DCC 2020

c-trie++: A Dynamic Trie Tailored for Fast Prefix Searches

Kazuya Tsuruta[1], Dominik Köppl[1,2],
Shunsuke Kanda[3], Yuto Nakashima[1],
Shunsuke Inenaga[1], Hideo Bannai[1], Masayuki Takeda[1]

1. Department of Informatics, Kyushu University, Fukuoka, Japan.
2. Japan Society for Promotion of Science.
3. RIKEN Center for Advanced Intelligence Project, Tokyo, Japan.

Problem Definition (1/4)

【Problem】Dynamic Prefix Search

Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern P, can compute the pair
a) $\max\{l : P[1..l\,] = T_i[1..l\,]$ for some $i \in [1..k]\}$ and
b) $I_P = \{i : T_i[1..l\,] = P[1..l\,]\}$ efficiently.

Example:

$S = \{T_1, T_2, T_3, T_4, T_5\}$

$T_1 =$ `idea`

$T_2 =$ `interface`

$T_3 =$ `internet`

$T_4 =$ `infinite`

$T_5 =$ `laboratory`

$P =$ `inter`

output = (5, {2, 3})

Problem Definition (2/4)

【Problem】Dynamic Prefix Search

Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern P, can compute the pair
a) $\max\{l : P[1..l\,] = T_i[1..l\,]$ for some $i \in [1..k]\}$ and
b) $I_P = \{i : T_i[1..l\,] = P[1..l\,]\}$ efficiently.

Example:

$S = \{T_1, T_2, T_3, T_4, T_5\}$

$T_1 =$ `idea`

$T_2 =$ `interface`

$T_3 =$ `internet`

$T_4 =$ `infinite`

$T_5 =$ `laboratory`

$P =$ `inner`

output $= (2, \{2, 3, 4\})$

Problem Definition (3/4)

【Problem】Dynamic Prefix Search

Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern P, can compute the pair
a) $\max\{l : P[1..l\,] = T_i[1..l\,]$ for some $i \in [1..k]\}$ and
b) $I_P = \{i : T_i[1..l\,] = P[1..l\,]\}$ efficiently.

Example:

$S = \{T_1, T_2, T_3, T_4, T_5, T_6\}$

$T_1 =$ `idea`

$T_2 =$ `interface`

$T_3 =$ `internet`

$T_4 =$ `infinite`

$T_5 =$ `laboratory`

$T_6 =$ `indexing`

Supports insertion of a string into S.

insert(T_6)

Problem Definition (4/4)

【Problem】Dynamic Prefix Search

Maintain a data structure for a dynamic set $S = \{T_1..T_k\}$ of strings that, given a query pattern P, can compute the pair

a) $\max\{l : P[1..l\,] = T_i[1..l\,]$ for some $i \in [1..k]\}$ and

b) $I_P = \{i : T_i[1..l\,] = P[1..l\,]\}$ efficiently.

Example:

$S = \{T_1, T_2, T_3, T_4, \sout{T_5}, T_6\}$

$T_1 =$ `idea`

$T_2 =$ `interface`

$T_3 =$ `internet`

$T_4 =$ `infinite`

~~$T_5 =$ `laboratory`~~

$T_6 =$ `indexing`

Supports deletion of a string from S.

delete(T_5)

Introduction (1/3)

- prefix search applications
  - **input method editors**
  - query auto-completion
  - range query filtering

(entering japani on an Android phone)

Introduction (2/3)

- prefix search applications
  - input method editors
  - **query auto-completion**
  - range query filtering

(entering japani on google)

https://www.google.com/

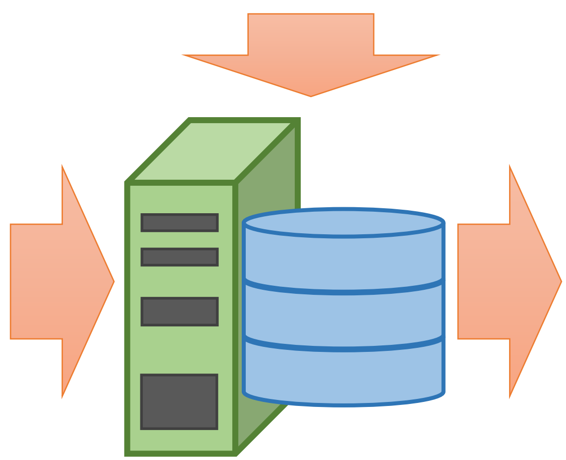
Introduction (3/3)

- prefix search applications
  - input method editors
  - query auto-completion
  - **range query filtering**

UserQueries

User	Date	Word
661	12/03/2020	refund-ticket
457	11/03/2020	trip-cancellation
139	01/03/2020	corona-virus
:	:	:

```
SELECT User, Word
  FROM UserQueries
  WHERE Word LIKE ‘japani%’
  AND …
```



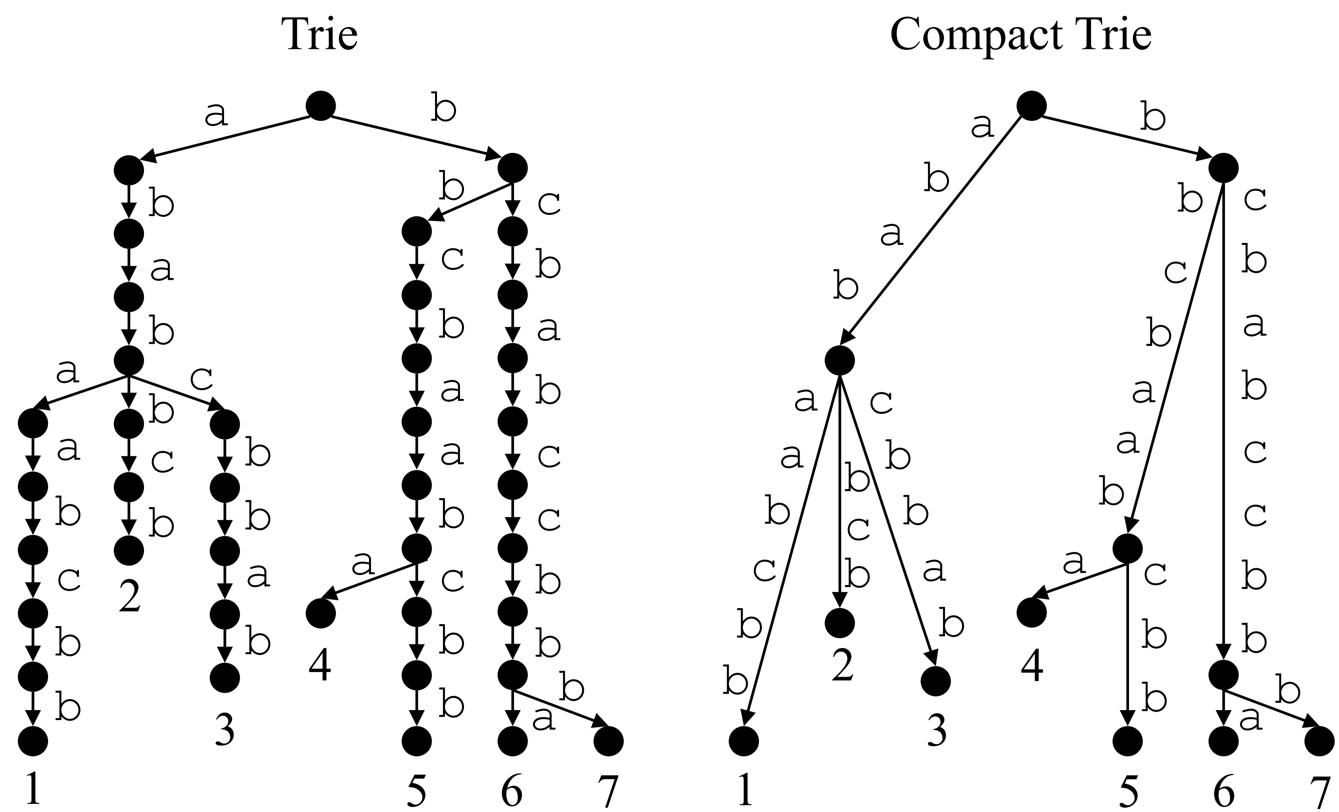
User	Word
79	japanican
83	japanification
89	japanistry
97	japani

Compact Trie [Morrison, 1968]

- Trie : represents strings where common prefixes are compressed to a single path (front encoding).
- Compact Trie : reduces the number of nodes by replacing branchless path segments with a single edge.

