String Indexing for Patterns with Wildcards

Philip Bille[1], Inge Li Gørtz[1], Hjalte Wedel Vildhøj[1], and Søren Vind[1]

[1]Technical University of Denmark, DTU Informatics

SWAT 2012, Helsinki
July 6, 2012

DTU

String Indexing for Patterns with Wildcards

Problem Definition

Build an index for a string $t \in \Sigma^*$, that, given a query pattern p, quickly can report where p occurs in t.

$$p = p_0 \ * \ p_1 \ * \ \ldots \ * \ p_j$$

Example

$$t = \texttt{combinatorialpatternmatching}$$

$$p = * \texttt{ at } * * * \texttt{ n}$$

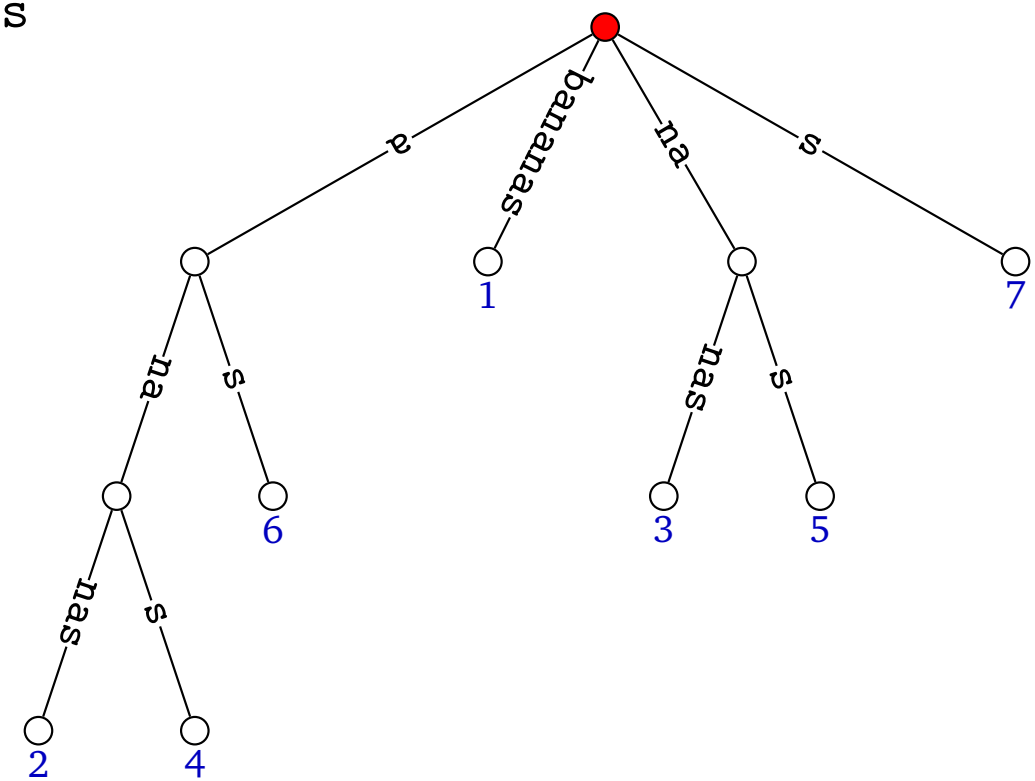
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
c	o	m	b	i	n	a	t	o	r	i	a	l	p	a	t	t	e	r	n	m	a	t	c	h	i	n	g
													*	a	t	*	*	*	n								
																				*	a	t	*	*	*	n	

Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7}{\texttt{bananas}}$

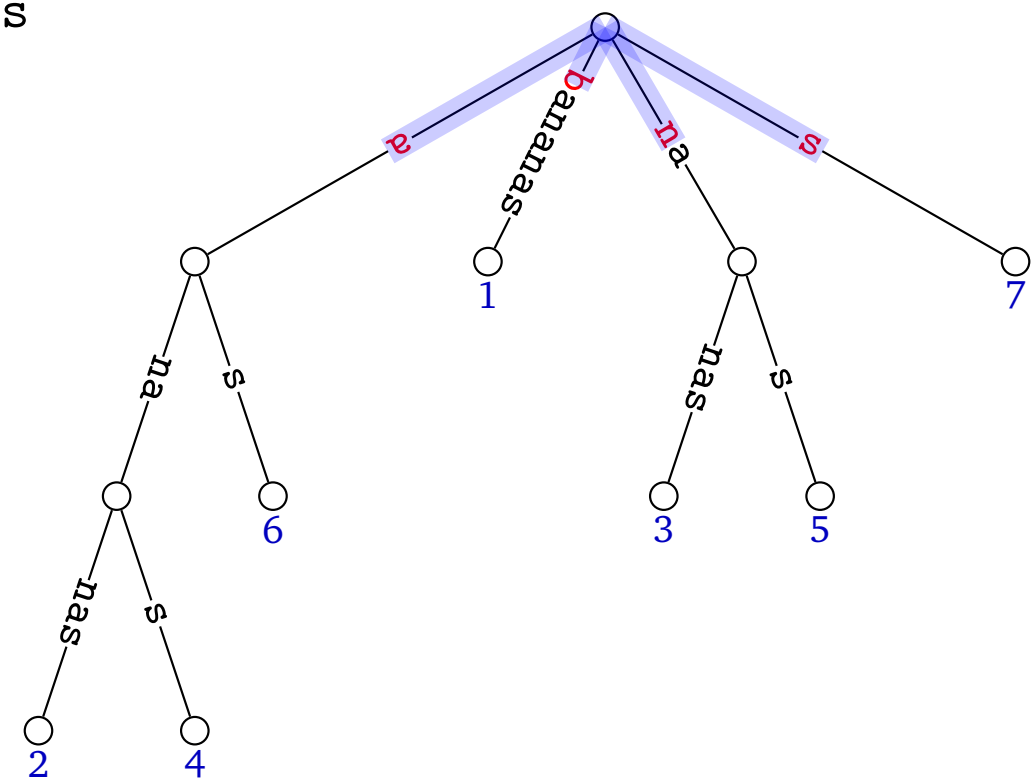


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7}{\texttt{bananas}}$

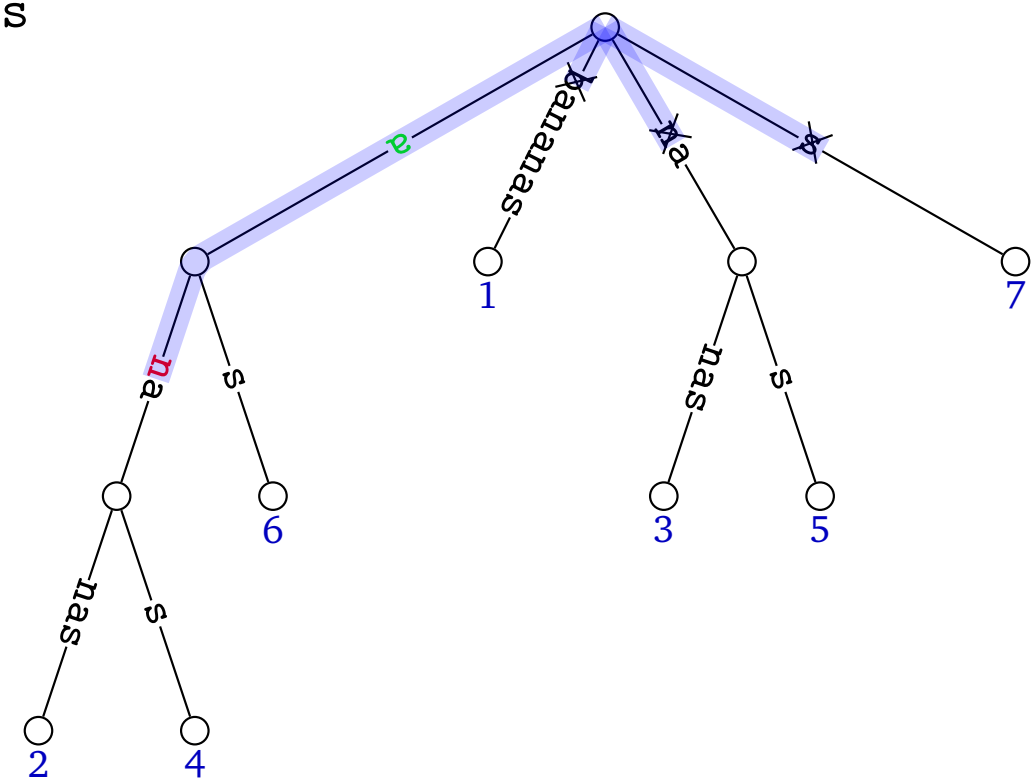


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7}{\texttt{bananas}}$

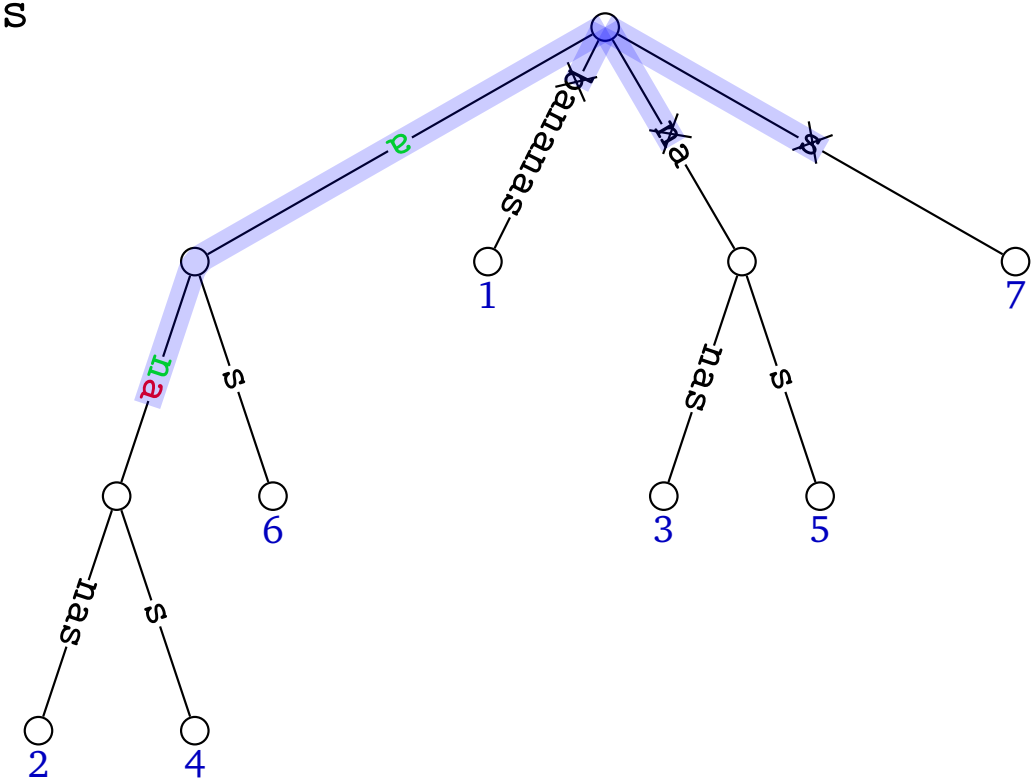


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7}{\texttt{bananas}}$

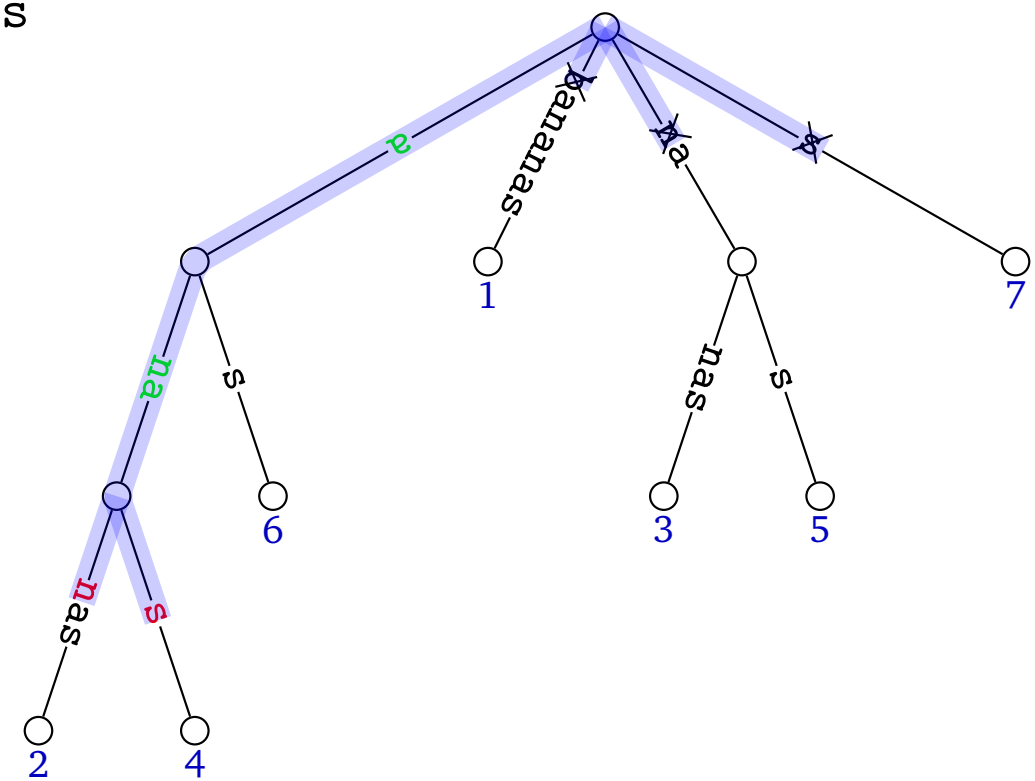


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t = \overset{1\ 2\ 3\ 4\ 5\ 6\ 7}{\texttt{bananas}}$

