Automata

Arnar Bjarni Arnarson

November 28, 2024

School of Computer Science
Reykjavík University

Material

- Sets
- String processing
- Trie
- Sets and languages
- Deterministic Finite Automata

Sets

Sets

- A set is an unordered collection of objects, known as its elements.
- We need a universal set U representing all elements.
- Empty set: $\varnothing = \{\,\}$
- Complement: $\overline{A} = \{x | x \notin A\}$
- Union: $A \cup B = \{x | x \in A \text{ or } x \in B\}$
- Intersection: $A \cap B = \{x | x \in A \text{ and } x \in B\}$
- Difference: $A \setminus B = \{x | x \in A \text{ and } x \notin B\}$
- Symmetric Difference: $A \Delta B = \{x | \text{either } x \in A \text{ or } x \in B\}$

Sets - Examples

- Universe as $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- Let $A = \{1, 2, 3, 6\}$ and $B = \{2, 4, 6, 8\}$.
- Complement: $\overline{A} = \{4, 5, 7, 8\}$
- Union: $A \cup B = \{1, 2, 3, 4, 6, 8\}$
- Intersection: $A \cap B = \{2, 6\}$
- Difference: $A \setminus B = \{1, 3\}$
- Symmetric Difference: $A \Delta B = \{1, 3, 4, 8\}$

String Processing

String processing

- Strings processing is quite common
  - I/O
  - Parsing
  - Identifiers/names
  - Data
- But sometimes strings play the key role
  - We want to find properties of some given strings
  - Is the string a palindrome?
- This can be hard, because the lengths of the strings are often huge

String matching

- Given a string S of length n,
- and a string T of length m,
- find all occurrences of T in S
- Note:
  - Occurrences may overlap
  - Assume strings contain characters from some alphabet Σ

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$
- Three occurrences:

String matching

Example:

- $S = \mathrm{cabcababacaba}$
- $T = \mathrm{aba}$
- Three occurrences:
  - cabc**aba**bacaba

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$
- Three occurrences:
  - cabc**aba**bacaba
  - cabcab**aba**caba

String matching

Example:

- $S = \text{cabcababacaba}$
- $T = \text{aba}$
- Three occurrences:
  - cabc**aba**bacaba
  - cabcab**aba**caba
  - cabcababac**aba**

Naive string matching algorithm

- For each substring of length m in S,
- check if that substring is equal to T.

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `  ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `  ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `    ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

- S: bacbababaabcbab
- T: ababaca

Naive string matching algorithm

- S: `bacbababaabcbab`
- T: `ababaca`

Naive string matching algorithm

```
int string_match(const string &s, const string &t) {
    int n = s.size(),
        m = t.size();

    for (int i = 0; i + m - 1 < n; i++) {
        bool found = true;
        for (int j = 0; j < m; j++) {
            if (s[i + j] != t[j]) {
                found = false;
                break;
            }
        }
        if (found) {
            return i;
        }
    }

    return -1;
}
```

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case

Naive string matching algorithm

- Double for-loop
  - outer loop is $O(n)$ iterations
  - inner loop is $O(m)$ iterations worst case
- Time complexity is $O(nm)$ worst case
- Can we do better?

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
    - S: `bacbababaabcbab`
    - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `   ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `    ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- The KMP algorithm avoids useless comparisons:
  - S: `bacbababaabcbab`
  - T: `ababaca`
- The number of shifts depend on which characters are currently matched

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \ldots k] \text{ is a suffix of } T[1 \ldots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	a	b	a	b	a	c	a
$\pi[i]$	0	0	1	2	3	0	1

Knuth–Morris–Pratt algorithm

- How are the number of shifts determined?
- Let $\pi[q] = \max\{k : k < q \text{ and } T[1 \dots k] \text{ is a suffix of } T[1 \dots q]\}$
- Example:

i	1	2	3	4	5	6	7
$T[i]$	`a`	`b`	`a`	`b`	`a`	`c`	`a`
$\pi[i]$	0	0	1	2	3	0	1

- If, at position i, q characters match (i.e. $T[1 \dots q] = S[i \dots i + q - 1]$), then
  - if $q = 0$, shift pattern 1 position right
  - otherwise, shift pattern $q - \pi[q]$ positions right

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions

Knuth–Morris–Pratt algorithm

- Example:
  - S: `bacbababaabcbab`
  - T: `ababaca`
  - 5 characters match, so $q = 5$
  - $\pi[q] = \pi[5] = 3$
  - Then shift $q - \pi[q] = 5 - 3 = 2$ positions
  - S: `bacbababaabcbab`
  - T: `ababaca`

Knuth–Morris–Pratt algorithm

- Given π, matching only takes $O(n)$ time
- π can be computed in $O(m)$ time
- Total time complexity of KMP therefore $O(n + m)$ worst case

Knuth–Morris–Pratt algorithm

```
vector<int> kmppi(string &p) {
  int m = p.size(), i = 0, j = -1;
  vector<int> b(m + 1, -1);
  while(i < m) {
    while(j >= 0 && p[i] != p[j]) j = b[j];
    b[++i] = ++j;
  }
  return b;
}

vi kmp(string &s, string &p) {
  int n = s.size(), m = p.size(), i = 0, j = 0;
  vector<int> b = kmppi(p), a{};
  while(i < n) {
    while(j >= 0 && s[i] != p[j]) j = b[j];
    ++i; ++j;
    if(j == m) {
      a.push_back(i - j);
      j = b[j];
    }
  }
  return a; }
```

Tries

Sets of strings

- We often have sets (or maps) of strings
- Insertions and lookups usually guarantee $O(\log n)$ comparisons

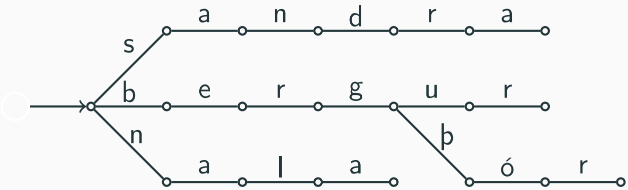
- But string comparisons are actually pretty expensive...

- There are other data structures, like tries, which do this in a more clever way

Tries

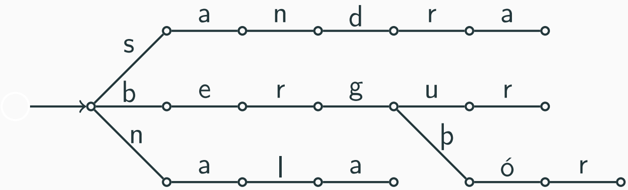
- Tries contain strings not at every node, but as paths in a tree.
- Each node only has a character and we say the trie contains the string if you can get it by walking along nodes starting at the root.
- The nodes can also carry additional data, quite a lot in fact, as we will see later.