Strings

$T_1 =$ `ababaabcbb`
$T_2 =$ `ababbcb`
$T_3 =$ `ababcbbab`
$T_4 =$ `bbcbaaba`
$T_5 =$ `bbcbaabcbb`
$T_6 =$ `bcbabccbba`
$T_7 =$ `bcbabccbbb`



Previous Works for Dynamic Prefix Search (1/2)

	Space [bits]	Prefix Search Time in Expectation
Trie	$O(\|T\| \log \|T\|)$	$O(m + occ)$
Compact Trie [Morrison, 1968]	$\|T\| \log \sigma + \Theta(k \log k)$	$O(m + occ)$

$|T|$: trie size (front encoding size)
$k = |S|$, $m = |P|$, $\sigma = |\Sigma|$, Σ : alphabet, occ : number of occurrences

Previous Works for Dynamic Prefix Search (2/2)

	Space [bits]	Prefix Search Expected Time
Trie	$O(\|T\| \log \|T\|)$	$O(m + occ)$
Compact Trie [Morrison, 1968]	$\|T\| \log \sigma + \Theta(k \log k)$	$O(m + occ)$
Z-Fast Trie [Belazzougui et al., 2010]	$\|T\| \log \sigma + \Theta(kw)$	$O(m / \alpha + occ + \log (m \log \sigma))$
Packed C-Trie [Takagi et al., 2017]	$\|T\| \log \sigma + \Theta(kw)$	$O(m / \alpha + occ + \log w)$

$|T|$: trie size (front encoding size)
$n = \sum_i |T_i|$, $k = |S|$, $m = |P|$, $\sigma = |\Sigma|$, Σ : alphabet, occ : number of occurrences
w : machine word size ($w = \Omega(\log n)$), $\alpha = O(w / \log \sigma)$

Our Contribution for Dynamic Prefix Search

	Space [bits]	Prefix Search Expected Time
Trie	$O(\|T\| \log \|T\|)$	$O(m + occ)$
Compact Trie [Morrison, 1968]	$\|T\| \log \sigma + \Theta(k \log k)$	$O(m + occ)$
Z-Fast Trie [Belazzougui et al., 2010]	$\|T\| \log \sigma + \Theta(kw)$	$O(m / \alpha + occ + \log (m \log \sigma))$
Packed C-Trie [Takagi et al., 2017]	$\|T\| \log \sigma + \Theta(kw)$	$O(m / \alpha + occ + \log w)$
C-Trie++ [Ours]	$\|T\| \log \sigma + \Theta(kw)$	$O(m / \alpha + occ + \log \min\{\alpha, m\})$

$\alpha < w$ always holds.

$|T|$: trie size (front encoding size)

$n = \sum_i |T_i|$, $k = |S|$, $m = |P|$, $\sigma = |\Sigma|$, Σ : alphabet, occ : number of occurrences

w : machine word size ($w = \Omega(\log n)$), $\alpha = O(w / \log \sigma)$

Word Packing

In the word RAM model with word size w bits,
we can compare (=, <, >) two $O(w)$ bits integers in $O(1)$ time.

⇒ Let $\alpha = w / \log \sigma$.

i	d	e	a
01101001	01100100	01100101	01100001

a character uses $\lceil \log \sigma \rceil$ bits
⇒ strings of length α use $O(w)$ bits

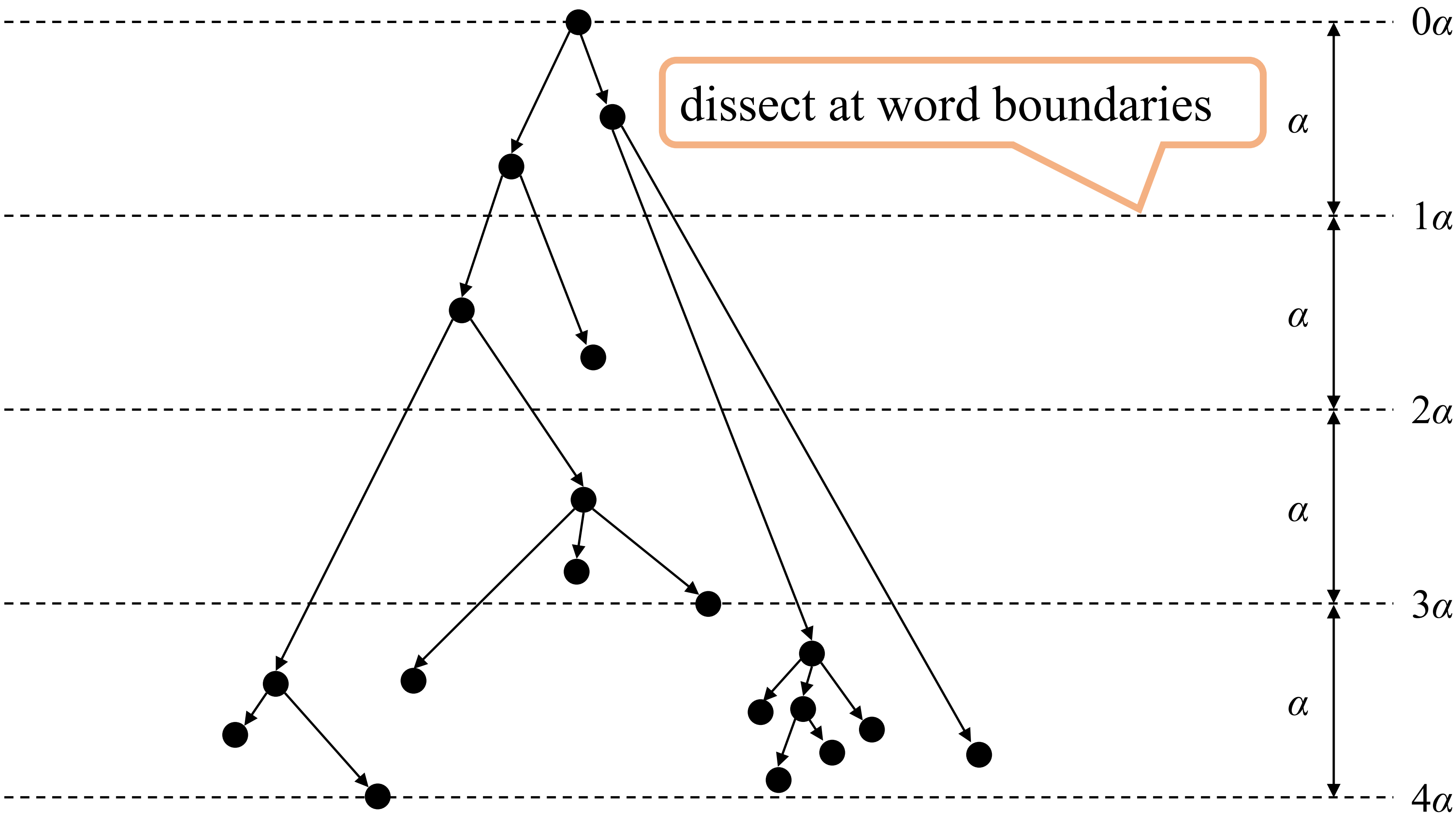
e.g.
64-bits architecture
$\sigma = 256$
$w = 32$
$\alpha = 4$

We can compare two strings of length α in $O(1)$ time.

⇒ We can compare two strings of length m in $O(m / \alpha)$ time.

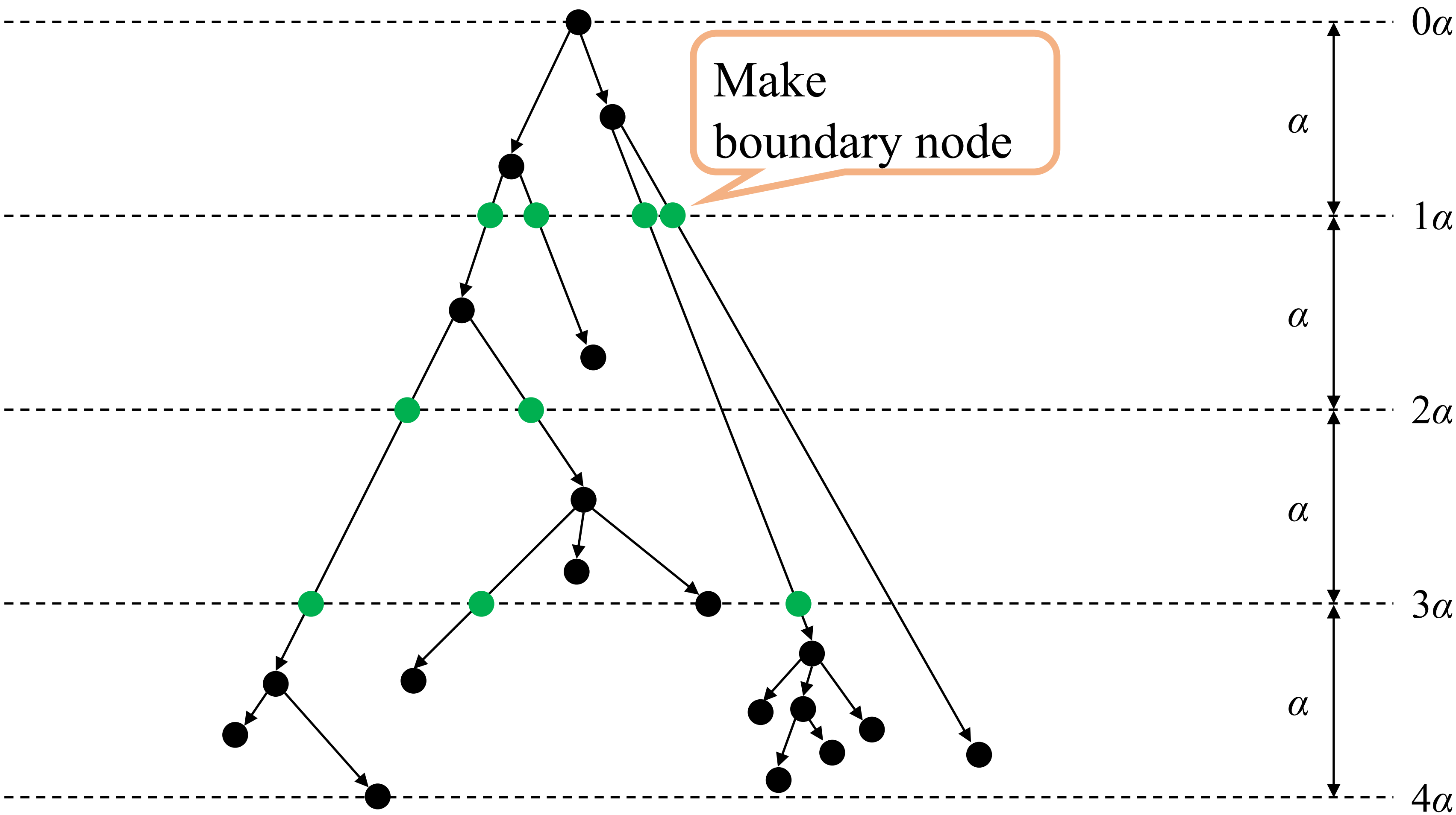
Introduction of Micro Trie (1/4)