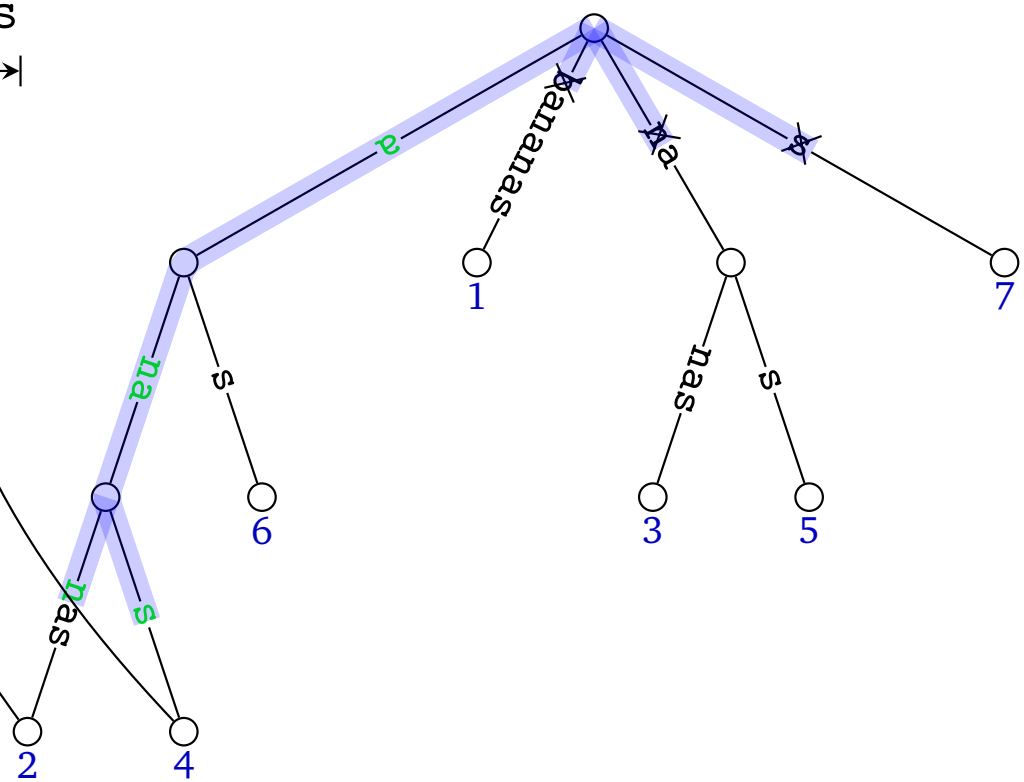


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t =$ 1 2 3 4 5 6 7
bananas

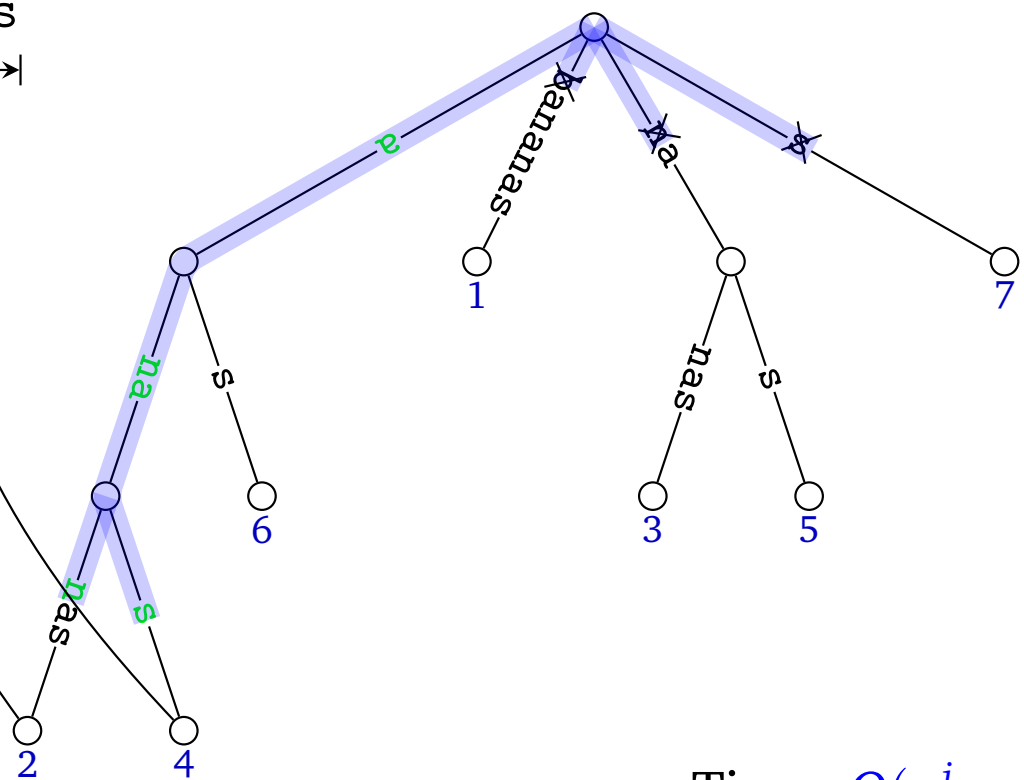


Two Simple Solutions

Suffix Tree Search

$p = *\texttt{na}*$

$t =$ 1 2 3 4 5 6 7
bananas

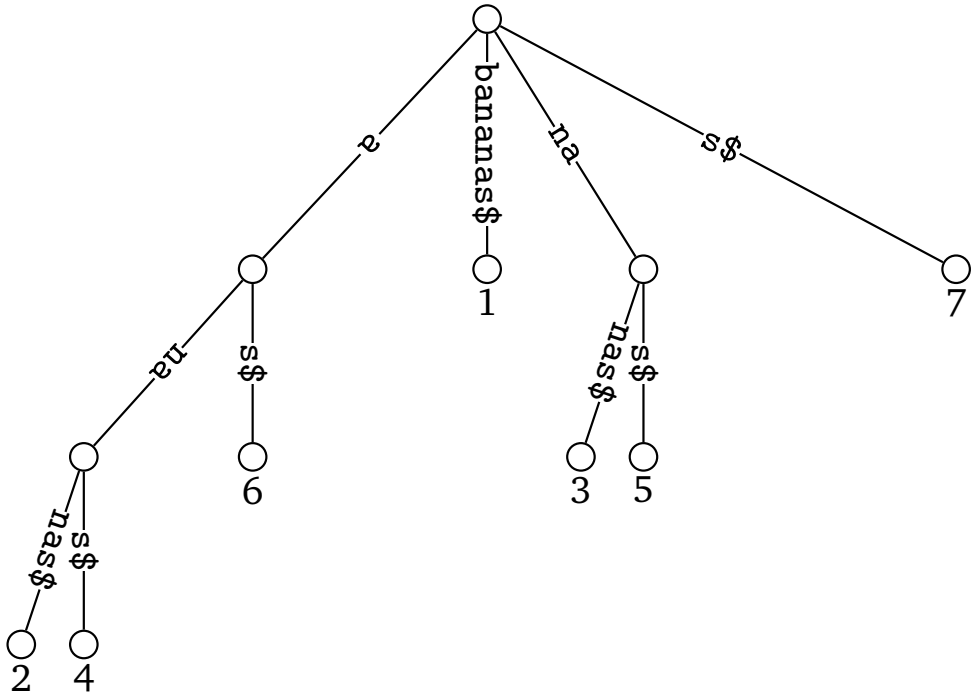


Time: $O(\sigma^j m + occ)$

Space: $O(n)$

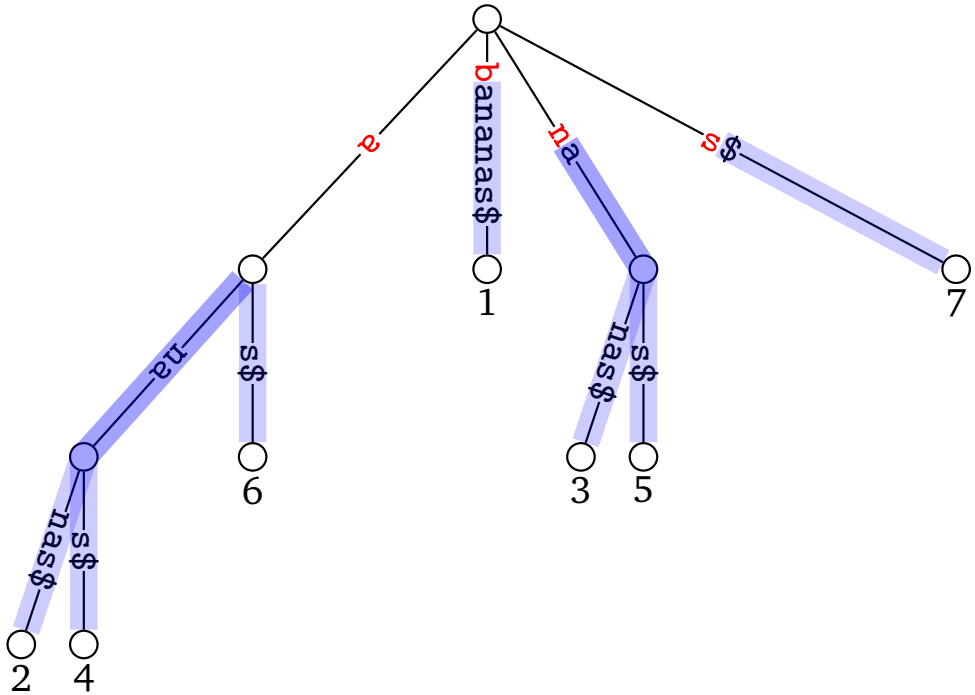
Two Simple Solutions

Simple Linear Time Index



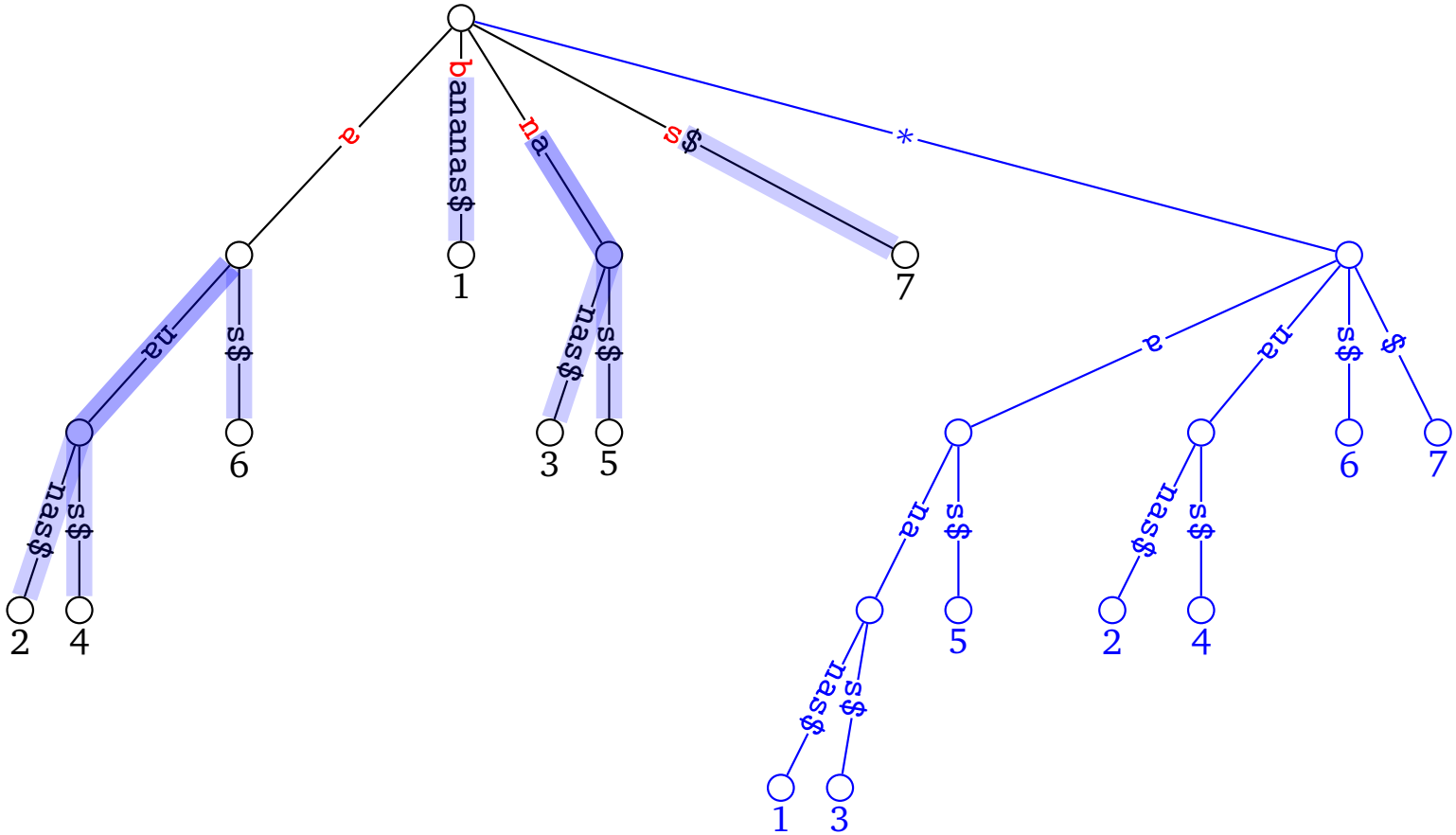
Two Simple Solutions

Simple Linear Time Index



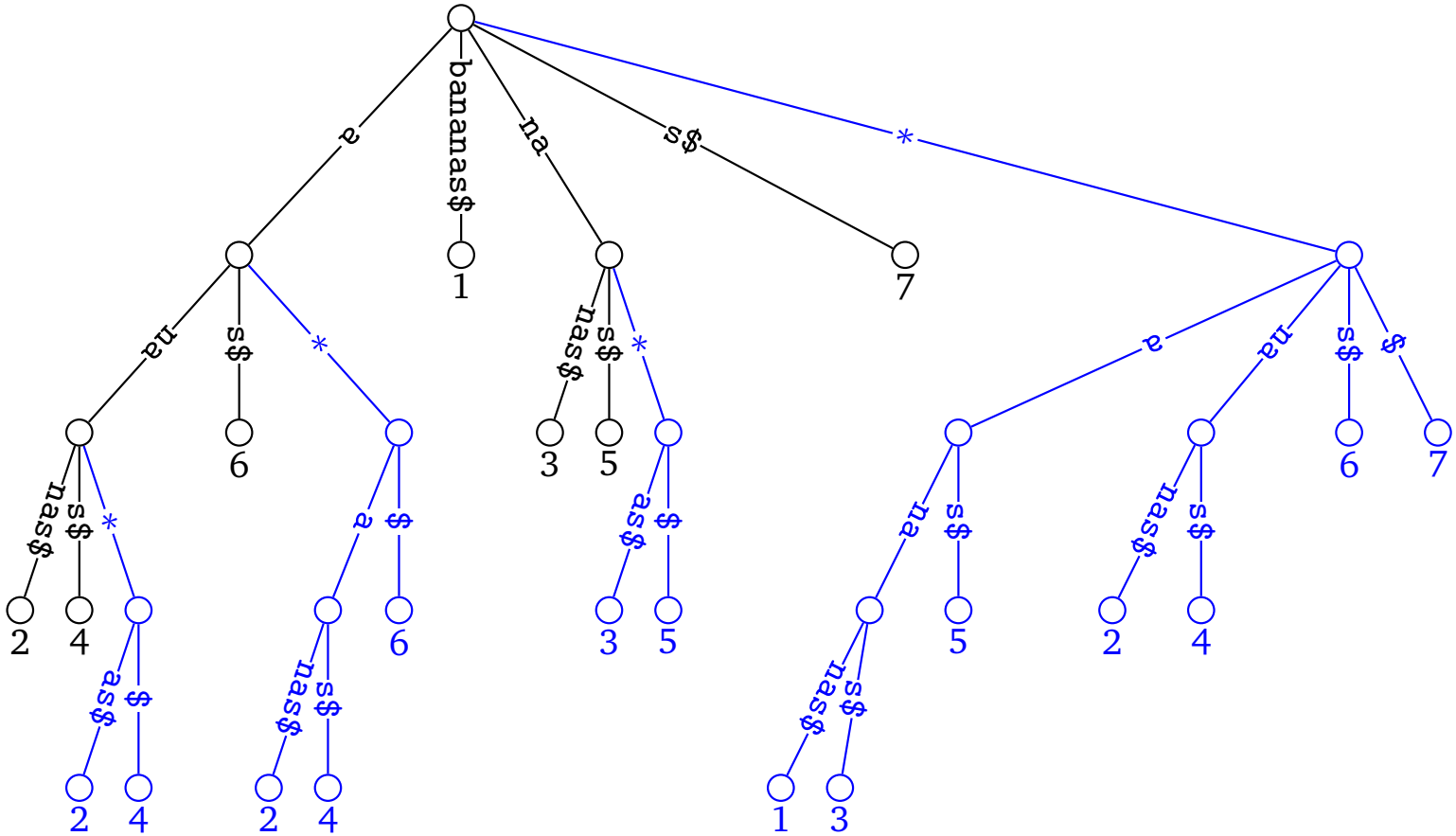
Two Simple Solutions

Simple Linear Time Index



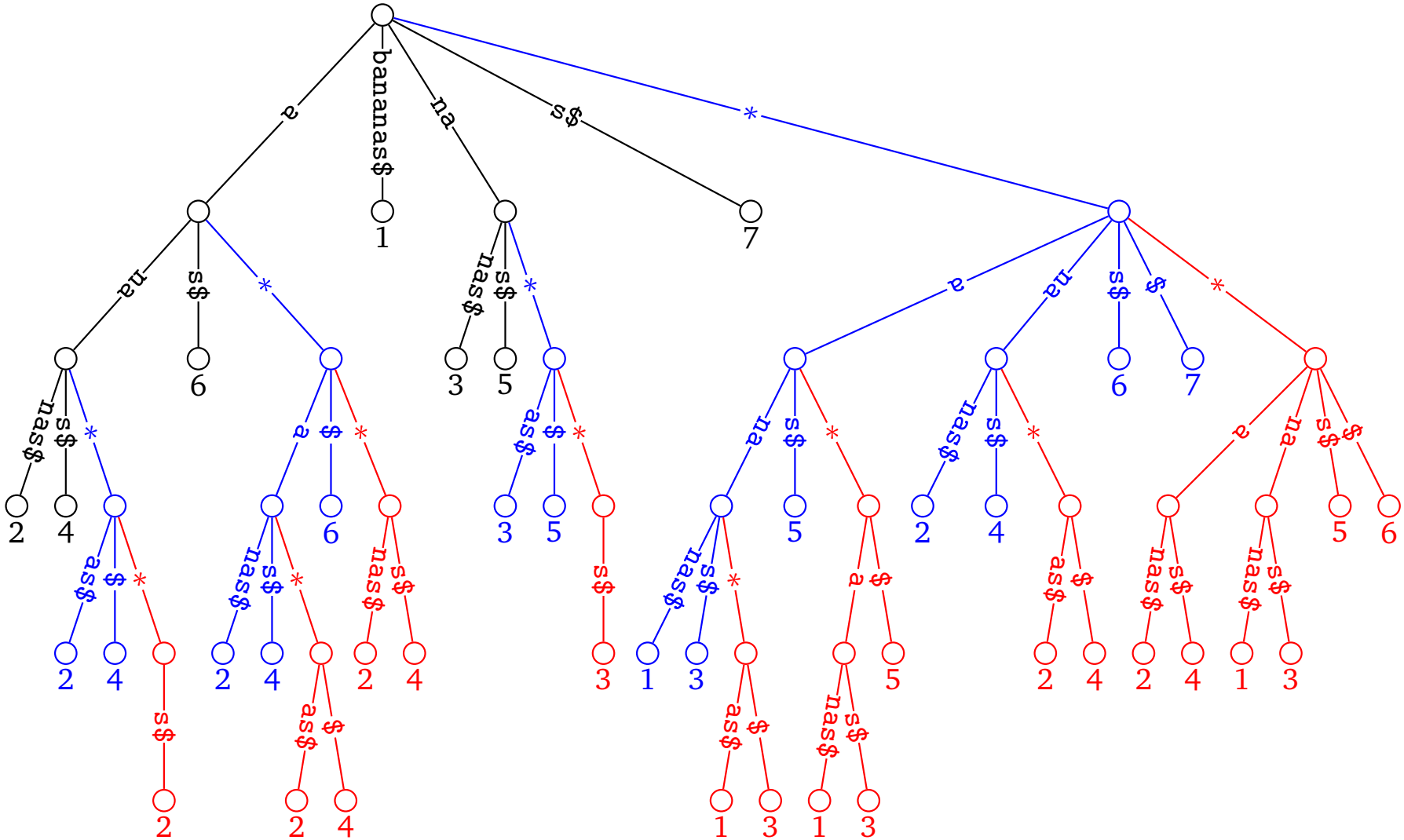
Two Simple Solutions

Simple Linear Time Index



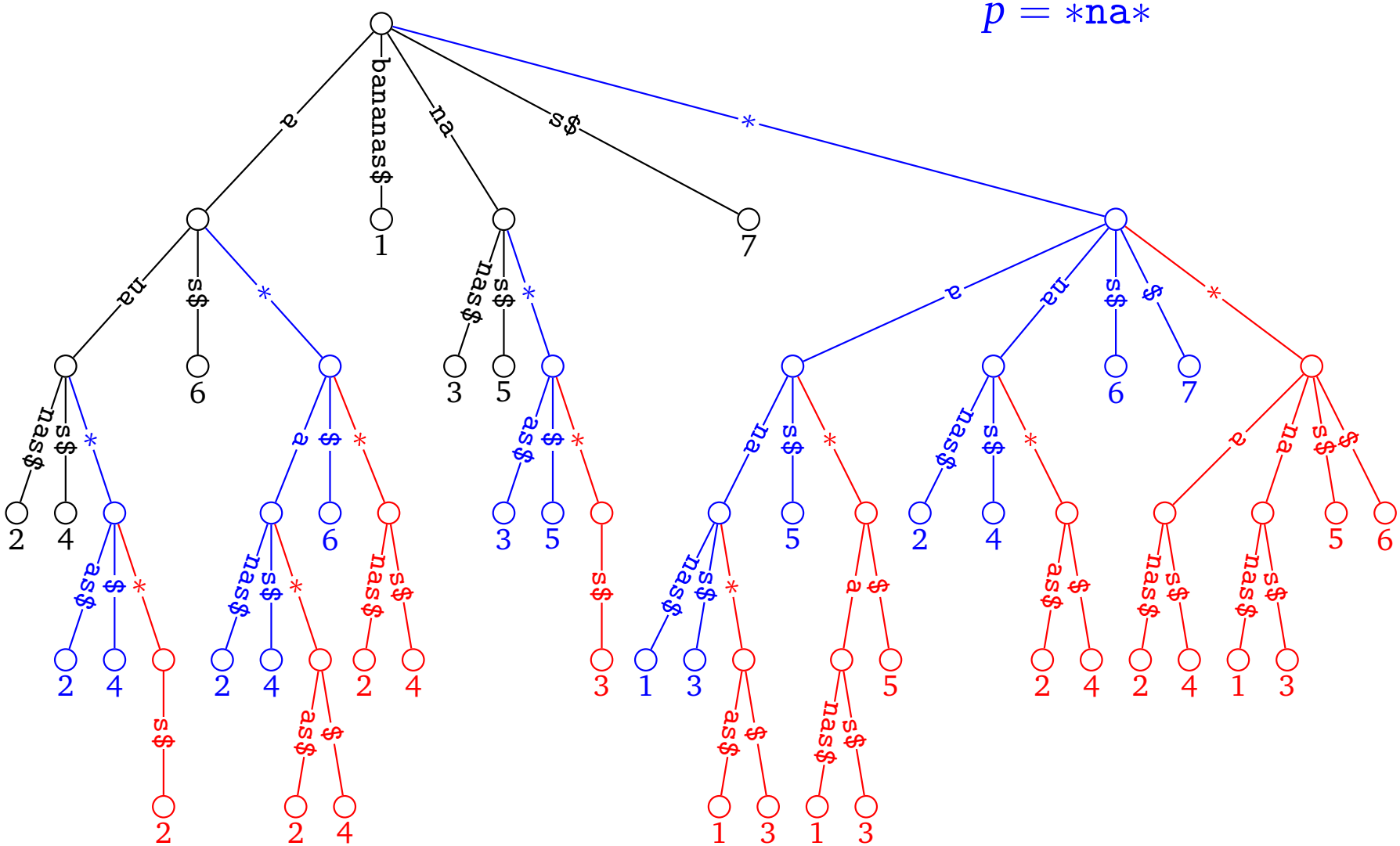
Two Simple Solutions

Simple Linear Time Index



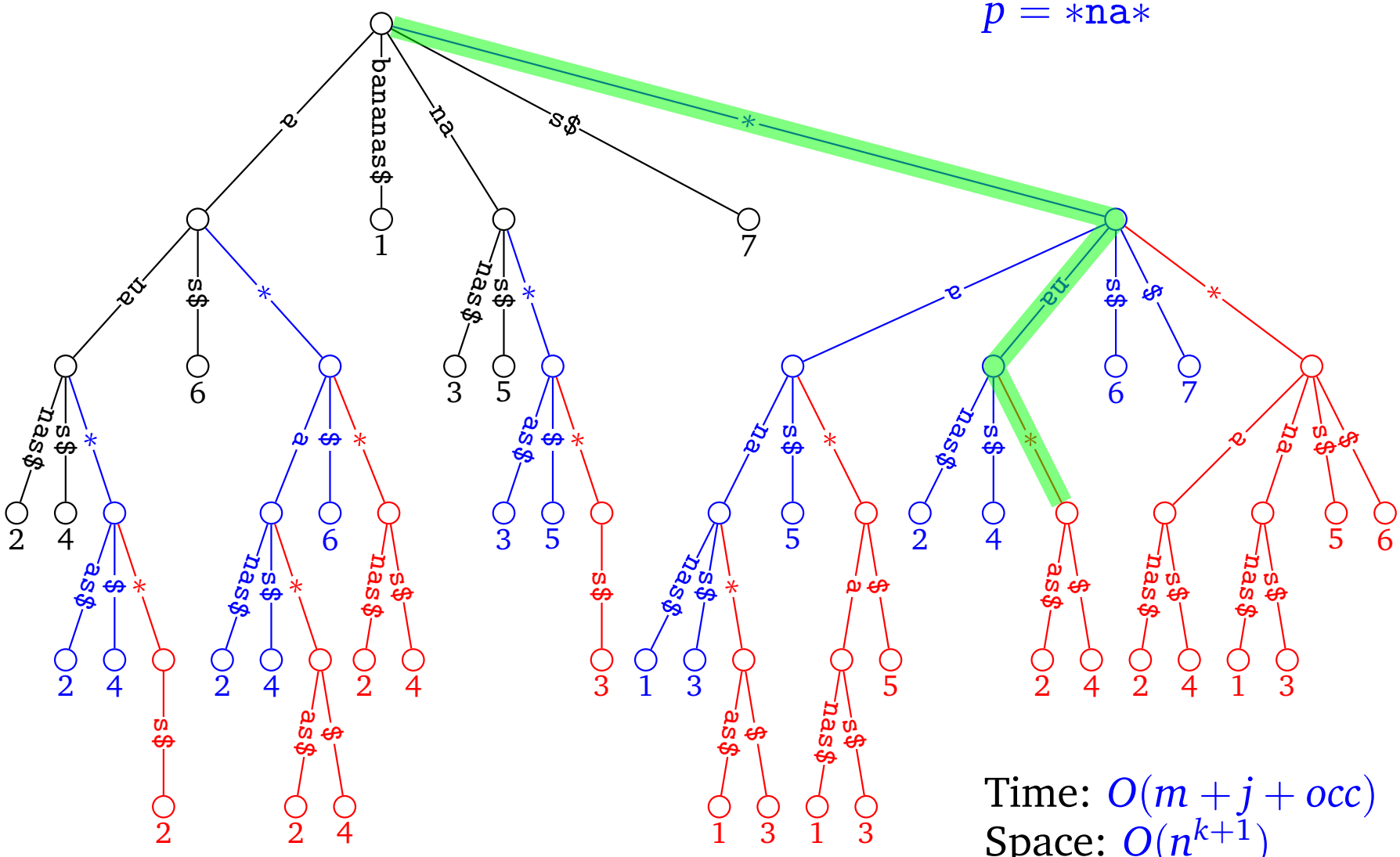
Two Simple Solutions

Simple Linear Time Index



Two Simple Solutions

Simple Linear Time Index



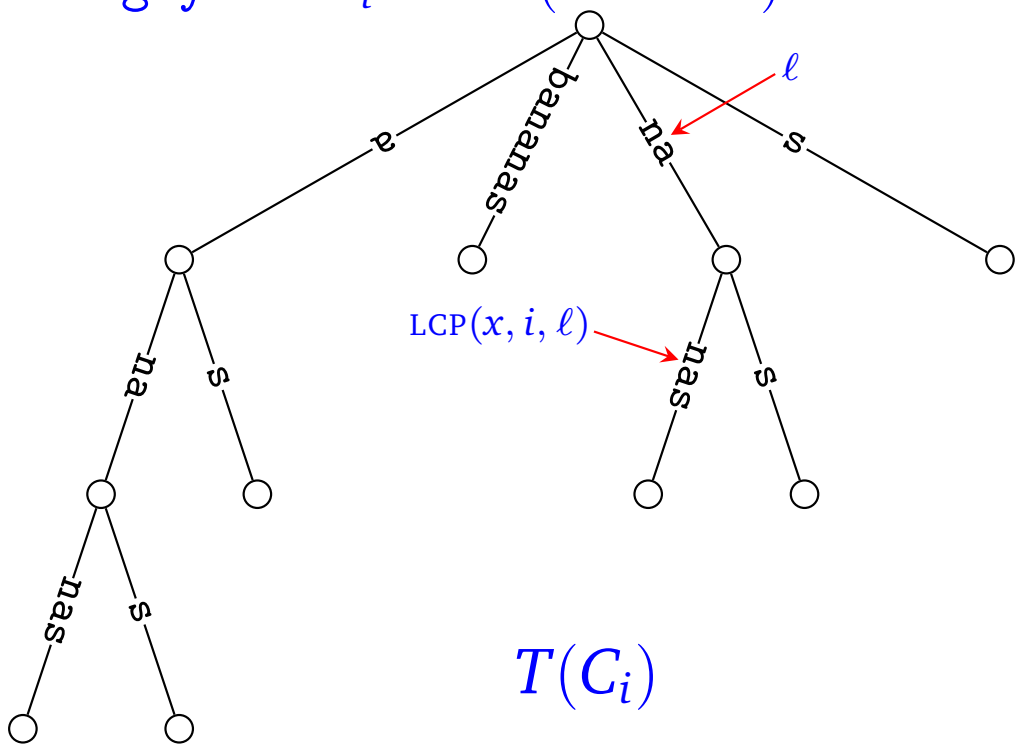
The Longest Common Prefix Data Structure [1]

LCP Queries

Let C_i be a set of substrings of the indexed string. Consider the following query on the compressed trie $T(C_i)$ storing the strings in C_i.

