Data Structures 1

NTIN066

Jirka Fink

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics
Charles University in Prague

Summer semester 2025/26

Last change on April 29, 2026

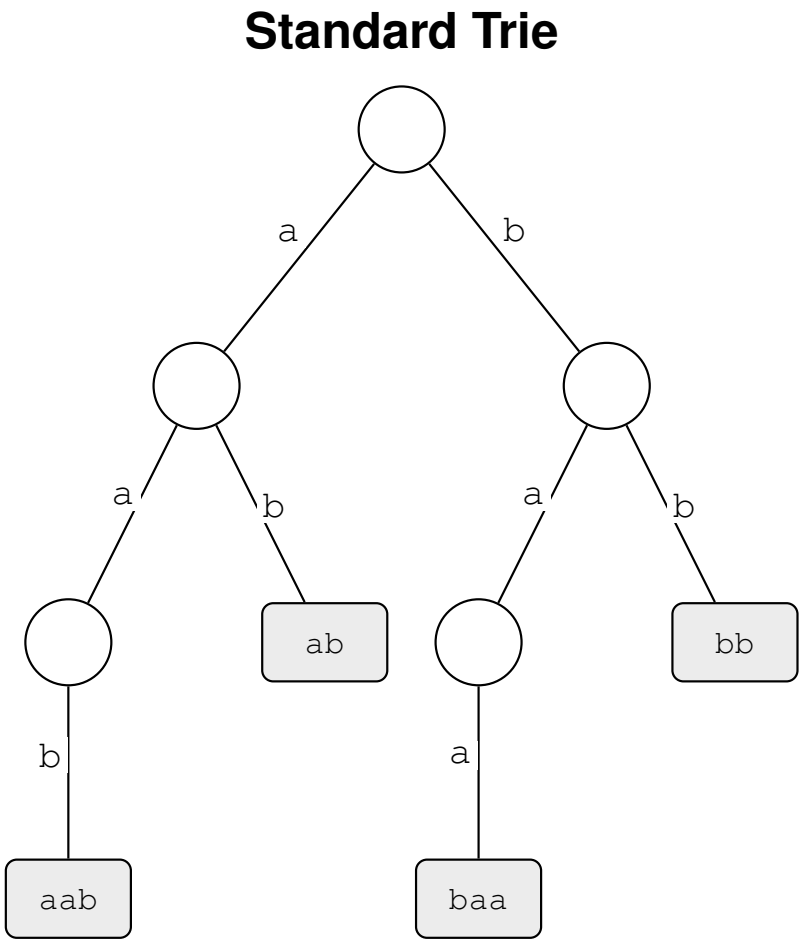
Trie and Compressed Trie

Definition

A *trie* (prefix tree) over an alphabet Σ is a rooted tree in which every edge is labeled with a symbol $a \in \Sigma$. The outgoing edges of any node carry pairwise distinct labels. Each root-to-leaf path spells a word stored in the trie.

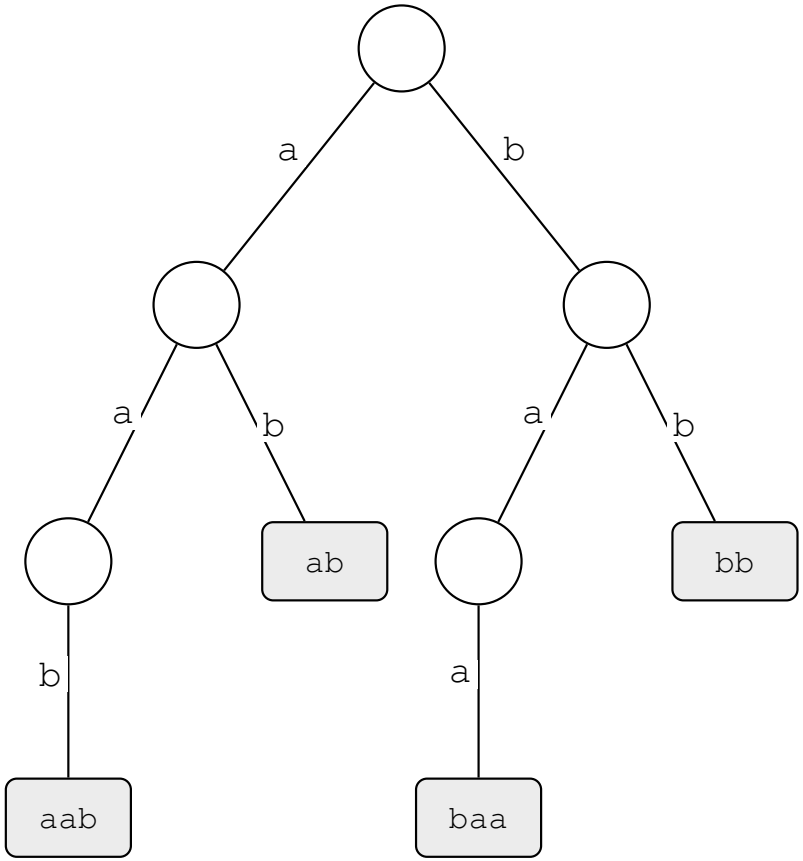
A *compressed trie* is obtained from a trie by collapsing every maximal path of internal nodes of degree 1 (except the root) into a single edge; the new edge is labeled by the concatenation of the original edge labels.