Compact Trie



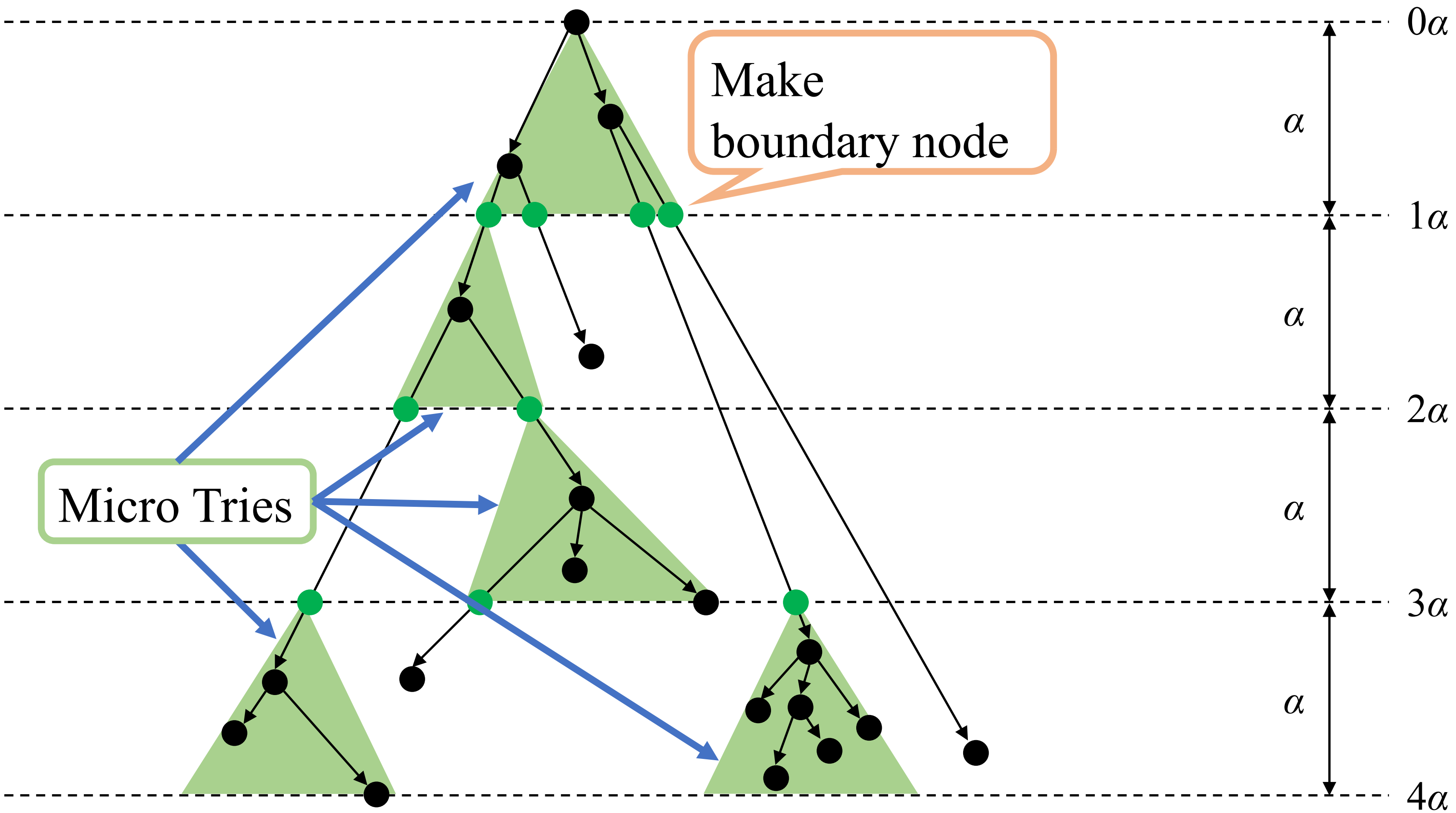
Introduction of Micro Trie (2/4)

Compact Trie



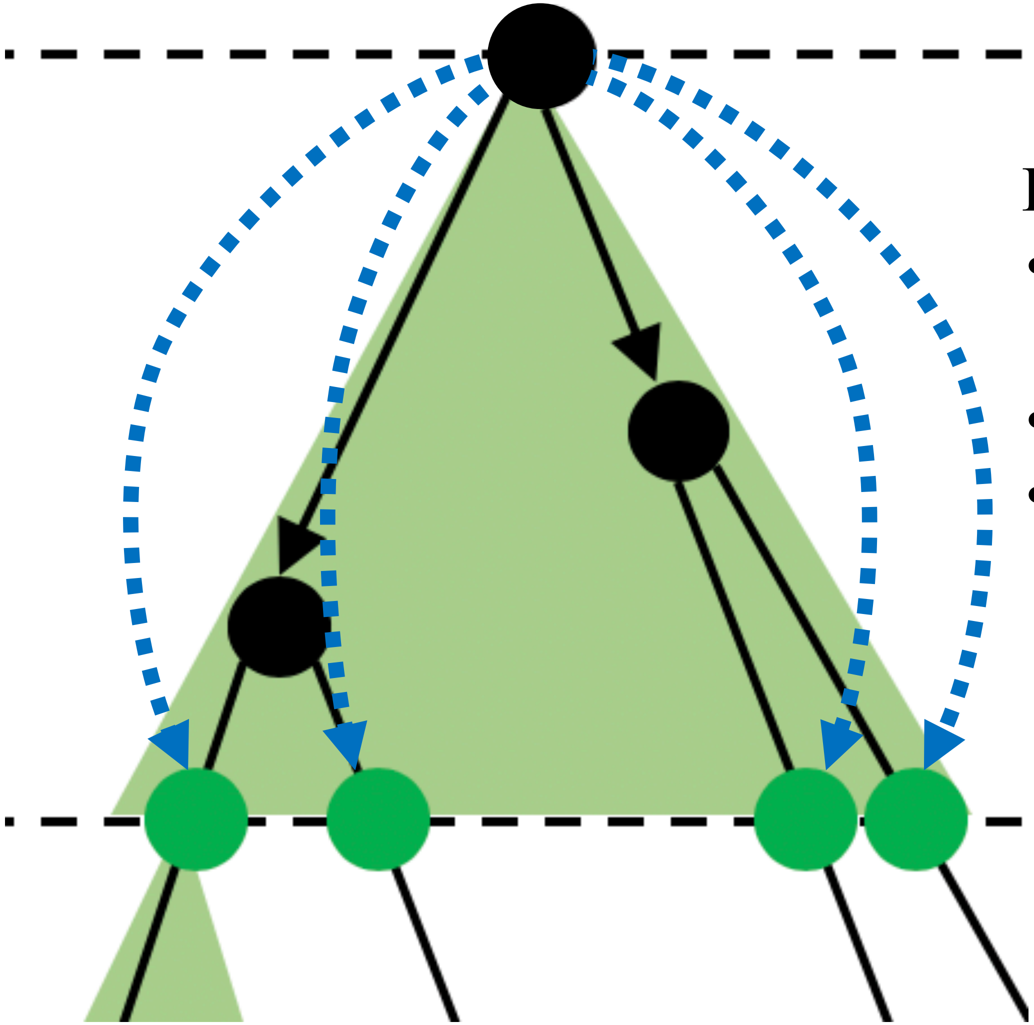
Introduction of Micro Trie (3/4)