Example



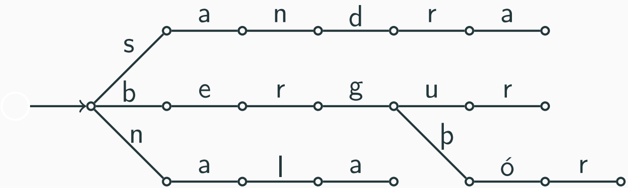
- Examples of strings in this trie include:

Example



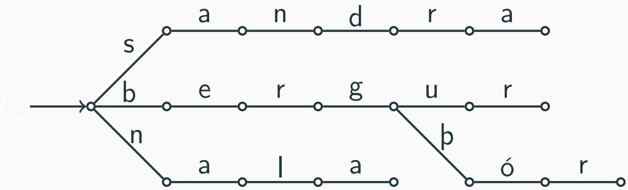
- Examples of strings in this trie include:

Example



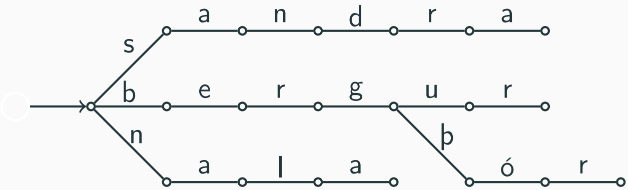
- Examples of strings in this trie include:
  - ,,sandra'',

Example



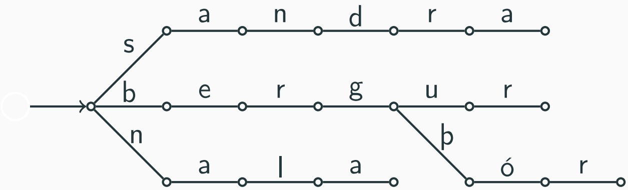
- Examples of strings in this trie include:
    - „sandra”,
    - „nala”,

Example



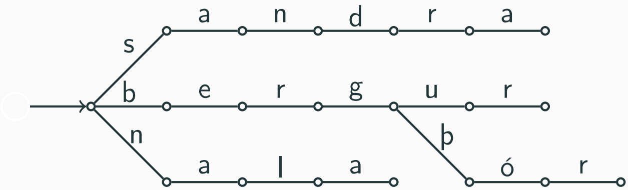
- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",

Example



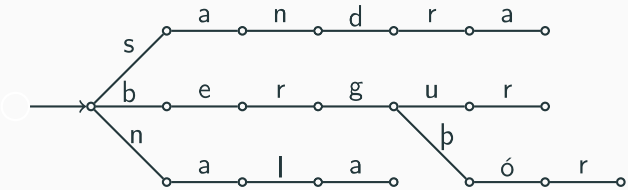
- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór",

Example



- Examples of strings in this trie include:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór",
  - „san" and

Example

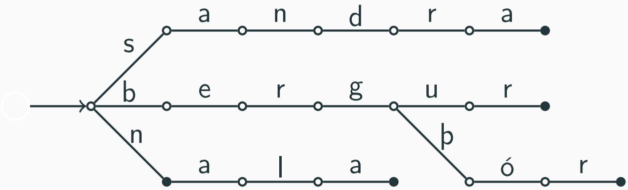


- Examples of strings in this trie include:
    - „sandra”,
    - „nala”,
    - „bergur”,
    - „bergþór”,
    - „san” and
    - „” (empty string)

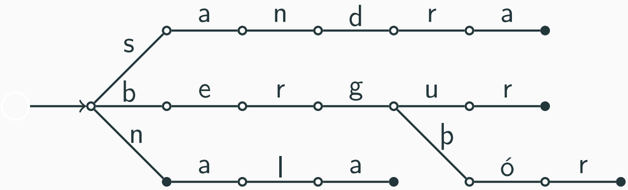
End nodes

- It is common to mark some nodes as end nodes.
- This is an example of extra data to put into nodes.
- Then we can consider a string s to be in the tree if you can walk through the tree to get the string **and** end at an end node.

Example

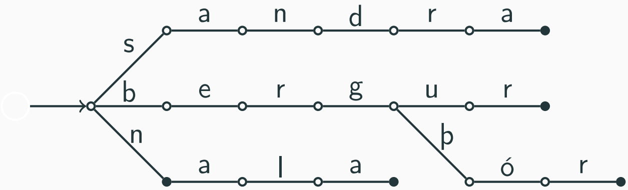


Example



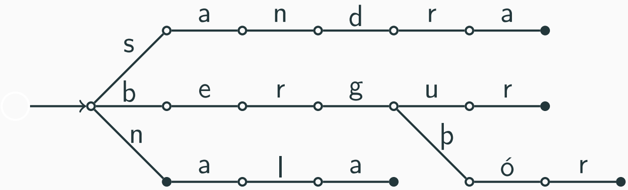
- The strings in the trie are:

Example



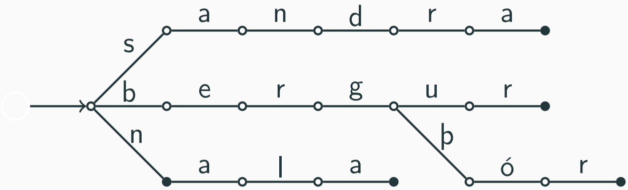
- The strings in the trie are:
    - „sandra",

Example



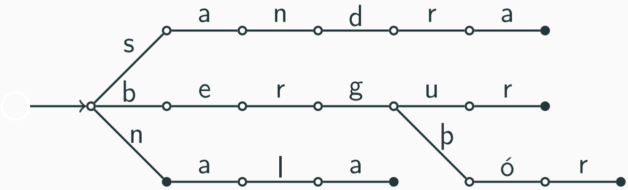
- The strings in the trie are:
  - „sandra",
  - „nala",

Example



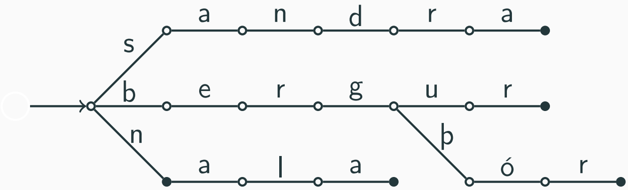
- The strings in the trie are:
    - „sandra",
    - „nala",
    - „bergur",

Example



- The strings in the trie are:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór" and

Example

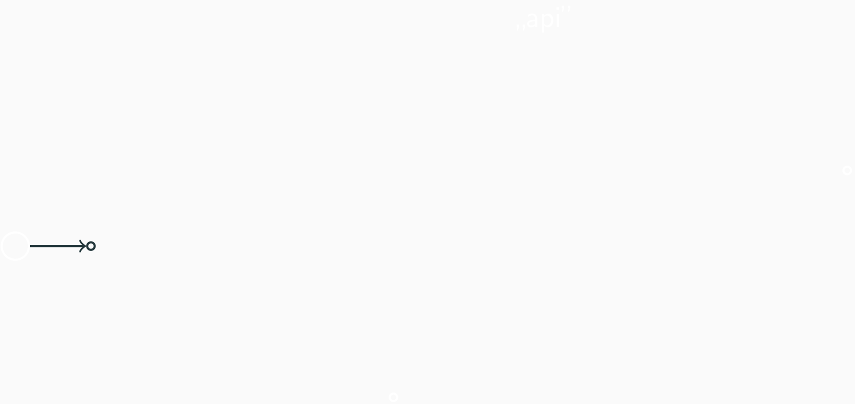


- The strings in the trie are:
  - „sandra",
  - „nala",
  - „bergur",
  - „bergþór" and
  - „n"

Adding strings

- What if we want to add a string to a trie?
- We walk through it as usual, but simply add nodes when we find ourself at a dead end with letters left to walk through.
- This increases the size of the tree by at most the size of the string.