> $\text{LCP}(x, i, \ell)$: *The location where the search for $x \in \Sigma^*$ stops when starting in location $\ell \in T(C_i)$.*

Example: $x = \texttt{angry}$ and $C_i = \text{suff}(\texttt{bananas})$.



$T(C_i)$

[1] R. Cole, L. Gottlieb, and M. Lewenstein.
Dictionary matching and indexing with errors and don't cares. Proc. 36th STOC, 2004.

The Longest Common Prefix Data Structure [1]

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

[1] R. Cole, L. Gottlieb, and M. Lewenstein.
Dictionary matching and indexing with errors and don't cares. Proc. 36th STOC, 2004.

The Longest Common Prefix Data Structure [1]

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

How fast can you answer an LCP query?

- $O(\log \log n)$ time and $O(n \log n)$ space.

  $\Rightarrow$ Index with query time $O(m + \sigma^j \log \log n + occ)$ and space $O(n \log n)$.

- We show that you can also do $O(\log n)$ time and $O(n)$ space.

  $\Rightarrow$ Index with query time $O(m + \sigma^j \log n + occ)$ and space $O(n)$.

[1] R. Cole, L. Gottlieb, and M. Lewenstein.
Dictionary matching and indexing with errors and don't cares. Proc. 36th STOC, 2004.

The Longest Common Prefix Data Structure [1]

An Application