C-Trie++



Introduction of Micro Trie (4/4)

Equip each Micro Trie with a Hash Table

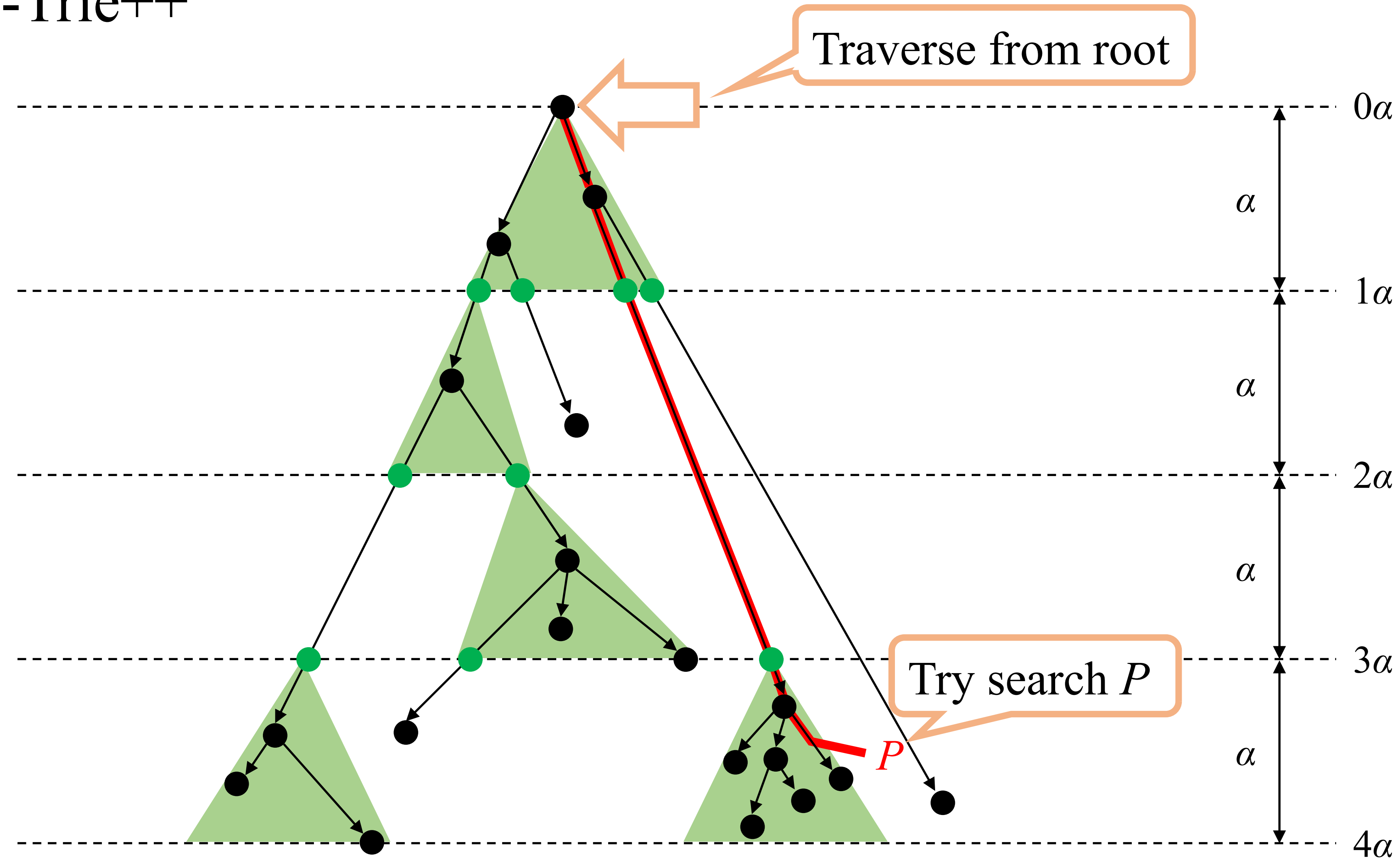


Hash Table:

- jump from root to leaf in $O(1)$ expected time.
- key: string of length α
- value: leaf node

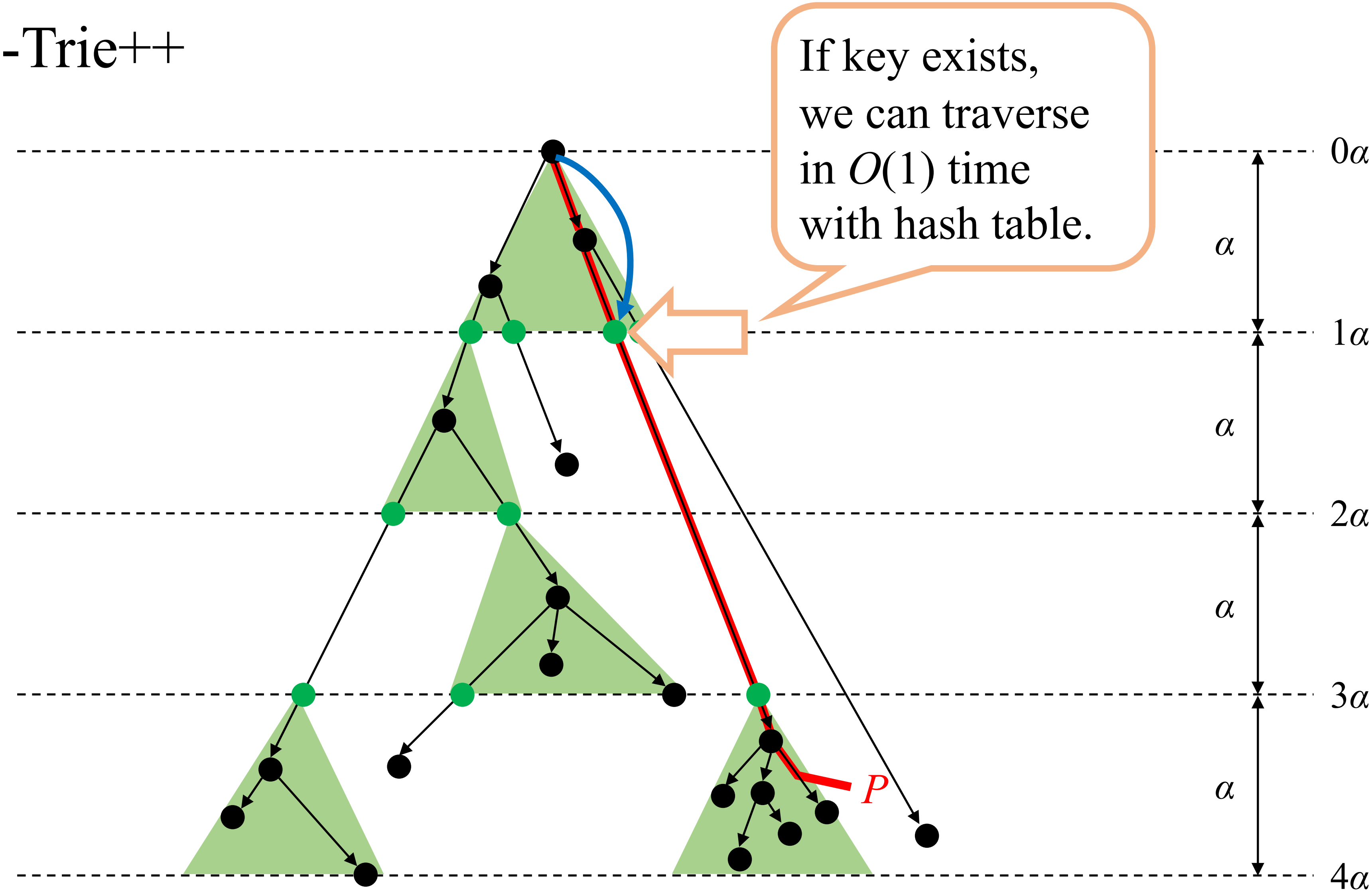
Trie Traversal (1/5)