Example



Example

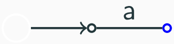
„api"

Example

„api"

Example

„pi”



Example

„i"

Example

Example

Example

„apar''

Example

,,apar''

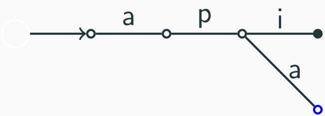
Example

„par"

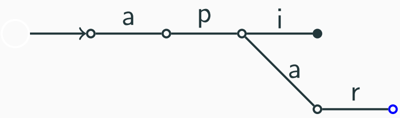
Example

„ar”

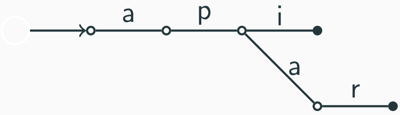
Example



Example

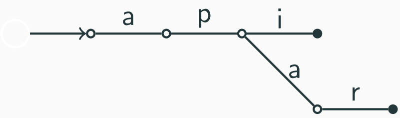


Example



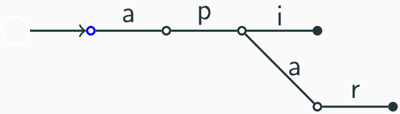
Example

„apaköttur"



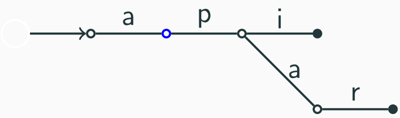
Example

„apaköttur”



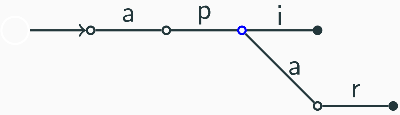
Example

„pakÖttur"



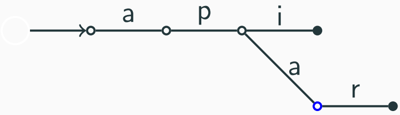
Example

„aköttur"



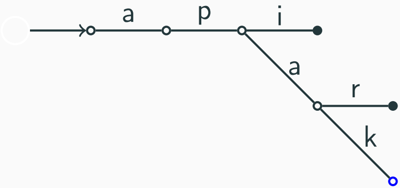
Example

„köttur"



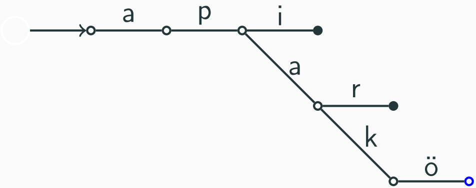
Example

„öttur”



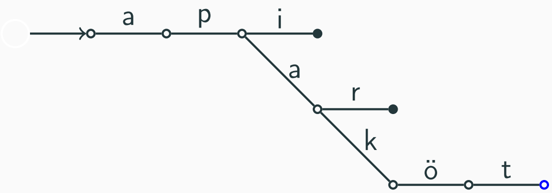
Example

„ttur"



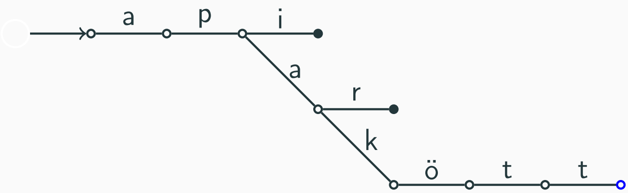
Example

„tur"



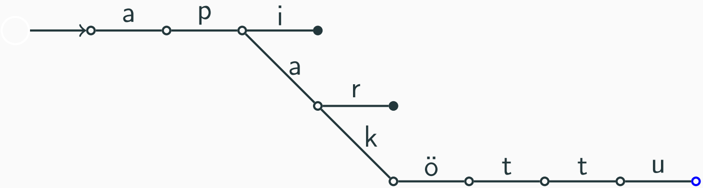
Example

„ur"

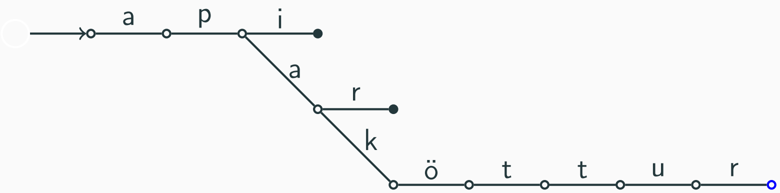


Example

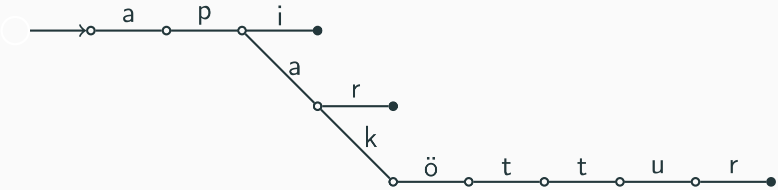
„r”



Example

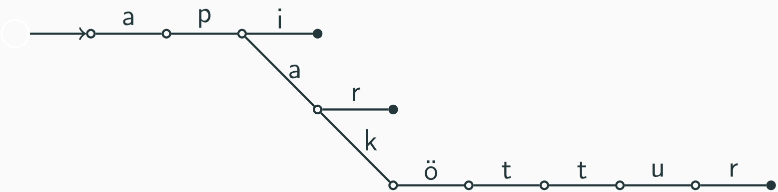


Example



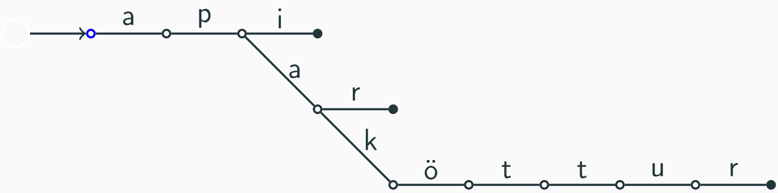
Example

„altari”



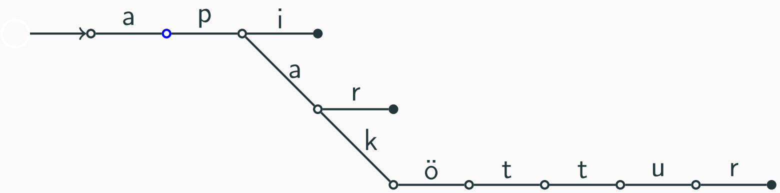
Example

„altari"



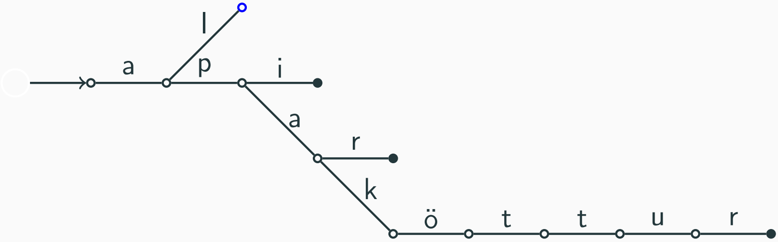
Example

„Itari"