Example: Representing the words {aab, ab, baa, bb}

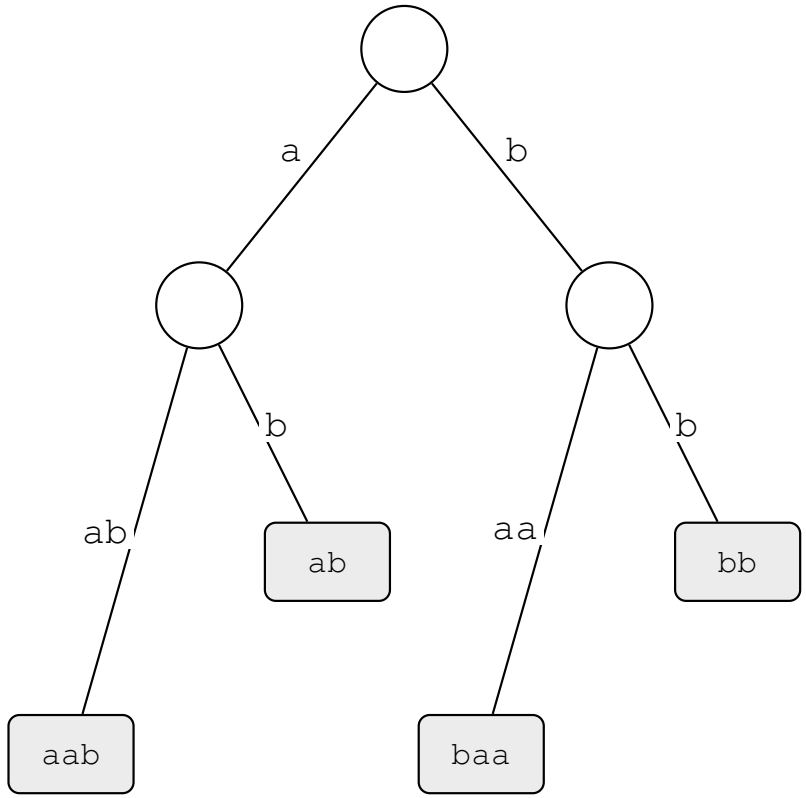


Example: Representing the words {aab, ab, baa, bb}

Standard Trie



Compressed Trie



Alphabet and String Notation

Definition

- Σ is a finite set of symbols (characters, letters).
- Σ^* is the set of finite sequences (words, strings) over Σ.
- ε denotes the special empty string.
- \$ is a special sentinel symbol marking the end of a word.
- $|\alpha|$ denotes the length of a string α.
- $\alpha[k]$ is the k-th character of α, indexed from 0 to $|\alpha| - 1$.
- $\alpha[k:l]$ is the substring starting at position k and ending immediately before position l.
- $\alpha[:l]$ is the prefix consisting of the first l characters.
- $\alpha[k:]$ is the suffix starting at position k.

Suffix Trees

- A *(compressed) suffix tree* is a (compressed) trie that contains all suffixes of a word.
- The children of every node are stored in lexicographic order.
- The *depth* of a node in the compressed trie is defined as the depth of the corresponding node in the uncompressed trie, i.e., the number of characters on the path from the root to that node.

Suffix Trees

- A *(compressed) suffix tree* is a (compressed) trie that contains all suffixes of a word.
- The children of every node are stored in lexicographic order.
- The *depth* of a node in the compressed trie is defined as the depth of the corresponding node in the uncompressed trie, i.e., the number of characters on the path from the root to that node.

Exercise

Create compressed suffix tree for the word ‘banana’.

Rank and Longest Common Prefix Arrays

Definition

- The *suffix array* X stores the lexicographic order of all suffixes of a given word α: $X[i]$ is the starting position of the i-th suffix in lexicographic order.
- The *rank array* R is the inverse permutation of the suffix array, i.e., $X[R[i]] = i$. The value $R[i]$ gives the lexicographic rank of the suffix $\alpha[i:]$.
- $\mathrm{LCP}(\alpha, \beta)$ denotes the length of the *longest common prefix* of strings α and β.
- The *LCP array* L stores the lengths of the longest common prefixes of lexicographically adjacent suffixes:

$$L[i] = \mathrm{LCP}\big(\alpha[X[i]:],\ \alpha[X[i+1]:]\big).$$

- The *lexicographic successor* of the suffix starting at position i begins at position $X\big[R[i]+1\big]$.