Search for subpatterns in the suffix tree using the LCP data structure:

- Build the LCP data structure for the suffix tree.
- Search with a query pattern containing wildcards:
  - Search for complete subpatterns using LCP queries.
  - Branch on a wildcard as in the simple suffix tree solution.

How fast can you answer an LCP query?

- $O(\log \log n)$ time and $O(n \log n)$ space.

  $\Rightarrow$ Index with query time $O(m + \sigma^j \log \log n + occ)$ and space $O(n \log n)$.

- We show that you can also do $O(\log n)$ time and $O(n)$ space.

  $\Rightarrow$ Index with query time $O(m + \sigma^j \log n + occ)$ and space $O(n)$.

[1] R. Cole, L. Gottlieb, and M. Lewenstein.
Dictionary matching and indexing with errors and don't cares. Proc. 36th STOC, 2004.

SOLUTION 1

An Unbounded Wildcard Index Using Linear Space

Query Time:	$O(m + \sigma^j \log\log n + occ)$
Space Usage:	$O(n)$

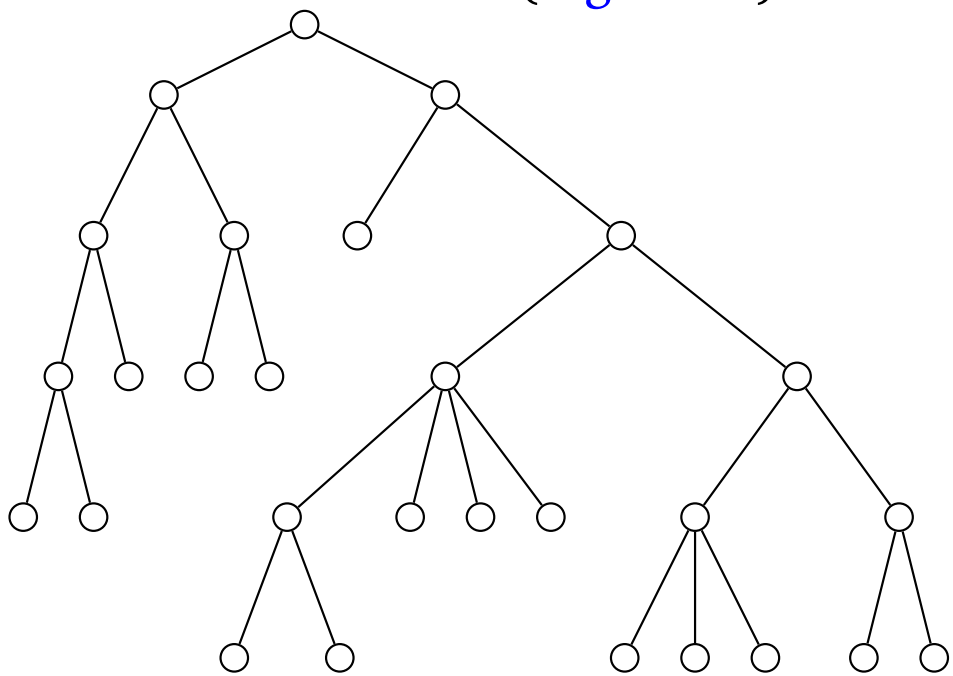
An Unbounded Wildcard Index Using Linear Space

ART Decomposition [2]

Definition:

- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

Example: A tree with $n = 16$ leaves ($\log n = 4$).



[2] S. Alstrup, T. Husfeldt, and T. Rauhe
Marked ancestor problems. Proc. 39th FOCS, 1998.