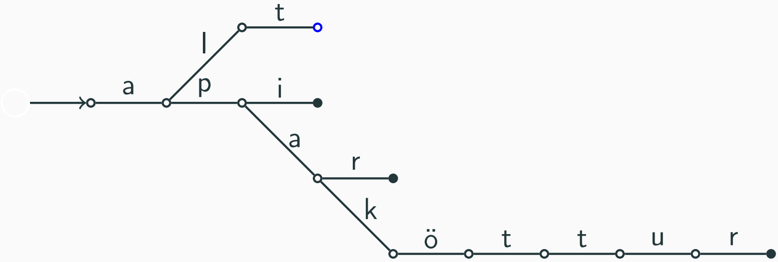
Example

„tari"



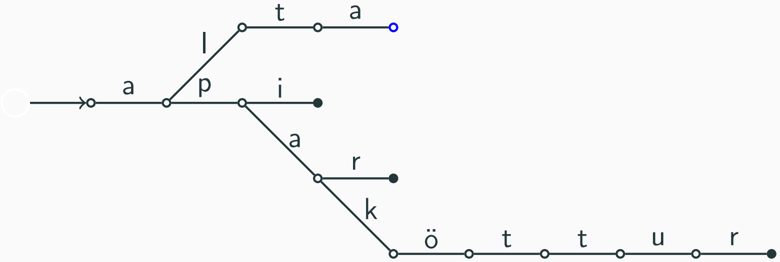
Example

„ari"



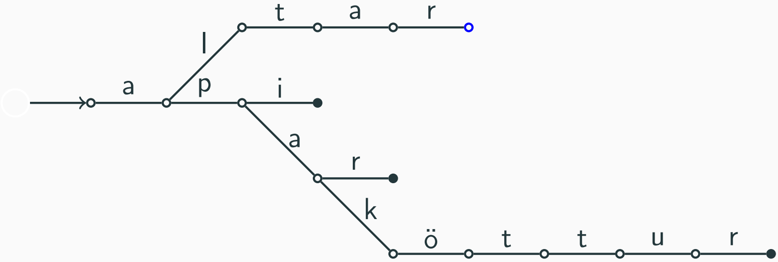
Example

„ri”

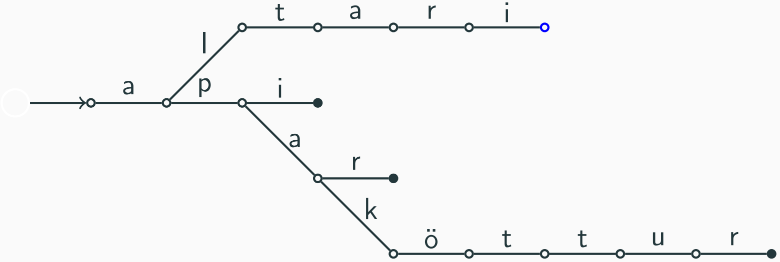


Example

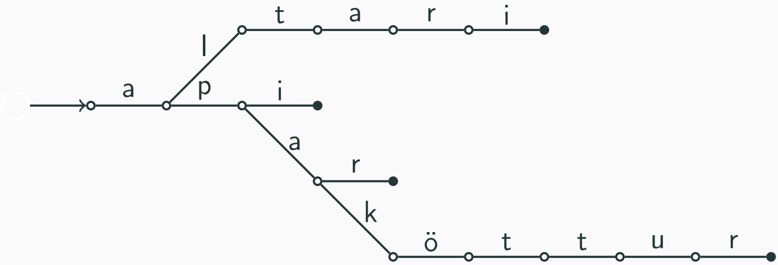
„i"



Example

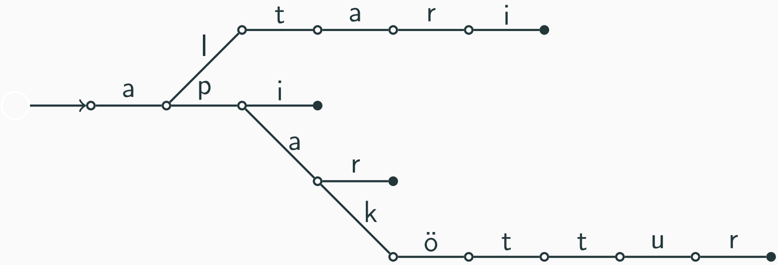


Example



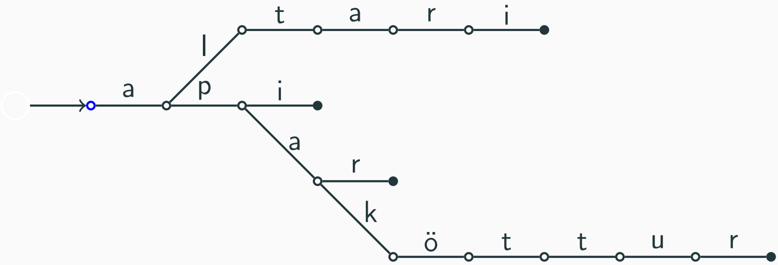
Example

„apaspil"



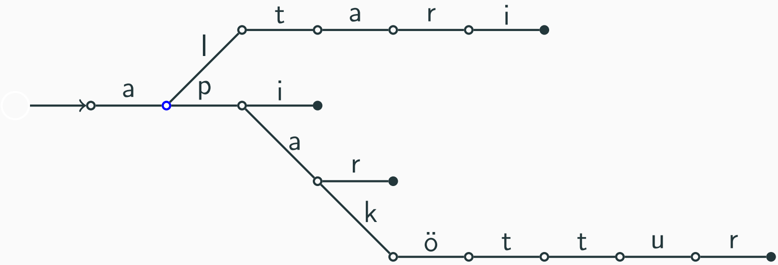
Example

„apaspil”



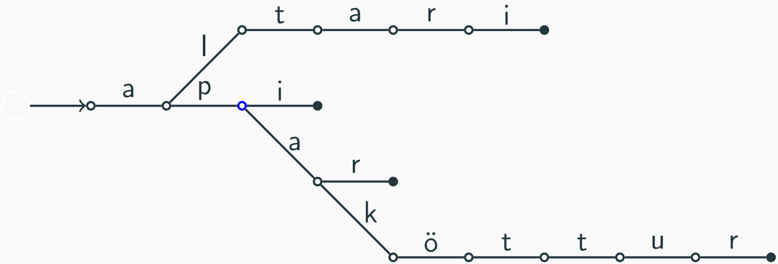
Example

„paspil"



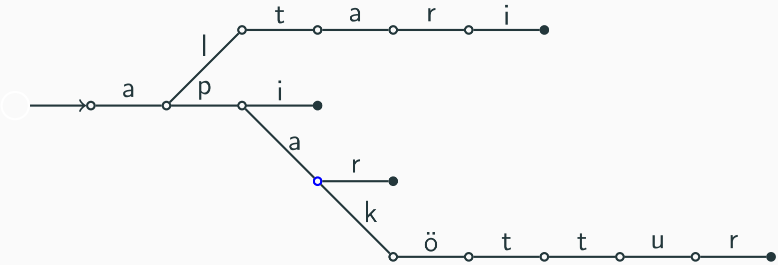
Example

„aspil"