Example: Suffix Array for `banana`

i	$X[i]$	$R[i]$	$L[i]$	Suffix
0	6	4	0	ε
1	5	3	1	a
2	3	6	3	ana
3	1	2	0	anana
4	0	5	0	banana
5	4	1	2	na
6	2	0	—	nana

Exercises: Applications

String-Analysis Tasks

- Counting k-grams: Determine the number of different substring of length k of a given string.

Exercises: Applications

String-Analysis Tasks

- Counting k-grams: Determine the number of different substring of length k of a given string.
- Longest common substring of two given strings.

Exercises: Applications

String-Analysis Tasks

- Counting k-grams: Determine the number of different substring of length k of a given string.
- Longest common substring of two given strings.
- Find all occurrences of one string in another one (*needle in a haystack*).

Exercises: Applications

String-Analysis Tasks

- Counting k-grams: Determine the number of different substring of length k of a given string.
- Longest common substring of two given strings.
- Find all occurrences of one string in another one (*needle in a haystack*).
- Longest palindromic substring of a given string.

Finding All Occurrences of a Pattern

To report all occurrences of a pattern (*needle*) in a text (*haystack*), search for the pattern in the suffix tree. The leaves in the resulting subtree correspond to all occurrences. If only the frequency (occurrence count) is required, it suffices to store in each internal node the number of leaves in its subtree.

Applications

Finding All Occurrences of a Pattern

To report all occurrences of a pattern (*needle*) in a text (*haystack*), search for the pattern in the suffix tree. The leaves in the resulting subtree correspond to all occurrences. If only the frequency (occurrence count) is required, it suffices to store in each internal node the number of leaves in its subtree.

Longest Substring with Two or More Occurrences

Locate the deepest internal node in the suffix tree (where depth is the number of characters on the path from the root). The corresponding path label is the longest substring occurring at least twice.

Applications

Finding All Occurrences of a Pattern

To report all occurrences of a pattern (*needle*) in a text (*haystack*), search for the pattern in the suffix tree. The leaves in the resulting subtree correspond to all occurrences. If only the frequency (occurrence count) is required, it suffices to store in each internal node the number of leaves in its subtree.

Longest Substring with Two or More Occurrences

Locate the deepest internal node in the suffix tree (where depth is the number of characters on the path from the root). The corresponding path label is the longest substring occurring at least twice.

Longest Common Substring of Two Words

Given $\alpha, \beta \in \Sigma^*$, build a suffix tree for the concatenated string $\alpha\#\beta\$$ (using distinct delimiters $\#$ and \$). The longest common substring corresponds to the deepest node whose subtree contains at least one leaf representing a suffix originating from α and at least one leaf representing a suffix originating from β.

Exercise: Construct LCP `barokoarokoko`

i	$X[i]$	$R[i]$	$L[i]$	Suffix
0	13	3		ε
1	1	1		`arokoarokoko`
2	6	12		`arokoko`
3	0	10		`barokoarokoko`
4	11	5		`ko`
5	4	8		`koarokoko`
6	9	2		`koko`
7	12	13		`o`
8	5	11		`oarokoko`
9	10	6		`oko`
10	3	9		`okoarokoko`
11	8	4		`okoko`
12	2	7		`rokoarokoko`
13	7	0	—	`rokoko`

Kasai's Algorithm (2001)

Lemma

For every word α *and all* $i = 0, \ldots, |\alpha| - 1$,

$$L\big[R[i+1]\big] \;\geq\; L\big[R[i]\big] - 1.$$

Algorithm constructing LCP array from Suffix array

```
1 l = 0
2 for i = 0, ..., |α| − 1 do
3 |   l = max(0, l − 1)
4 |   j = X[R[i] + 1]
5 |   while i + l < |α| && j + l < |α| && α[i + l] == α[j + l] do
6 |   |   l = l + 1
7 |   L[R[i]] = l
```

The resulting table `barokoarokoko`

i	$X[i]$	$R[i]$	$L[i]$	Suffix
0	13	3	0	ε
1	1	1	5	`arokoarokoko`
2	6	12	0	`arokoko`
3	0	10	0	`barokoarokoko`
4	11	5	2	`ko`
5	4	8	2	`koarokoko`
6	9	2	0	`koko`
7	12	13	1	`o`
8	5	11	1	`oarokoko`
9	10	6	3	`oko`
10	3	9	3	`okoarokoko`
11	8	4	0	`okoko`
12	2	7	4	`rokoarokoko`
13	7	0	—	`rokoko`