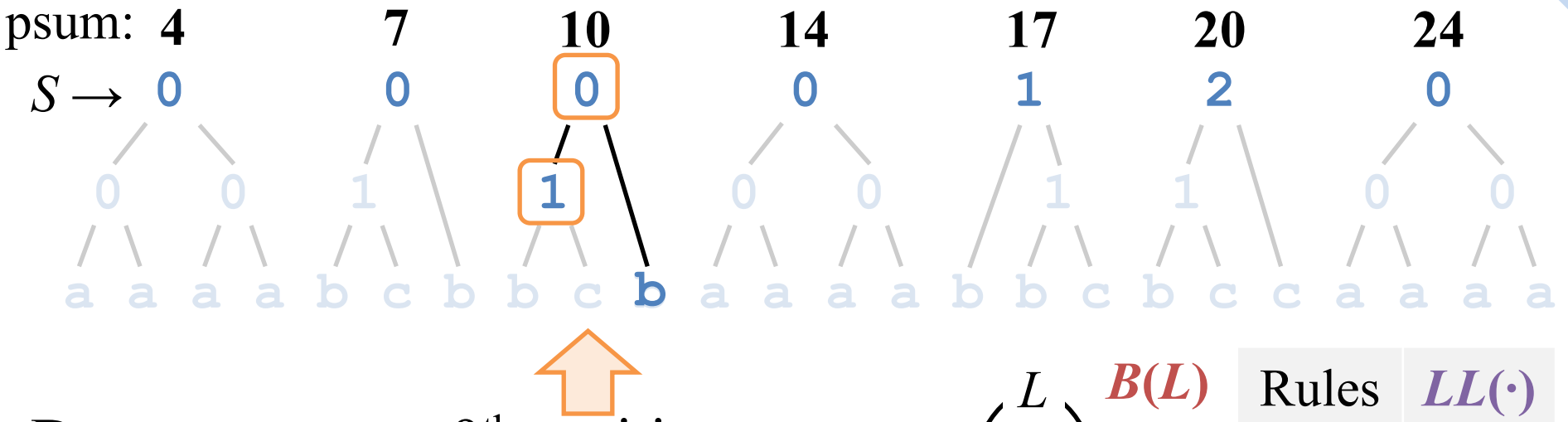


Demo: access to 9th position

- Predecessor query on psum gives $L = 3$, offset = 0
- $B(3) + 0$ leads to children's information