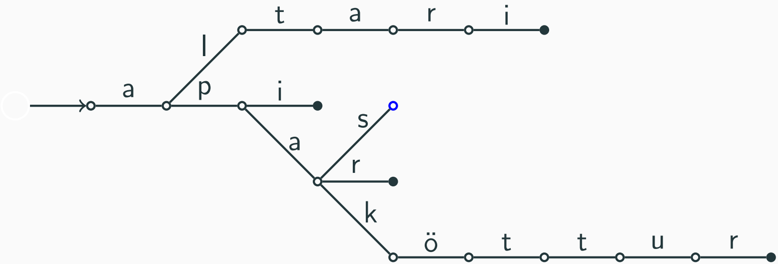
Example

„spil"



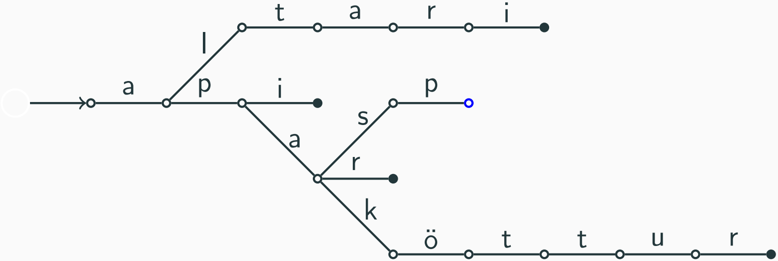
Example

„pil"

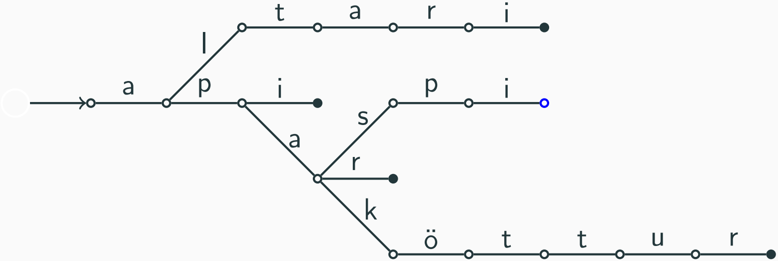


Example

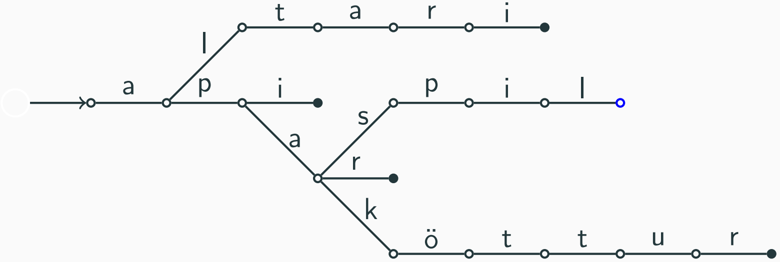
„il”



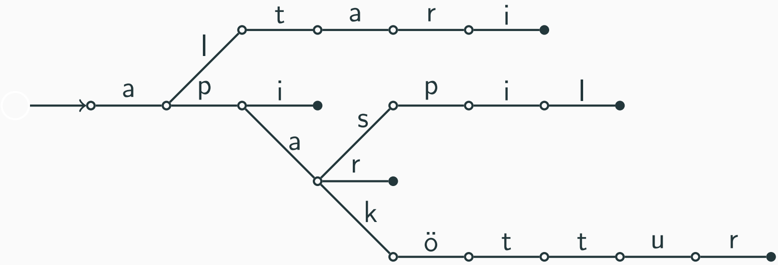
Example



Example

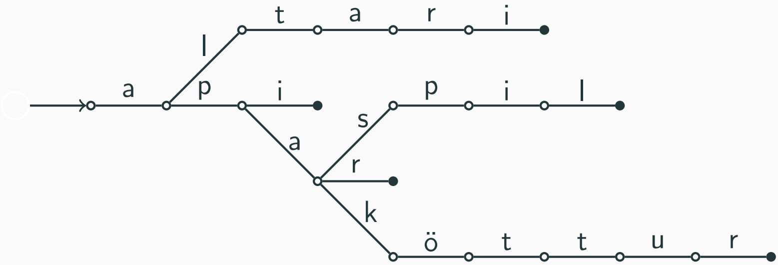


Example



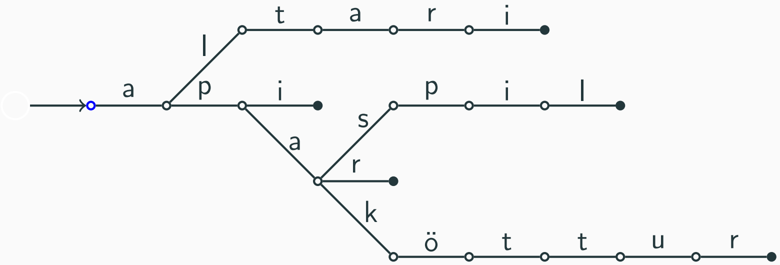
Example

„altaristafla"



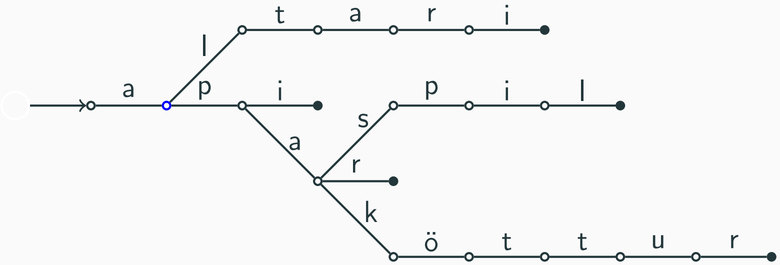
Example

„altaristafla"



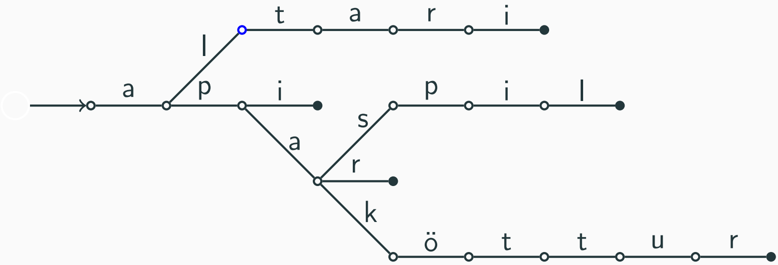
Example

„ltaristafla”



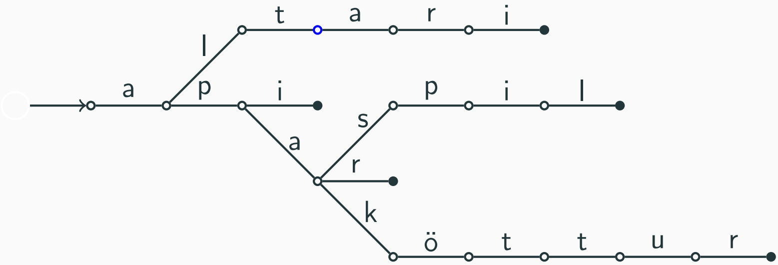
Example

„taristafla”