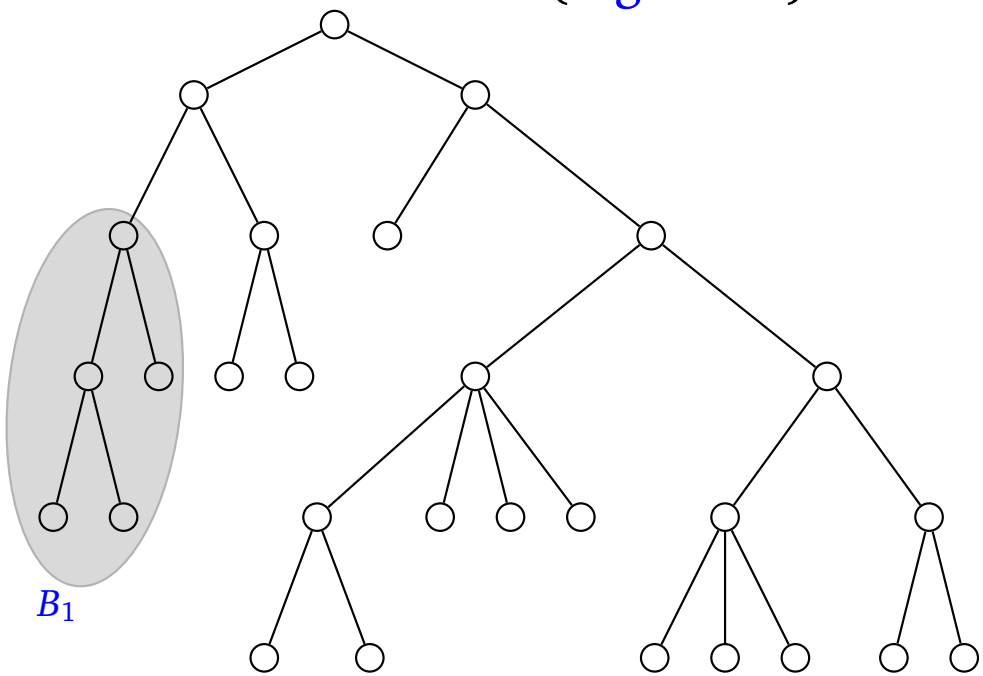
An Unbounded Wildcard Index Using Linear Space

ART Decomposition [2]

Definition:

- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

Example: A tree with $n = 16$ leaves ($\log n = 4$).



[2] S. Alstrup, T. Husfeldt, and T. Rauhe
Marked ancestor problems. Proc. 39th FOCS, 1998.

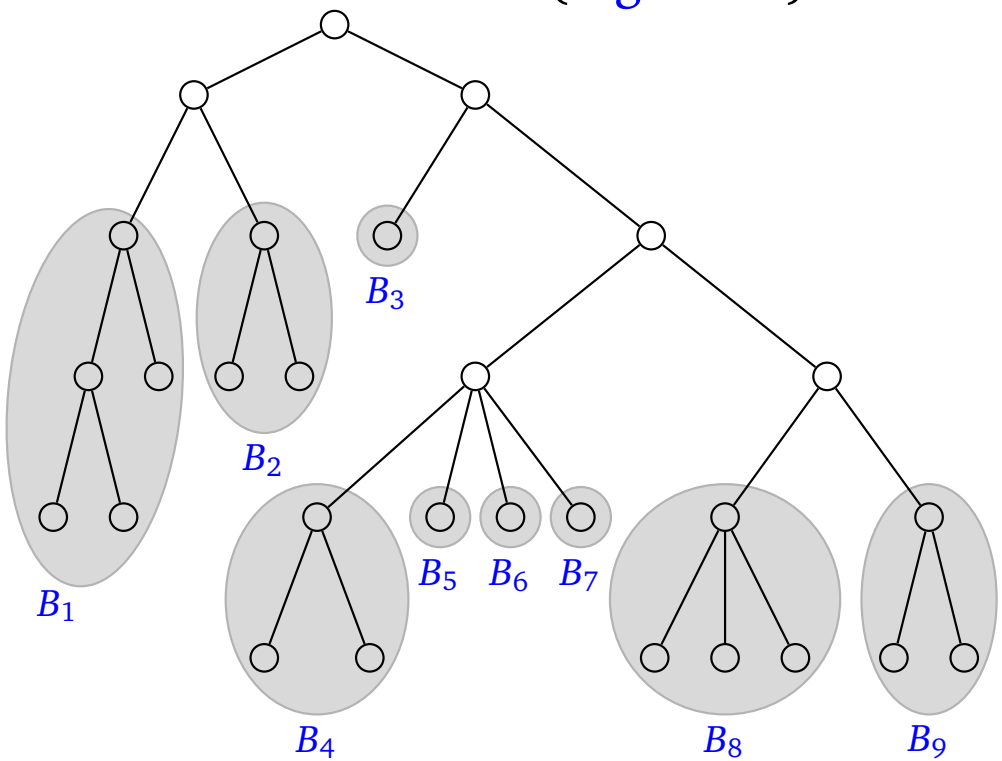
An Unbounded Wildcard Index Using Linear Space

ART Decomposition [2]

Definition:

- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

Example: A tree with $n = 16$ leaves ($\log n = 4$).



[2] S. Alstrup, T. Husfeldt, and T. Rauhe
Marked ancestor problems. Proc. 39th FOCS, 1998.

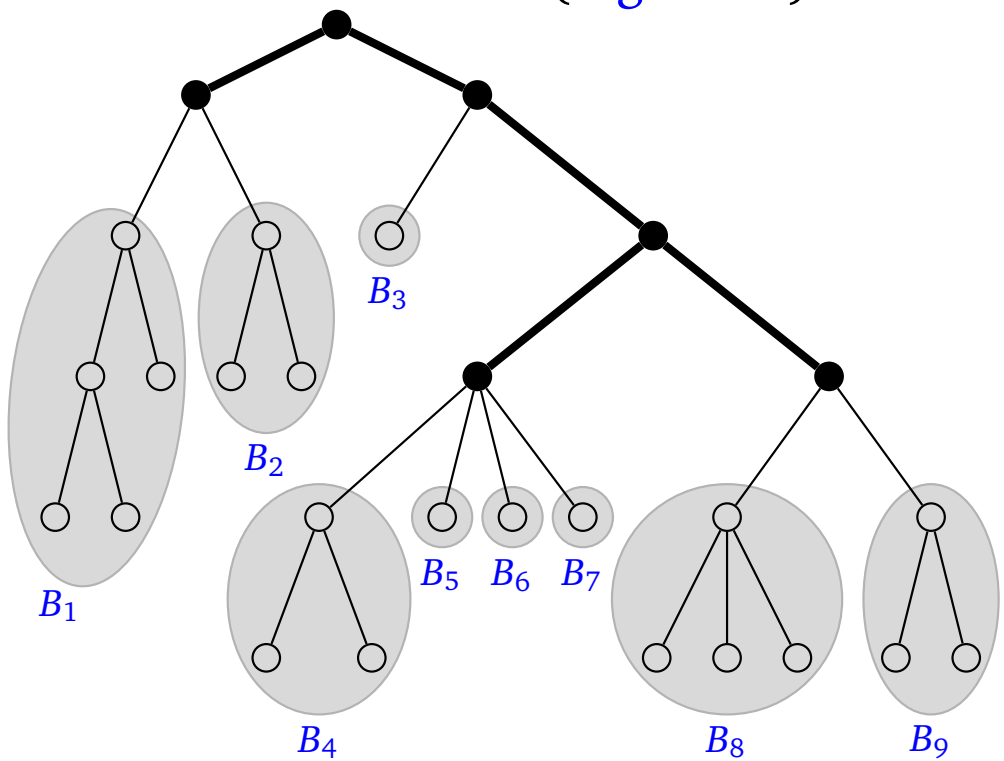
An Unbounded Wildcard Index Using Linear Space

ART Decomposition [2]

Definition:

- A *bottom tree* is a maximal subtree with at most $\log n$ leaves.
- Vertices not in a bottom tree constitute the *top tree*.

Example: A tree with $n = 16$ leaves ($\log n = 4$).



Property: The top tree has $O(\frac{n}{\log n})$ leaves.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe
Marked ancestor problems. Proc. 39th FOCS, 1998.

An Unbounded Wildcard Index Using Linear Space

Obtaining the Index

- Use the ART decomposition to decompose the suffix tree into a number of logarithmic sized bottom trees and a single top tree containing $O(\frac{n}{\log n})$ leaves.
- Store the top and bottom trees in LCP data structure.
- On the top tree T': Add support for $O(\log \log n)$ time LCP queries using the method by Cole et al. [3]
  - This requires space $O(|T'| \log |T'|) = O(\frac{n}{\log n} \log(\frac{n}{\log n})) = O(n)$.
- On the bottom trees $T(C_1), \ldots, T(C_q)$: Add support for $O(\log n)$ time LCP queries using our new method.
  - This requires $O(\sum_{i=1}^{q} |C_i|) = O(n)$ space.
  - The query time becomes $O(\log |C_i|) = O(\log \log n)$.

This gives an unbounded wildcard index using $O(n)$ space with query time $O(m + \sigma^j \log \log n + occ)$.

[3] R. Cole, L. Gottlieb, and M. Lewenstein.
Dictionary matching and indexing with errors and don't cares. Proc. 36th STOC, 2004.