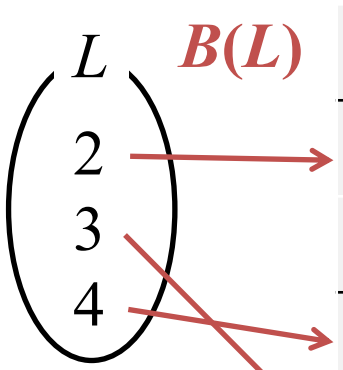


Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

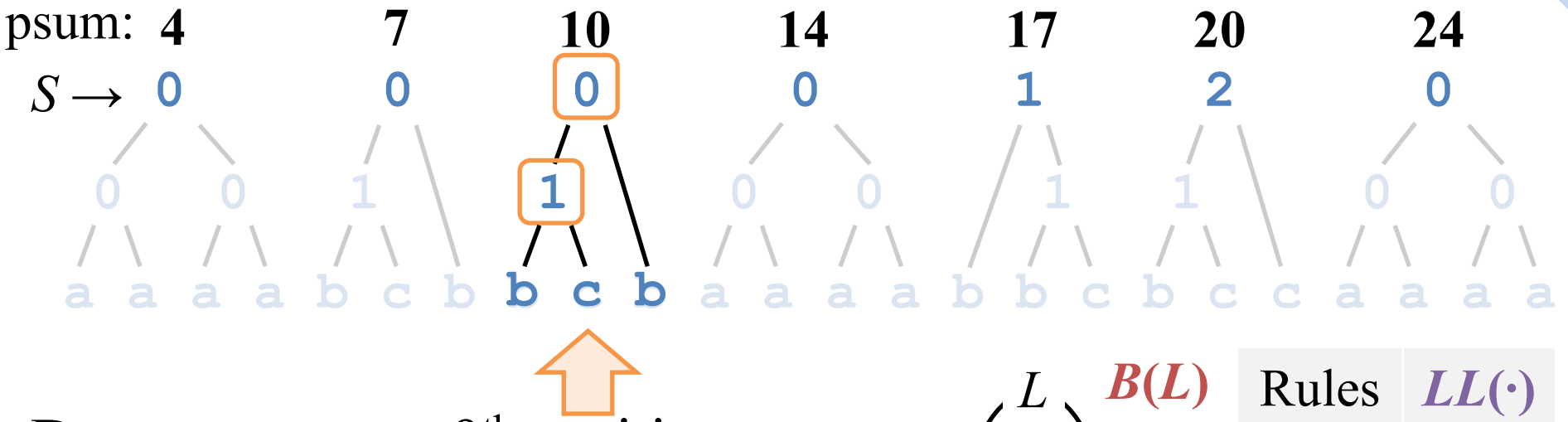


Demo: access to 9th position

- Predecessor query on psum gives $L = 3$, offset = 0
- $B(3) + 0$ leads to children's information
- Down to left child with $L = 2$, offset = 1

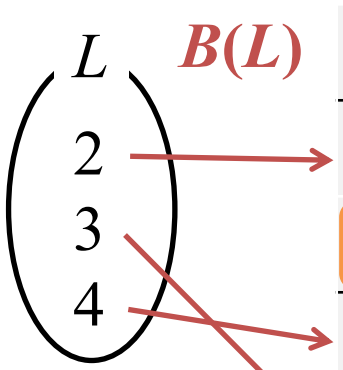


Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

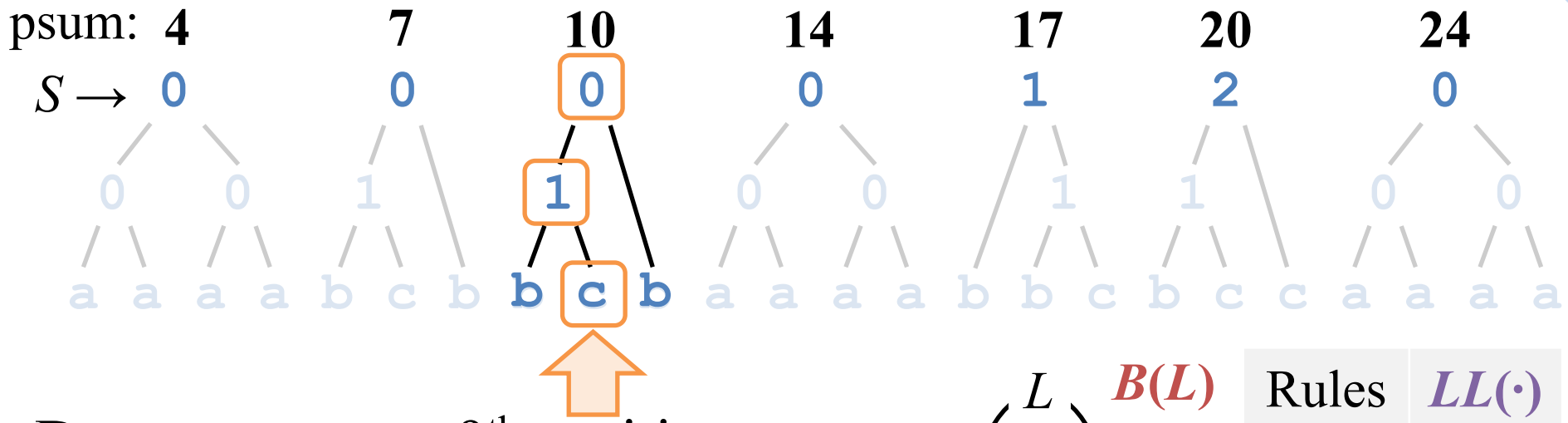


Demo: access to 9th position

- Predecessor query on psum gives $L = 3$, offset = 0
- $B(3)$ + 0 leads to children's information
- Down to left child with $L = 2$, offset = 1
- $B(2)$ + 1 leads to children's information