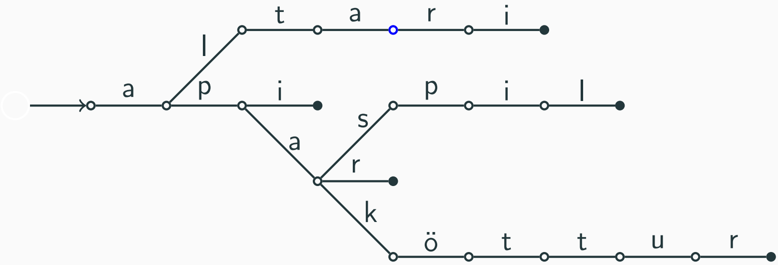
Example

„aristafla"



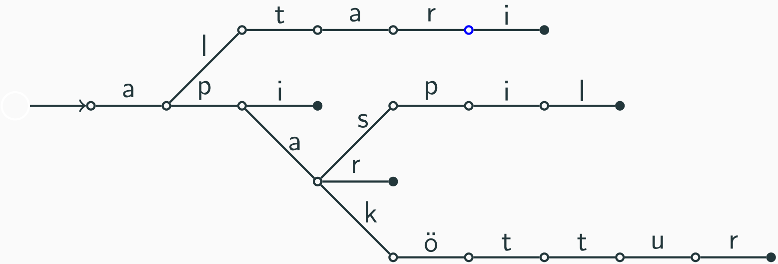
Example

„ristafla"



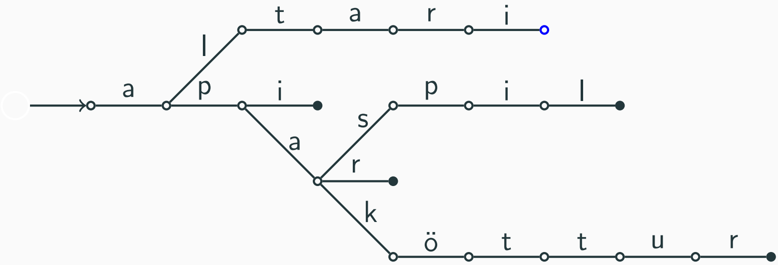
Example

„istafla"



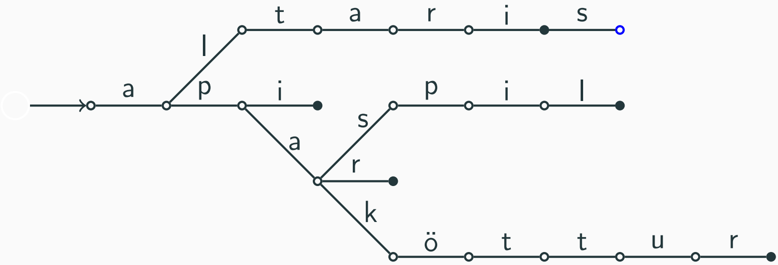
Example

„stafla"



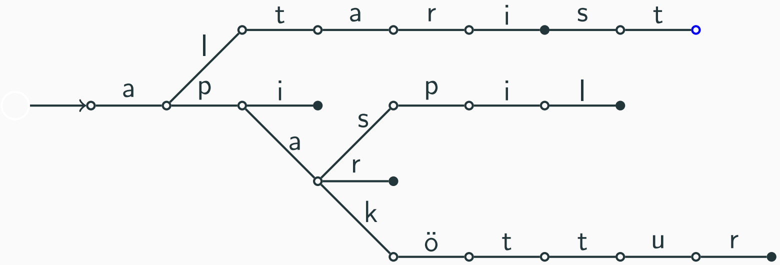
Example

„tafla"



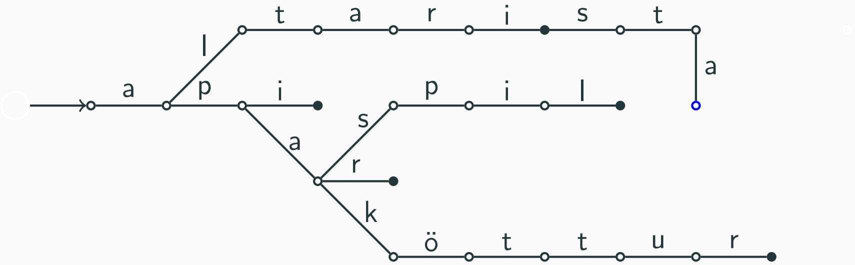
Example

„afla"



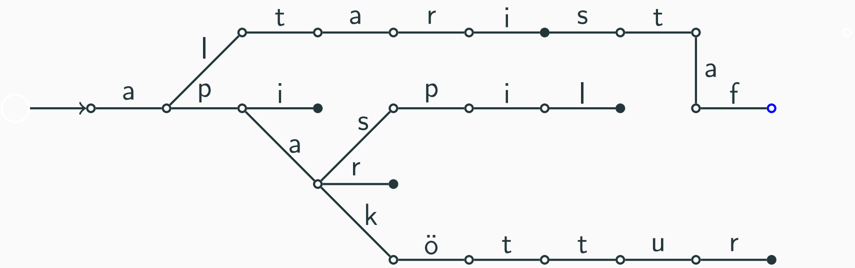
Example

„fla"



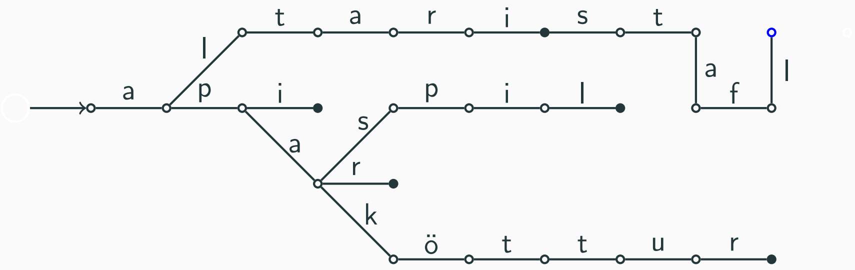
Example

„la”

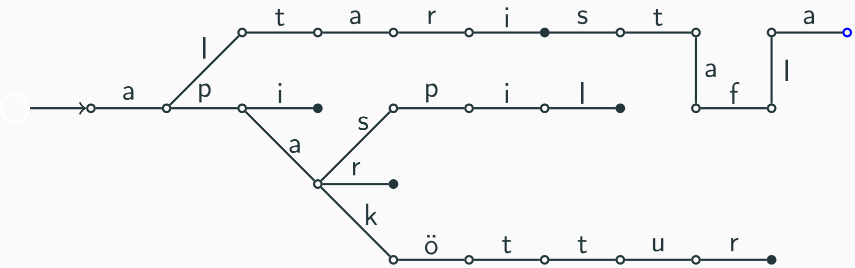


Example

„a"