C-Trie++



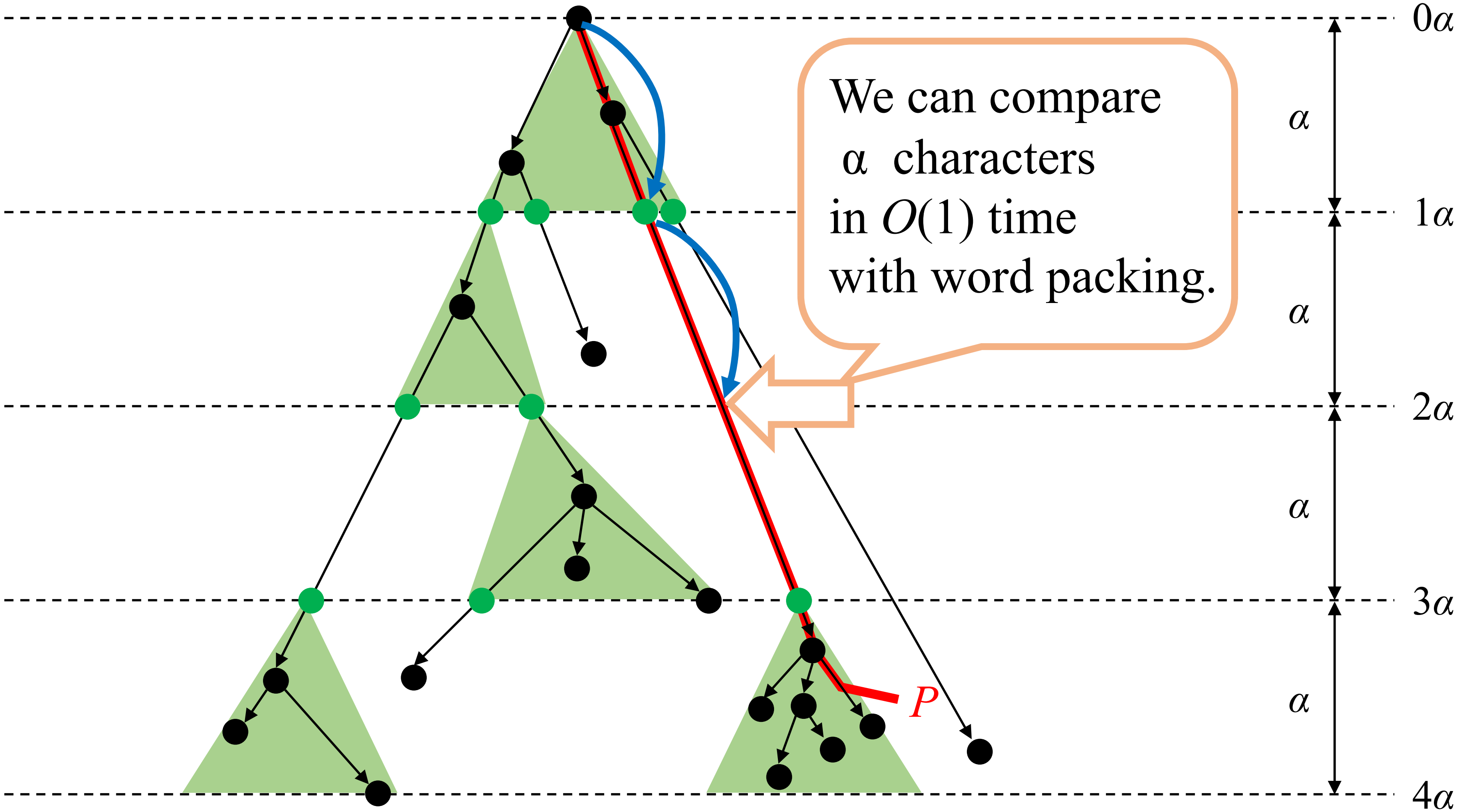
Trie Traversal (2/5)

C-Trie++



Trie Traversal (3/5)

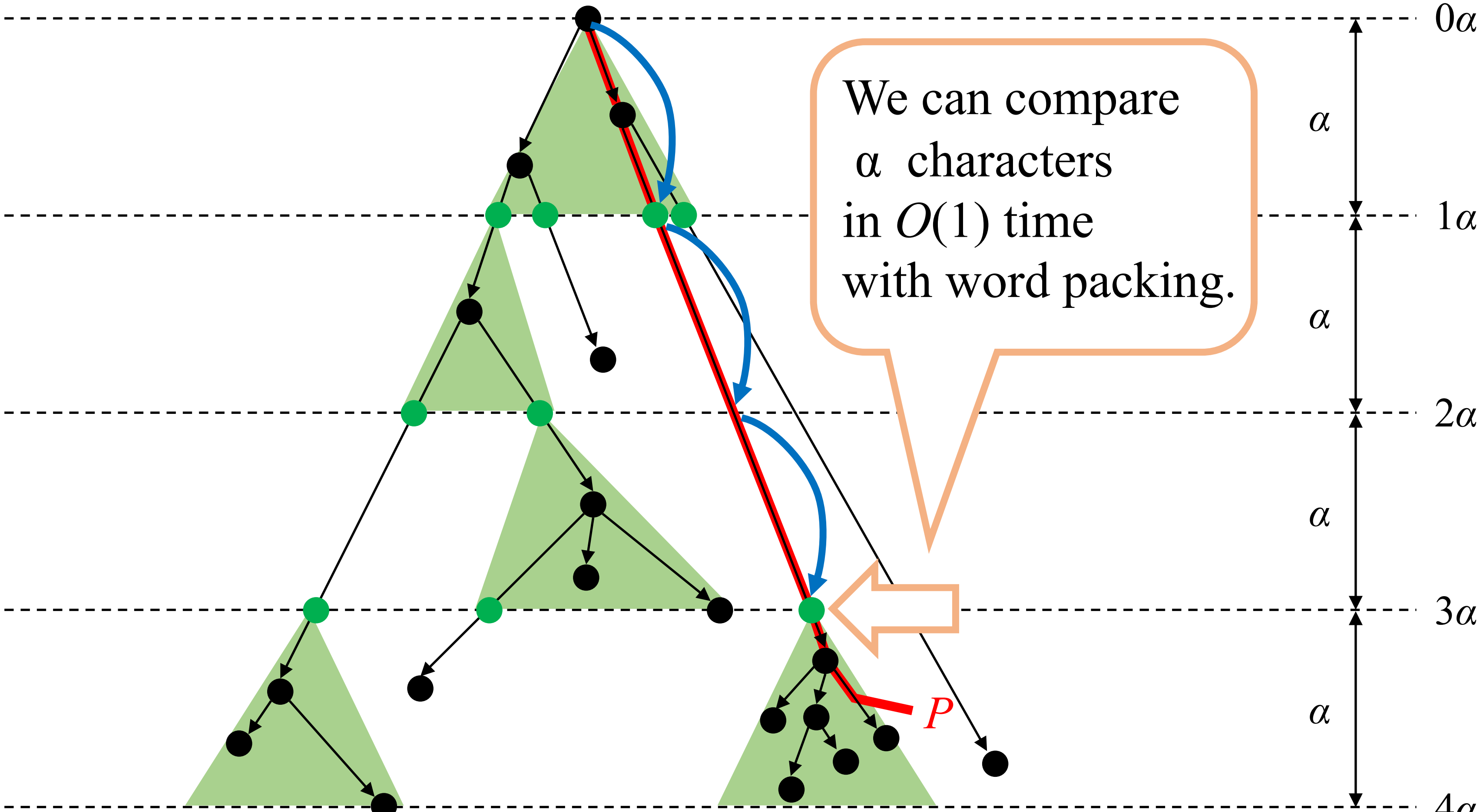
C-Trie++



We can compare α characters in $O(1)$ time with word packing.