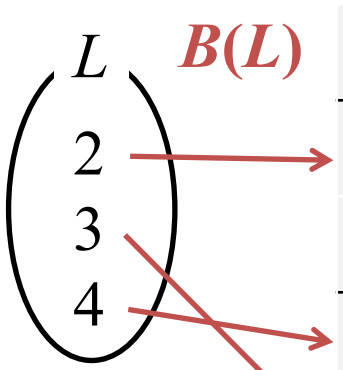


Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2



Demo: access to 9th position

- Predecessor query on psum gives $L = 3$, offset = 0
- $B(3)$ + 0 leads to children's information
- Down to left child with $L = 2$, offset = 1
- $B(2)$ + 1 leads to children's information
- Down to right child and get c



Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

Implementation notes

Re-ordering vars in a group for smaller encoding

$S \rightarrow$ `0000120`

Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

$S \rightarrow$ `0000210`

Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
1 c	2
b 1	1

Group vars having same $LL(\cdot)$ to sparsify $LL(\cdot)$ (accessed with rank data structure)

Re-ordering vars in a group for smaller encoding

$S \rightarrow$ `0000120`

Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
b 1	1
1 c	2

$S \rightarrow$ `0000210`

Rules	$LL(\cdot)$
a a	1
b c	1
0 0	2
1 b	2
1 c	2
b 1	1

$S \rightarrow$ `0000210`

Rules	$LL(\cdot)$
b c	1
a a	1
1 1	2
0 b	2
0 c	2
b 0	1

Group vars having same $LL(\cdot)$ to sparsify $LL(\cdot)$ (accessed with rank data structure)

Sort by frequency to give smaller offset to frequent vars

Experiments

Experiments

- Given SLPs compressed by RePair or Bigrepair, build data structure (encoding) for random access

datasets	description	size
chr19x1000	chromosome19	59 GB
salx11264	salmonella	57 GB
einstein	Wikipedia article	468 MB
kernel	source code	258 MB

} in repcorpus

- Compared with
  - NAIVE: store rules and $L(\cdot)$'s by fixed-length codes
  - MTSS: store rules with a succinct tree in $q \lg q + O(q)$ bits
    store information of $L(\cdot)$'s in $q \lg N - q$ bits

[Maruyama et al., 2013]

q: #vars + len of rhs of S

N: text length

	DS size [MB]			Construction time [ms]		
	NAIVE	MTSS	OURS	NAIVE	MTSS	OURS
chr19x1000	217 (0.37%)	86 (0.15%)	**81 (0.14%)**	524	4576	17649
salx11264	2896 (5.1%)	**799 (1.4%)**	957 (1.7%)	5457	53147	370175
einstein.en	1.90 (0.41%)	0.67 (0.14%)	**0.63 (0.14%)**	3	22	92
kernel	12.96 (5.0%)	**4.47 (1.7%)**	5.04 (2.0%)	30	158	866

Random access time [μs] to x consecutive chars for chr19x1000

	NAIVE	MTSS	OURS
$x = 1$	1.8	25.9	6.9
$x = 10$	2.2	29.6	9.3
$x = 100$	5.2	63.5	31.7
$x = 1000$	31.6	394.6	349.6

Conclusions

- We proposed a new encoding that uses expansion lengths to get smaller DS supporting fast random access on SLPs
- Experiments showed that the proposed method achieved
  - smallest DS size for some datasets,
  - reasonable access speed

- Application:
  - Matching statistics algorithm that requires random access in compressed space