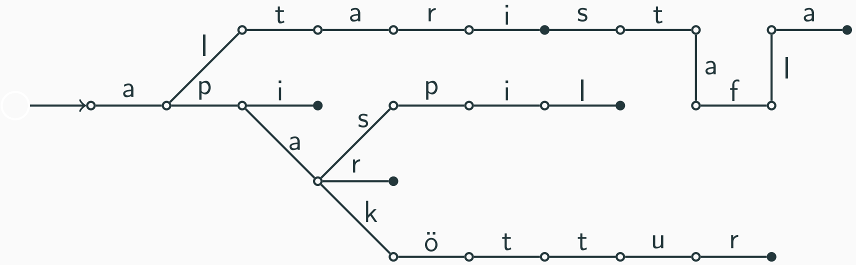


Example



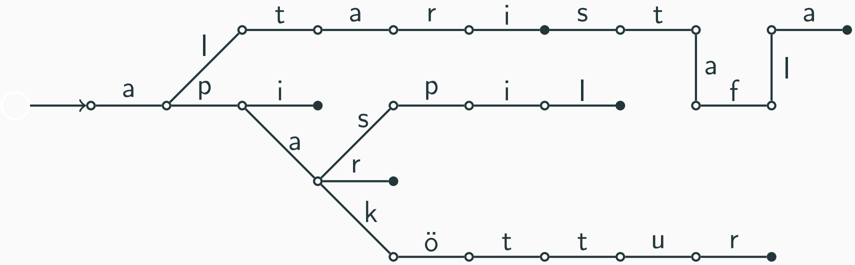
Example

„api"



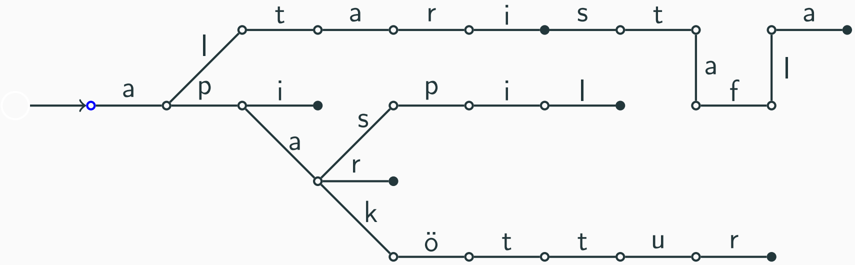
Example

„altarisganga”



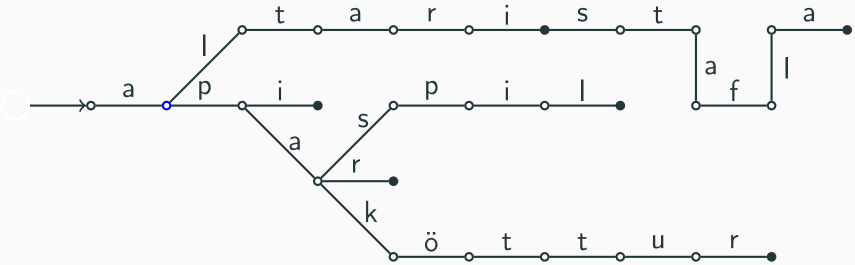
Example

„altarisganga”



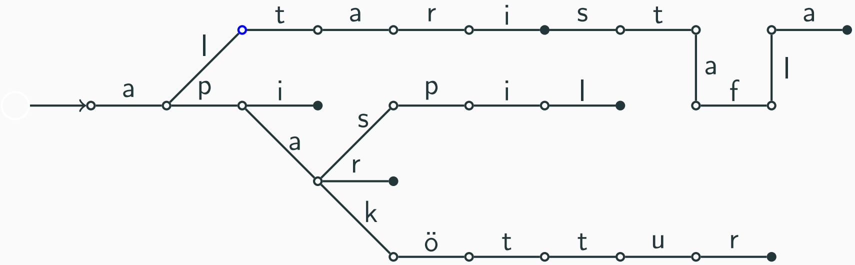
Example

„ltarisganga"



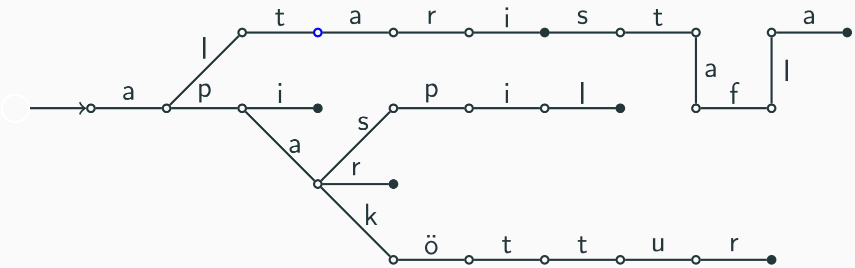
Example

„tarisganga"



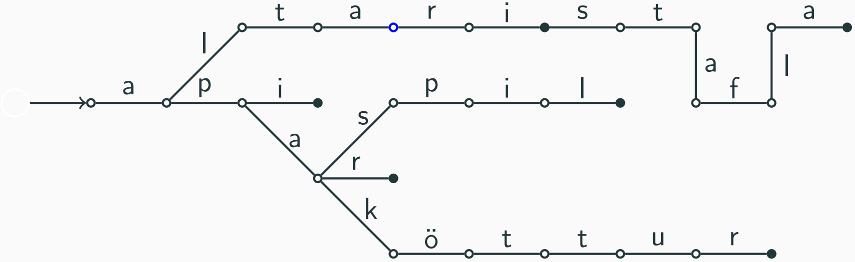
Example

„arisganga"



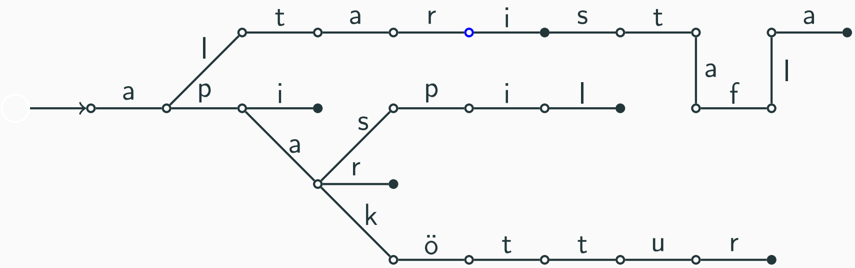
Example

„risganga"



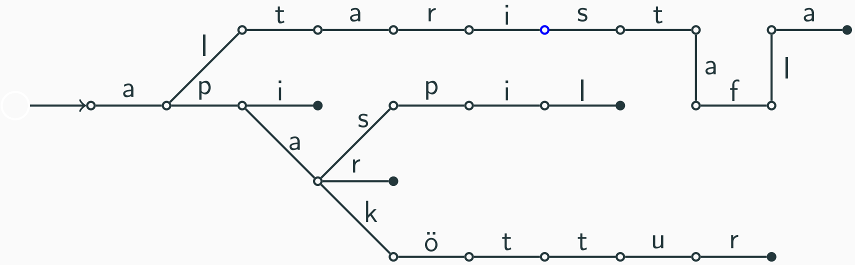
Example

„isganga”