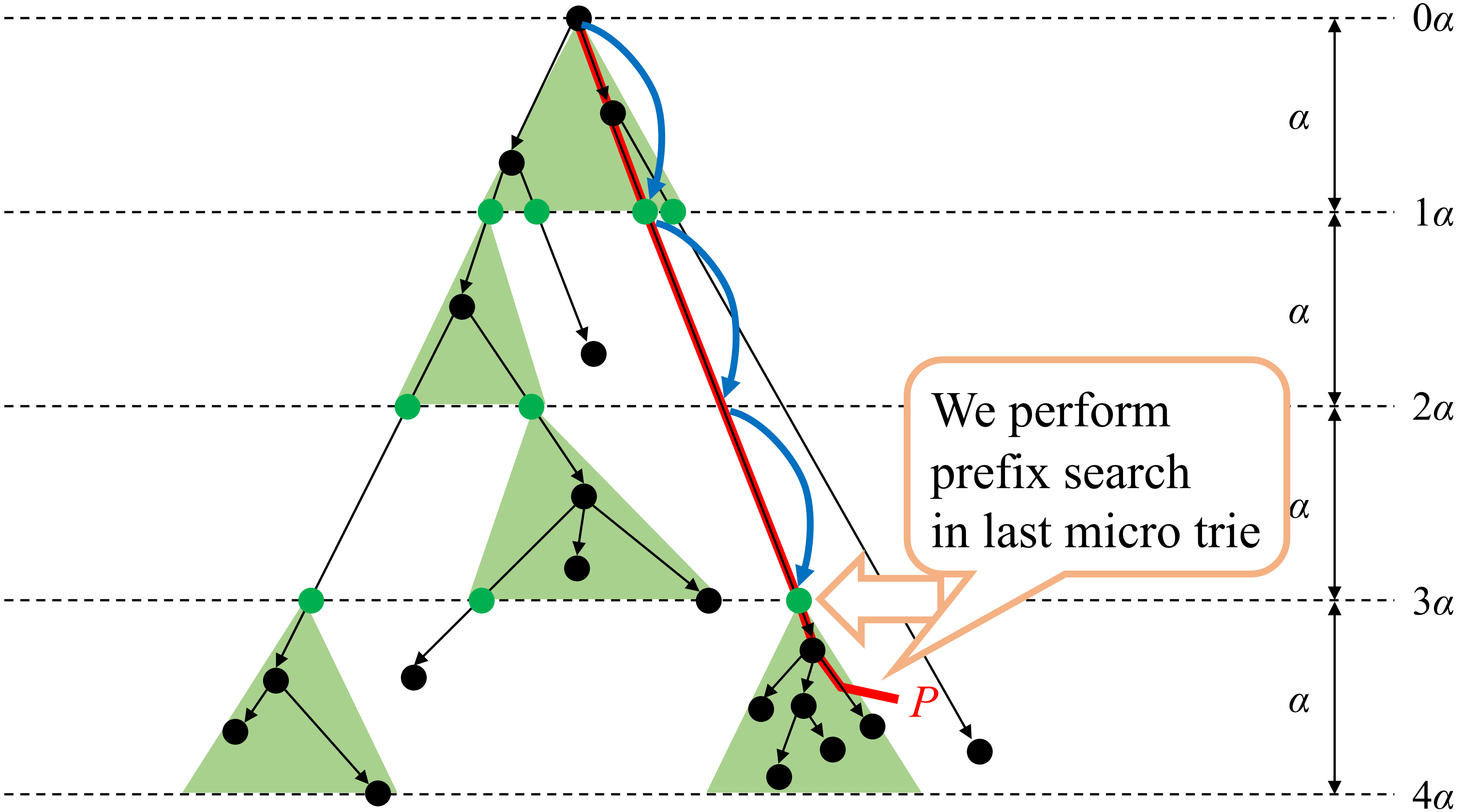
Trie Traversal (4/5)

C-Trie++



Trie Traversal (5/5)

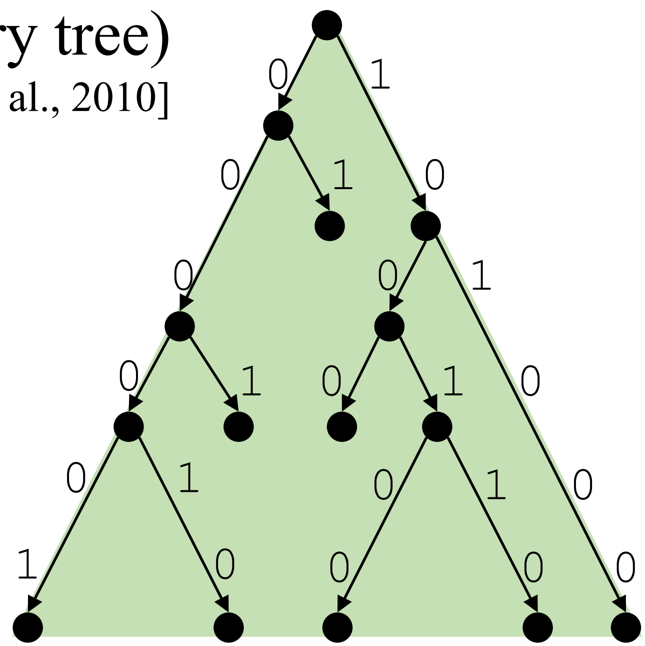
C-Trie++



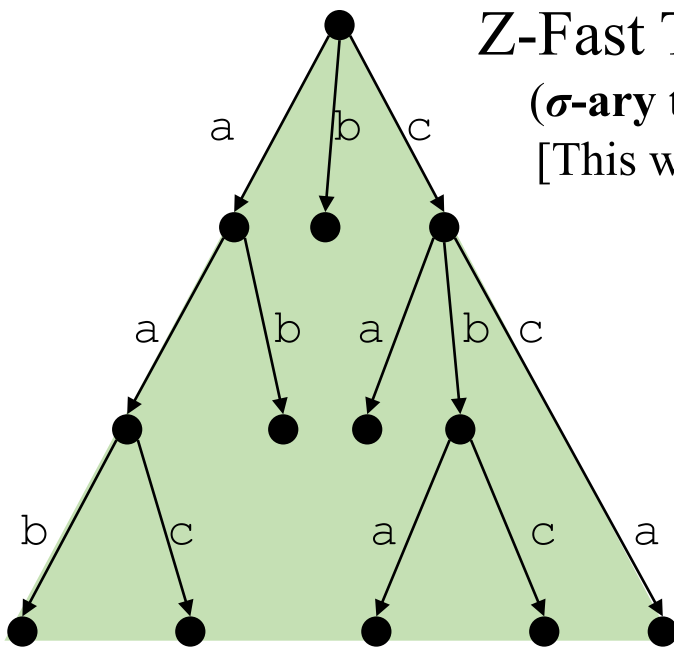
Dynamic Prefix Search in Micro Tries

- We improve prefix search (expected) time in micro trie
  - Belazzougui et al., 2010 : $O(\log (m \log \sigma) + occ)$
  - Takagi et al., 2017 : $O(\log w + occ)$
  - **This work** : $\boldsymbol{O(\log \min\{\alpha, m\} + occ)}$

Z-Fast Trie
(binary tree)
[Belazzougui et al., 2010]



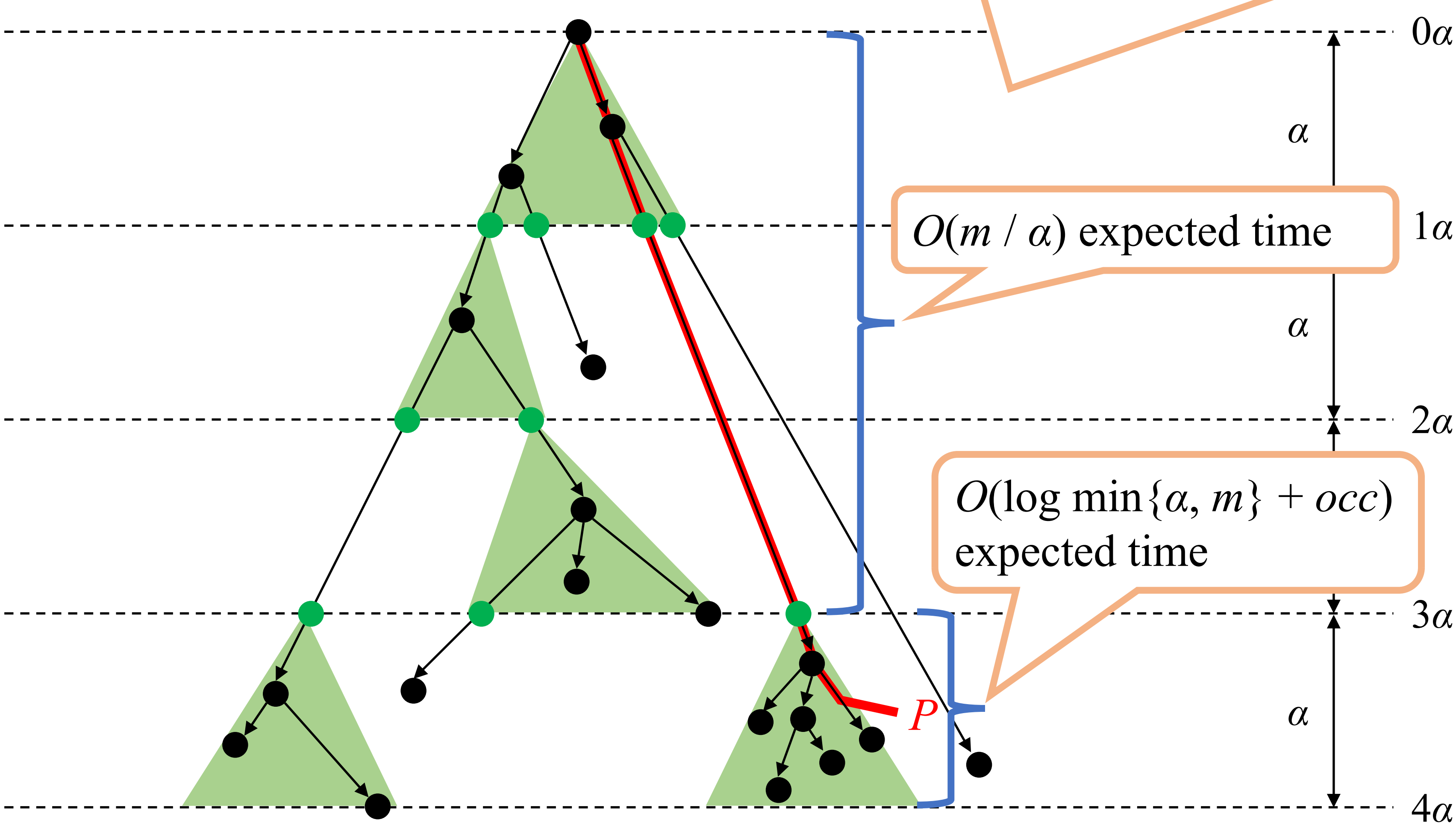
Alphabet Aware
Z-Fast Trie
(**$\boldsymbol{\sigma}$-ary tree**)
[This work]