SOLUTION 2

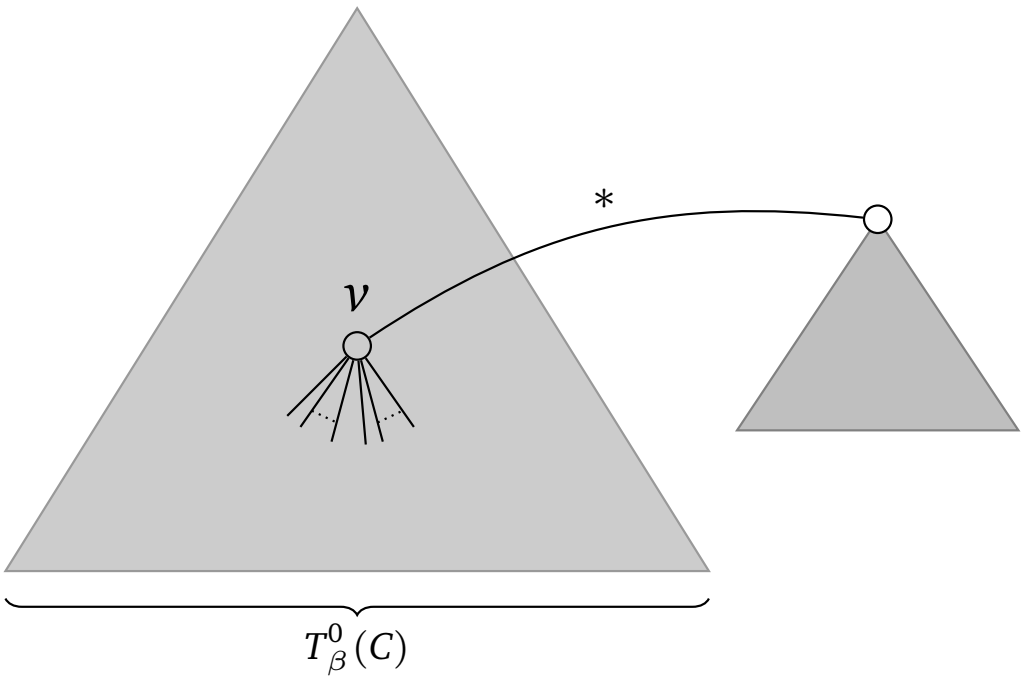
A Time-Space Trade-Off for k-Bounded Wildcard Indexes

Query Time:	$O(m + \beta^j \log\log n + occ)$
Space Usage:	$O(n \log_\beta^{k-1}(n) \log n)$

A Time-Space Trade-Off for Bounded Wildcard Indexes

General Idea

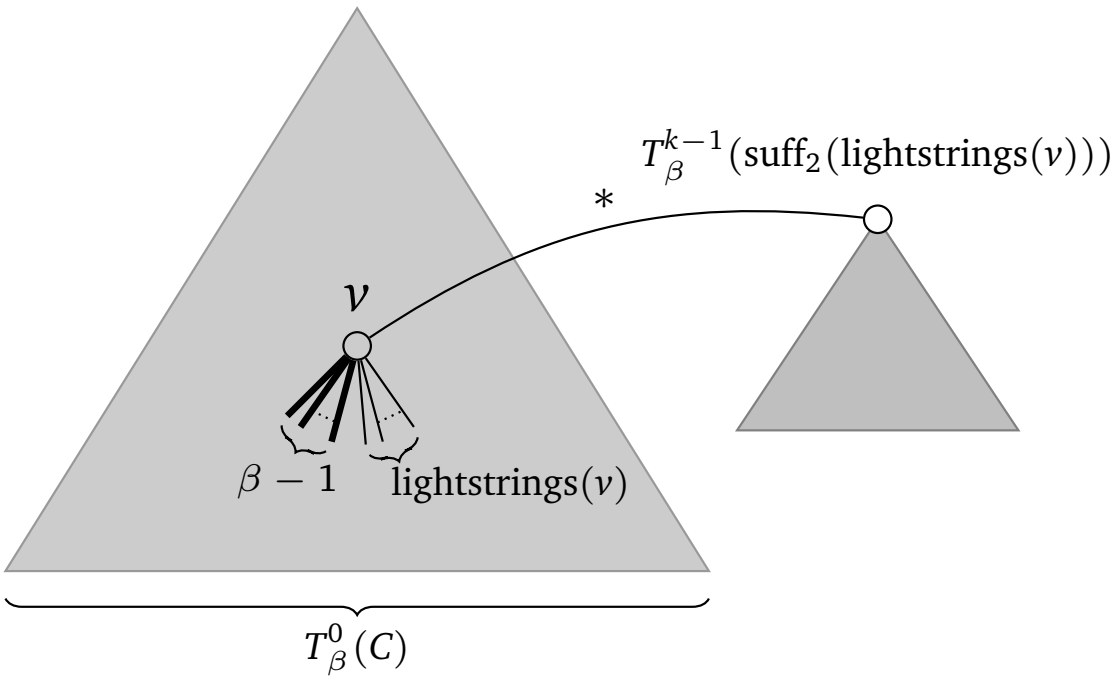
Reduce the branching factor of the suffix tree search from σ to β by creating wildcard trees. Query time: $O(m + \beta^j \log\log n + occ)$ when using the LCP data structure.



A Time-Space Trade-Off for Bounded Wildcard Indexes

General Idea

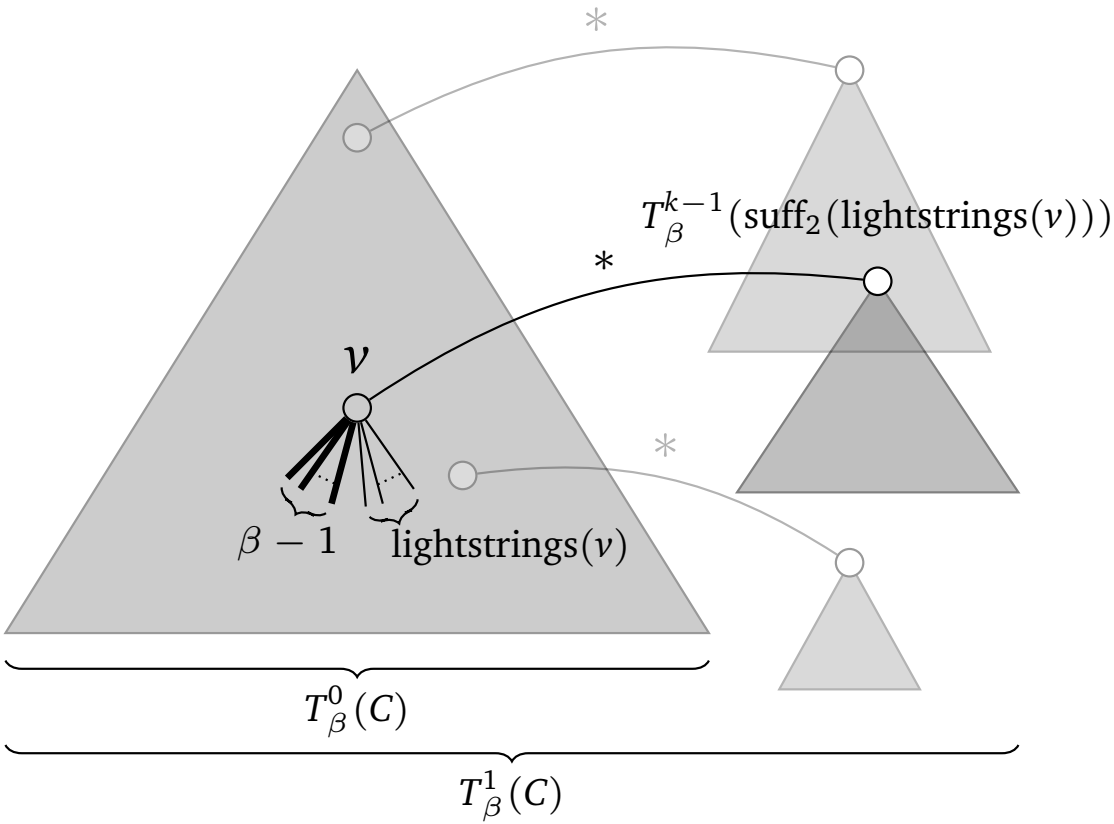
Reduce the branching factor of the suffix tree search from σ to β by creating wildcard trees. Query time: $O(m + \beta^j \log\log n + occ)$ when using the LCP data structure.



A Time-Space Trade-Off for Bounded Wildcard Indexes

General Idea

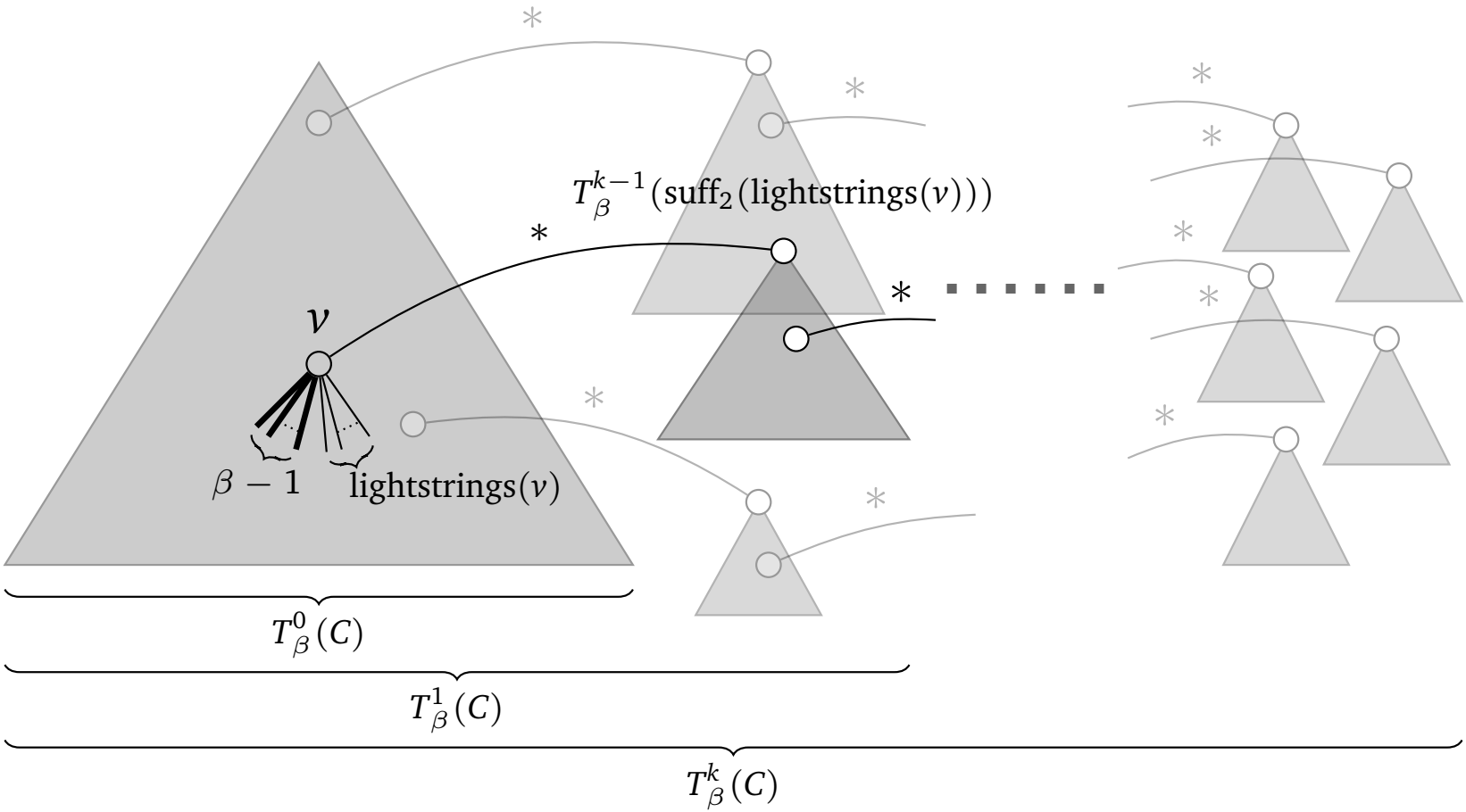
Reduce the branching factor of the suffix tree search from σ to β by creating wildcard trees. Query time: $O(m + \beta^j \log\log n + occ)$ when using the LCP data structure.