Prefix Search Time

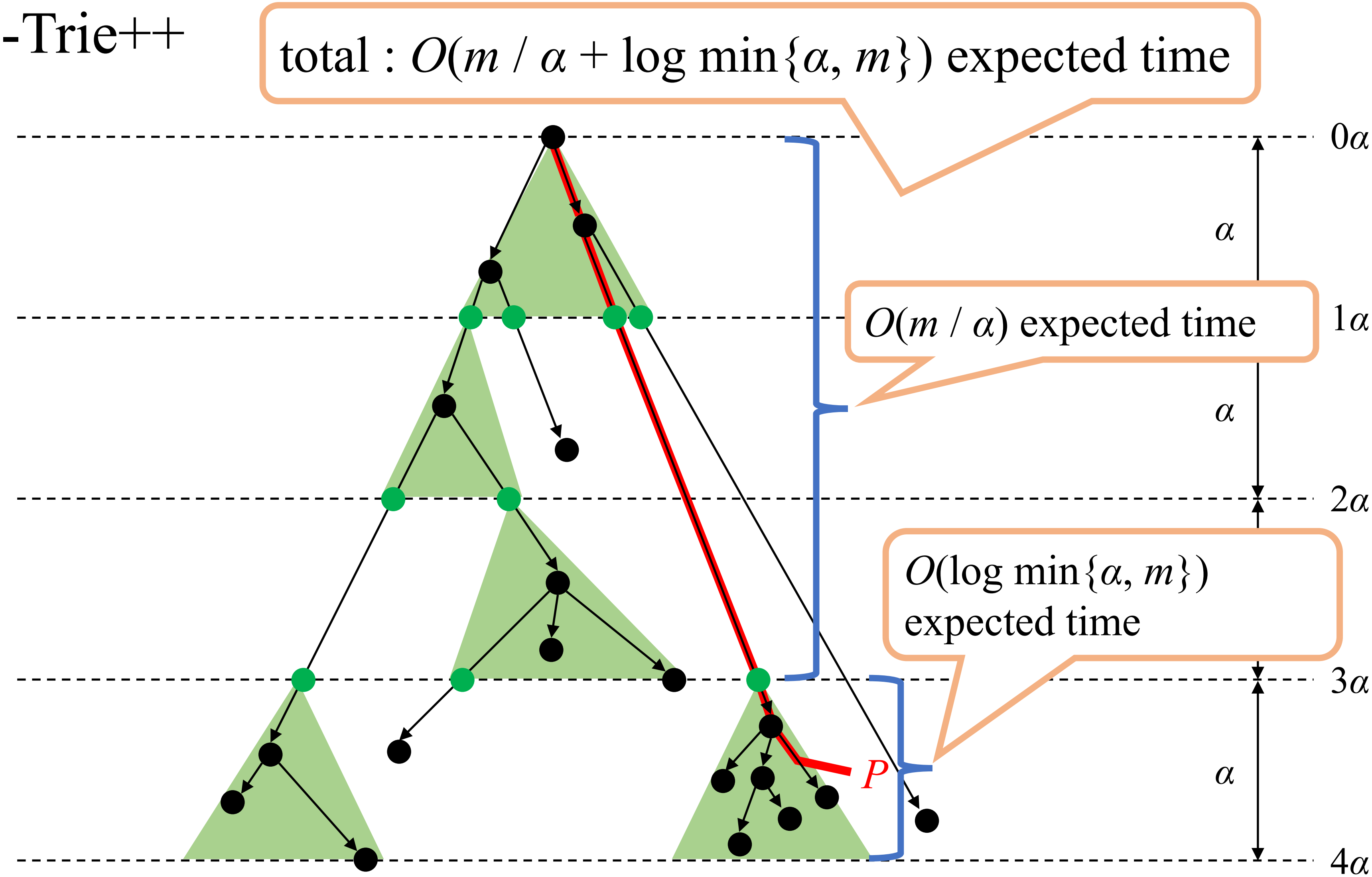
C-Trie++

total : $O(m / \alpha + \log \min\{\alpha, m\} + occ)$ expected time



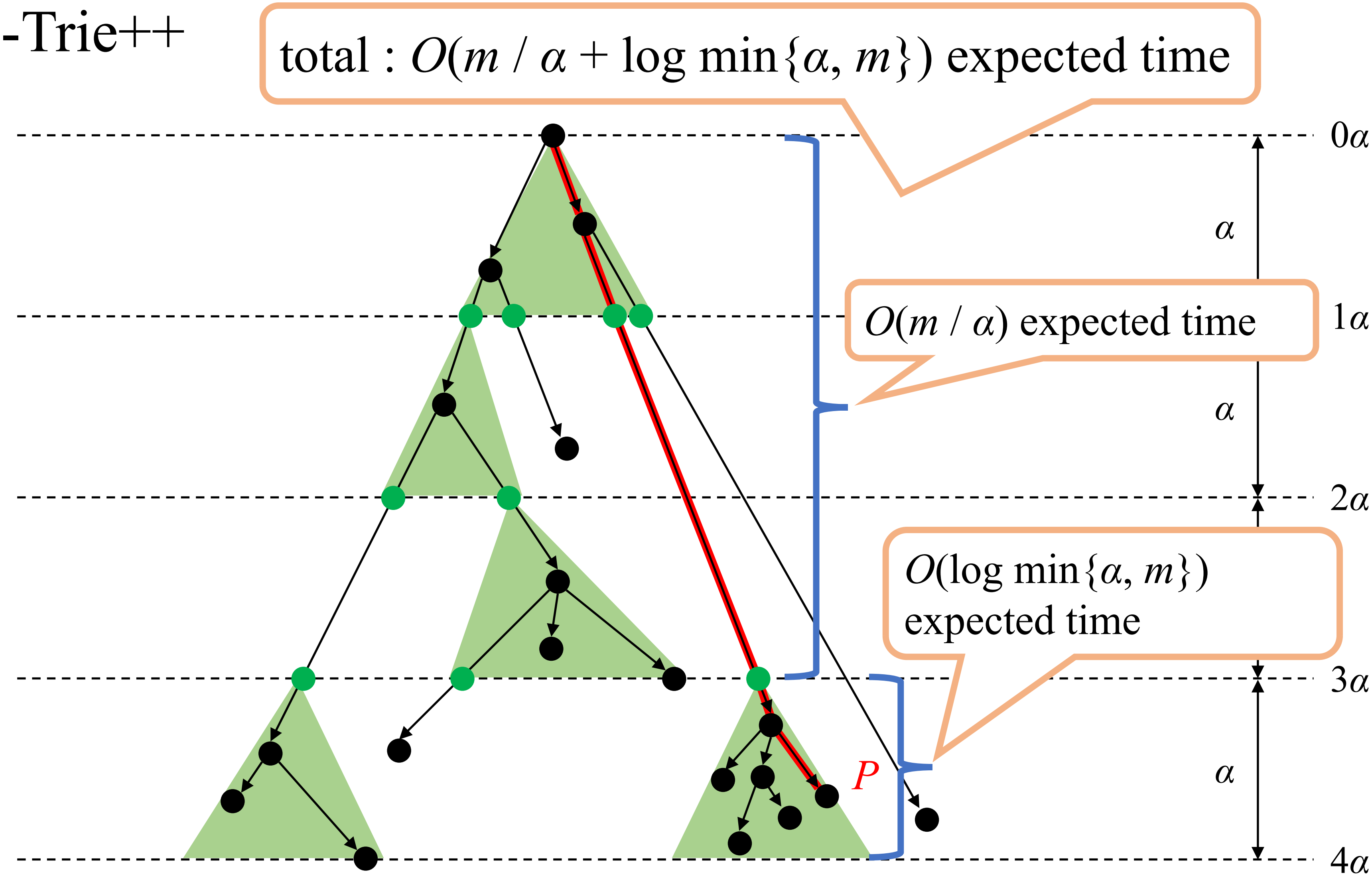
Insertion Time

C-Trie++



Deletion Time

C-Trie++



Experimental Setup

- CPU : Intel Xeon X5560 @2.80 GHz
- Memory : 198GB
- OS : CentOS 6.10
- Language : C++
- Implementations
  - Compact Trie [Takagi et al.]
  - Z-Fast Trie [Ours]
  - Packed C-Trie [Takagi et al.]
  - **C-Trie++** [Ours]

Datasets

- Characteristics

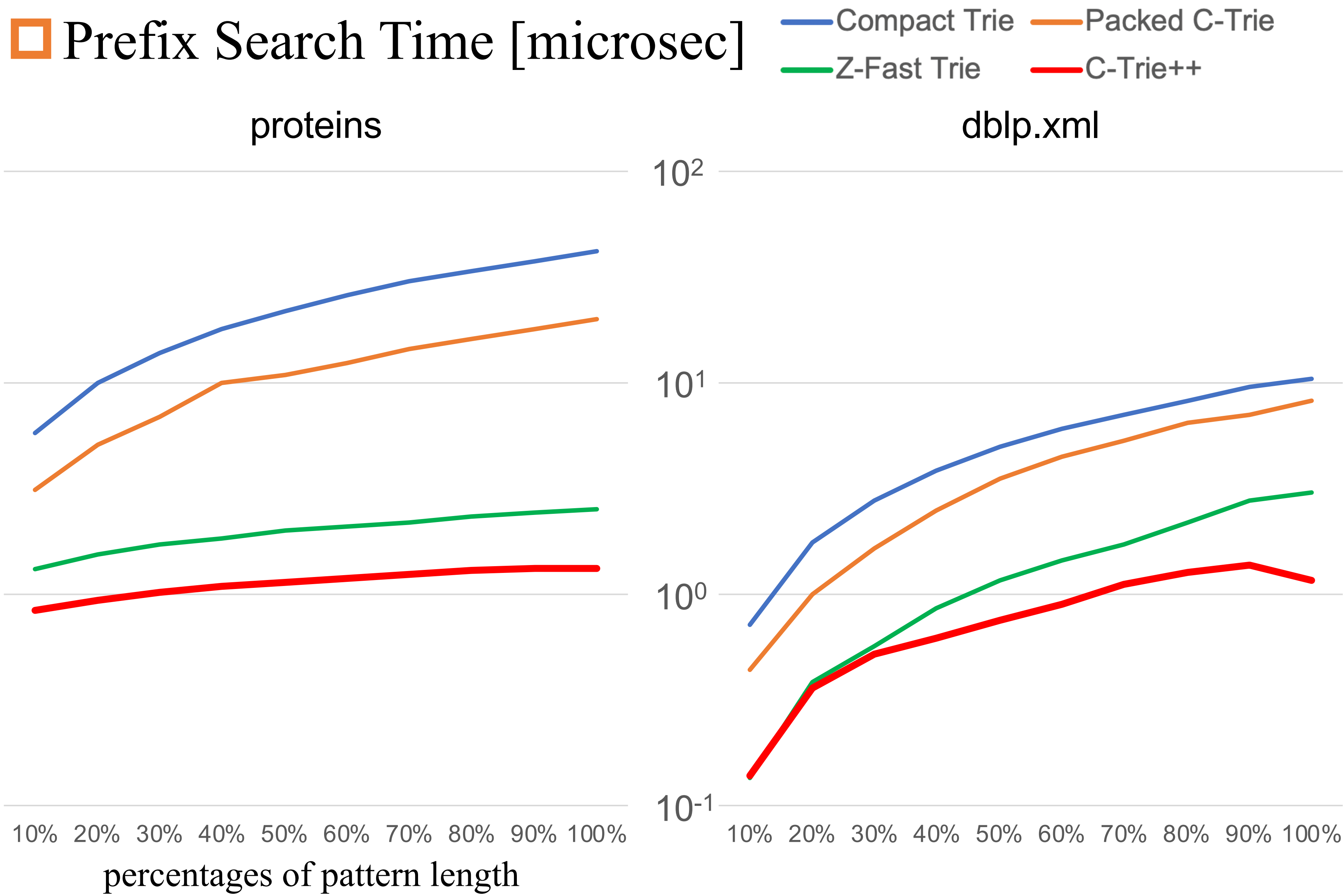
Datasets	size[MB]	σ	$k[10^3]$	average length	avg. LCP	C-Tries nodes$[10^3]$
proteins	864.14	26	2,982	302.8	38.8	5,778
dblp.xml	164.89	96	2,950	57.6	34.4	5,899

- We split a data set into strings at delimiters such as carriage returns, which form our input set S.

- In our experiment for prefix search, we took the prefixes of length 10%, 20%, …, 100% of the strings of S as patterns, and measured the average query time.

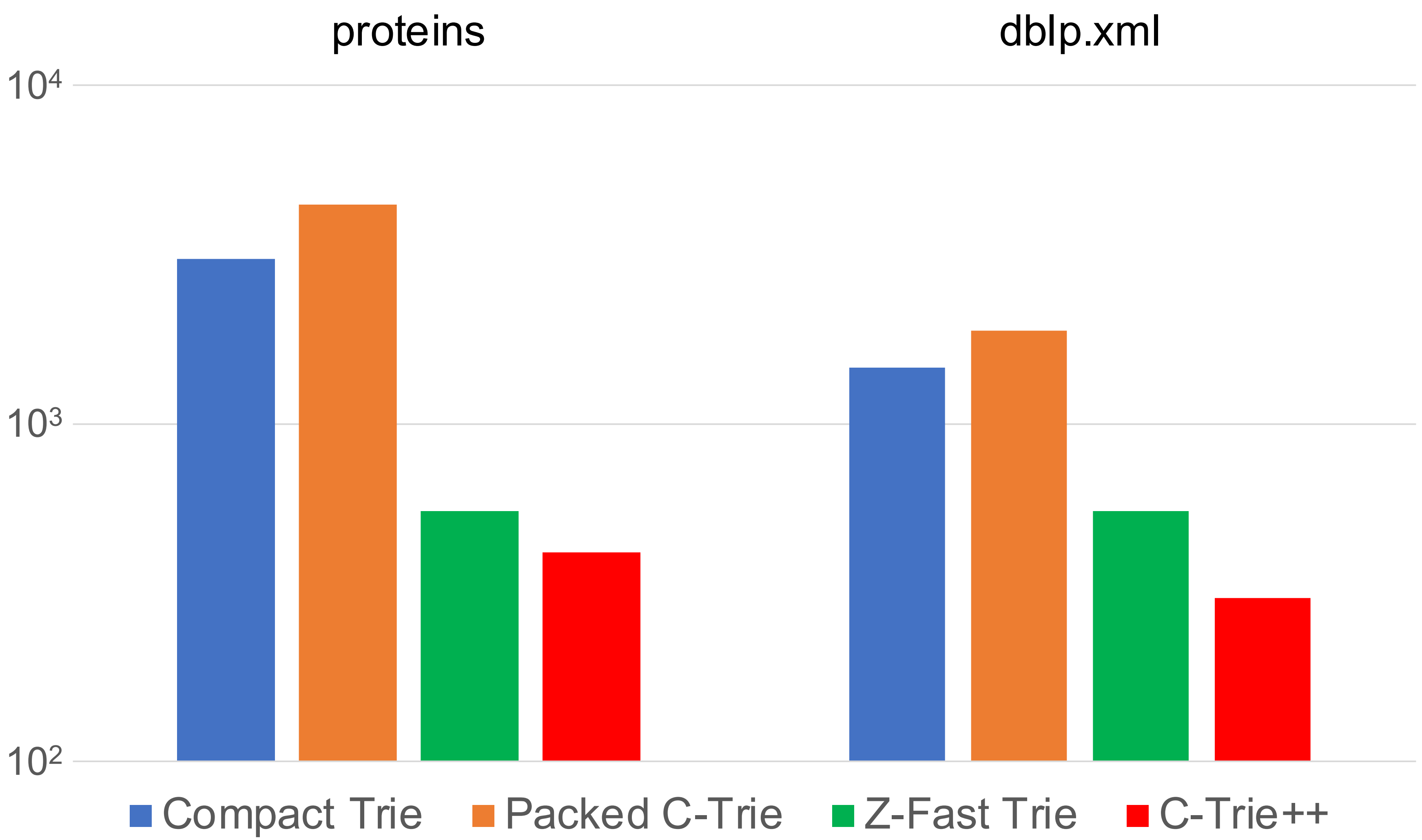
Experimental Results

- Prefix Search Time [microsec]



Experimental Results

- Memory Usage [MB]



Conclusions

- Summary
  - We proposed c-trie++:
    - Space: $|T| \log \sigma + \Theta(kw)$ bits.
    - Prefix Search : $\boldsymbol{O(m / \alpha + \log \min\{\alpha, m\} + occ)}$ **time**.
    - Insert : $\boldsymbol{O(m / \alpha + \log \min\{\alpha, m\})}$ **time**.
    - Delete : $\boldsymbol{O(m / \alpha + \log \min\{\alpha, m\})}$ **time**.
  - Our computational experiments support the claim that c-trie++ is the fastest trie for prefix search.

- Future Work
  - Use SIMD instruction sets that allow larger machine word sizes (here $w = 64$ bits).