A Time-Space Trade-Off for Bounded Wildcard Indexes

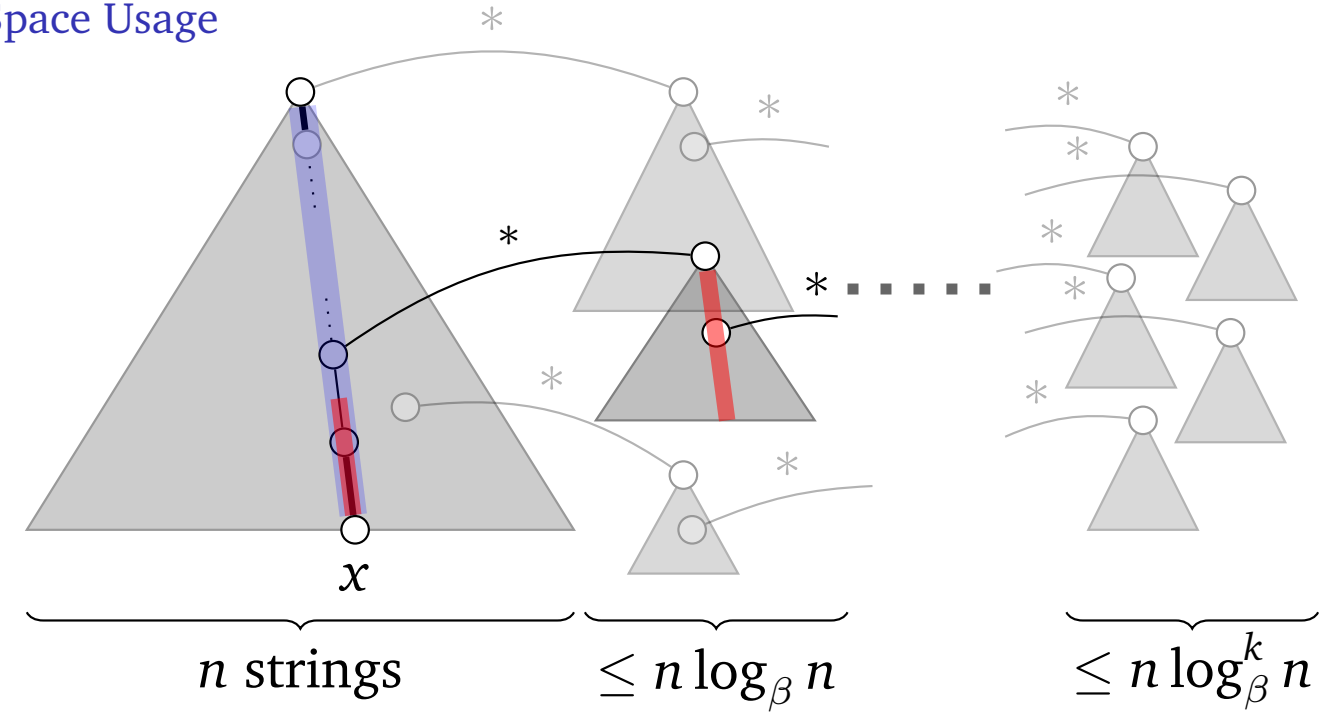
General Idea

Reduce the branching factor of the suffix tree search from σ to β by creating wildcard trees. Query time: $O(m + \beta^j \log\log n + occ)$ when using the LCP data structure.



A Time-Space Trade-Off for Bounded Wildcard Indexes

Analysing the Space Usage



Each string in $T(C)$ gives rise to at most $\text{lightdepth}(x) \leq \log_\beta n$ strings on the next level. So the number of strings in a k-level index is at most

$$\sum_{i=0}^{k} n \log_\beta^i n = O(n \log_\beta^k n) \ .$$

By using the LCP data structure to support LCP queries on every subtrie, we obtain a k-bounded wildcard index with query time $O(m + \beta^j \log\log n + occ)$ using space $O(n \log_\beta^{k-1}(n) \log n)$.

SOLUTION 3

A k-Bounded Wildcard Index with Linear Query Time

Query Time:	$O(m + j + occ)$
Space Usage:	$O(n\sigma^{k^2} \log^k \log n)$

A k-Bounded Wildcard Index with Linear Query Time

General Idea

Consider the previously described unbounded wildcard index $\mathcal{A}$ with

- linear space usage, and
- query time $O(m + \sigma^j \log\log n + occ)$.

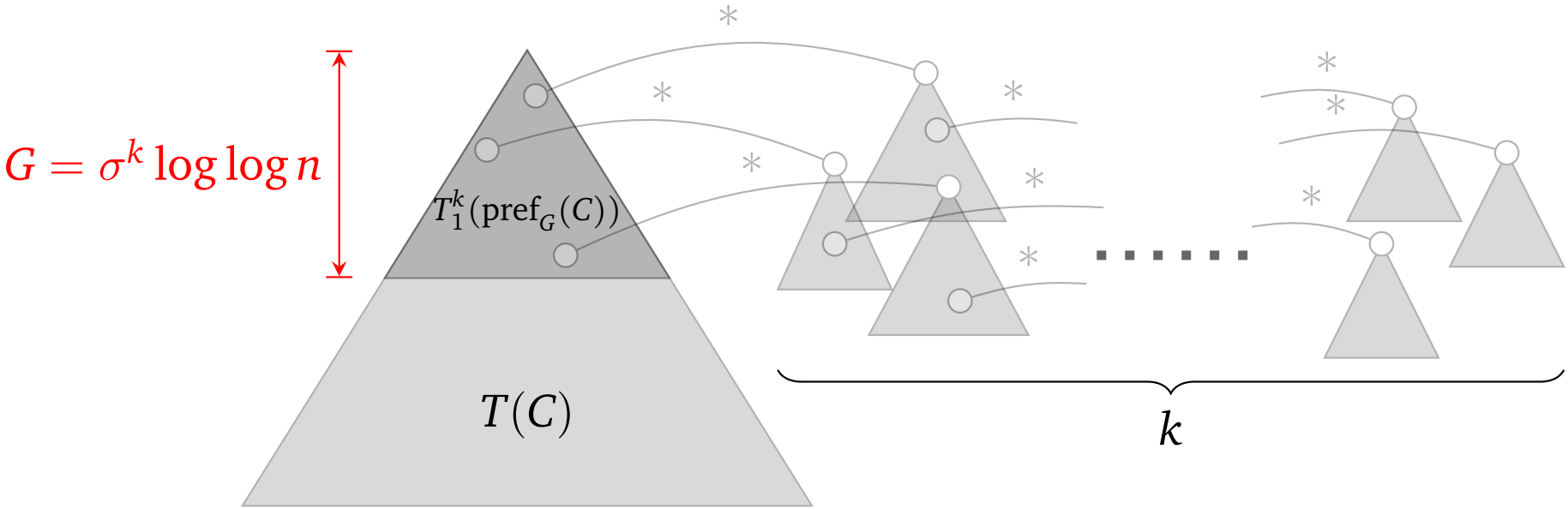
Suppose the pattern is restricted to contain a maximum of k wildcards.

- If $m + j > \sigma^k \log\log n > \sigma^j \log\log n$, (i.e., the query pattern is long) the query time becomes linear: $O(m + j + occ)$.
- If $m + j \leq \sigma^k \log\log n$, we query a special wildcard index $\mathcal{B}$ for short patterns with query time $O(m + j + occ)$.

In any case the query time is $O(m + j + occ)$. The space used by the index is $O(|\mathcal{A}| + |\mathcal{B}|)$.

A k-Bounded Wildcard Index with Linear Query Time

A Special Index for Patterns Shorter than $\sigma^k \log\log n$



$T_1^0(\text{pref}_G(C))$ contains at most n strings. Consider a string x in one of the subtries. At most $|x| \leq G$ suffixes of x appear in tries on the next level. Consequently, the number of strings in $T_1^k(\text{pref}_G(C))$ is bounded by

$$\sum_{i=0}^{k} nG^i = O(n(\sigma^k \log\log n)^k) = O(n\sigma^{k^2} \log^k \log n) \ .$$

Result: A k-bounded wildcard index with linear query time $O(m + j + occ)$ using space $O(n\sigma^{k^2} \log^k \log n)$.

Conclusions

- Three new solutions for string indexing for patterns with wildcards:
  - The fastest linear space index.
  - A trade-off for k-bounded wildcard indexes.
  - The first non-trivial linear time index.
- All solutions generalize to string indexing for patterns with variable length gaps.

Conclusions

- Three new solutions for string indexing for patterns with wildcards:
  - The fastest linear space index.
  - A trade-off for k-bounded wildcard indexes.
  - The first non-trivial linear time index.
- All solutions generalize to string indexing for patterns with variable length gaps.

Thank you!