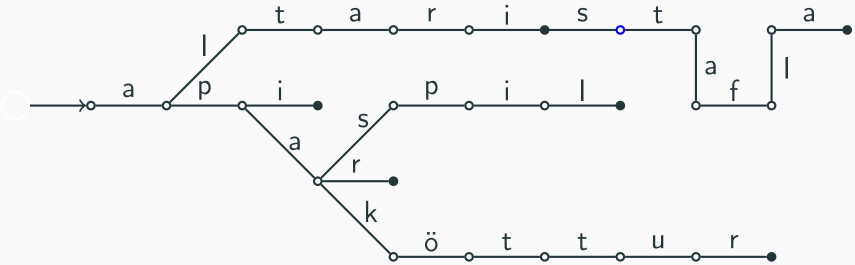
Example

„sganga"



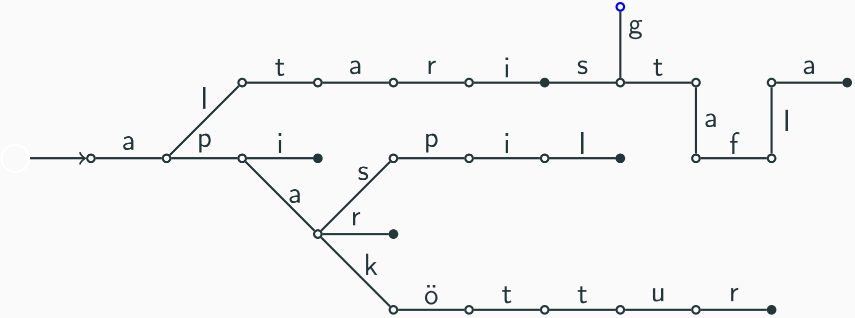
Example

„ganga"

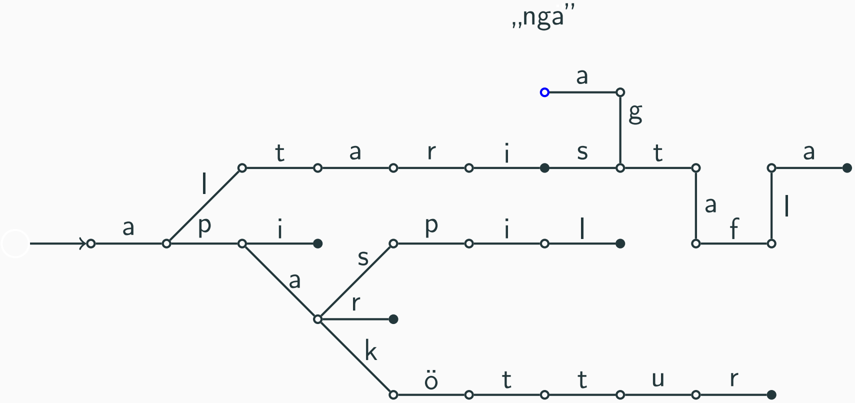


Example

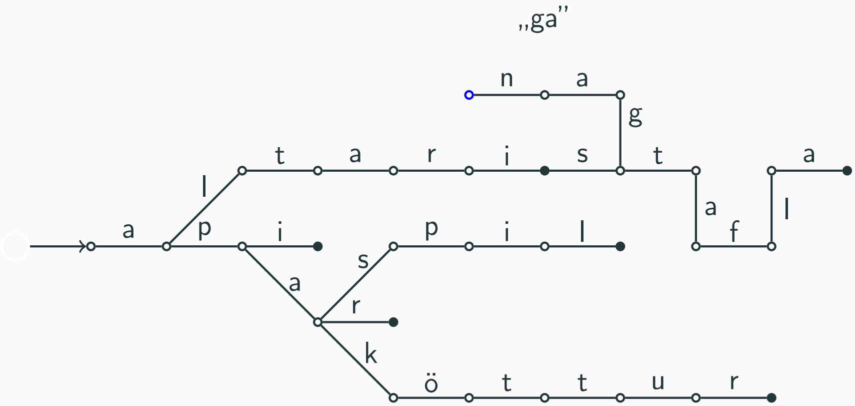
„anga"



Example

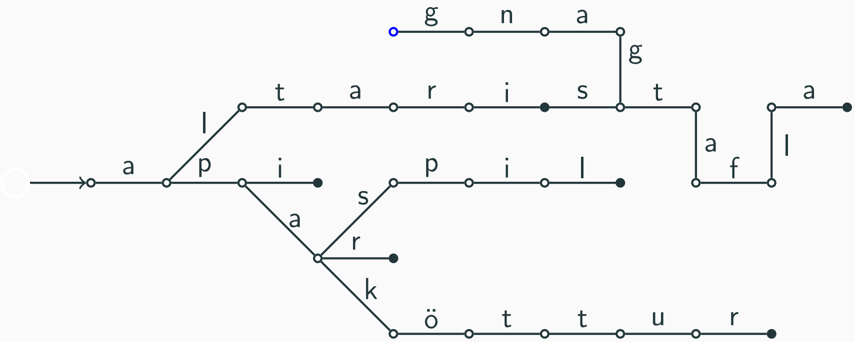


Example

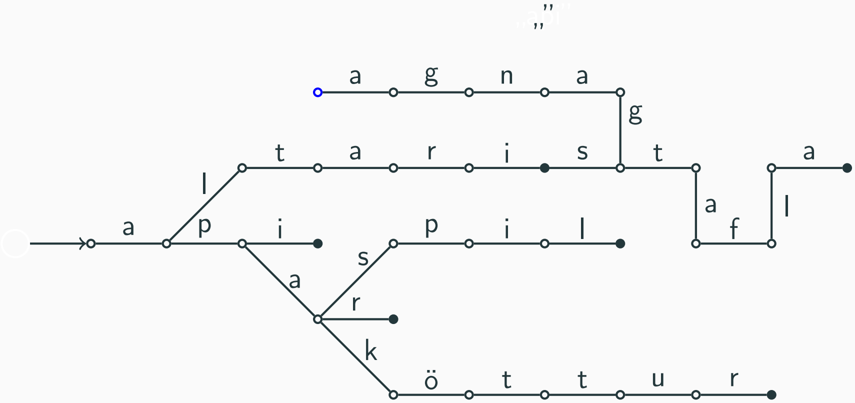


Example

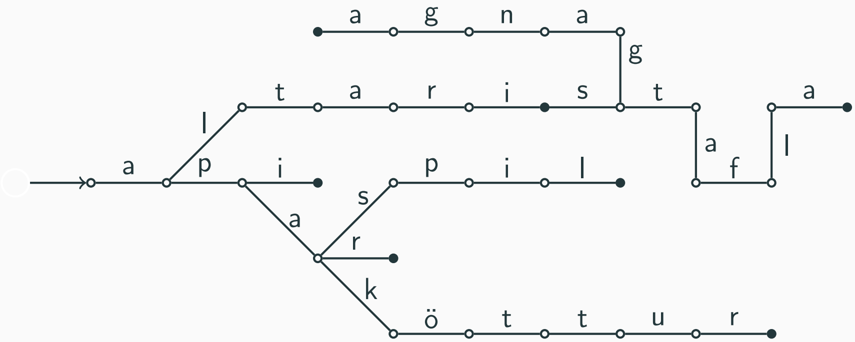
„a"



Example



Example



Tries

```
struct node {
    node* children[26];
    bool is_end;

    node() {
        memset(children, 0, sizeof(children));
        is_end = false;
    }
};
```

Tries

```
void insert(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            nd->children[*s - 'a'] = new node();

        insert(nd->children[*s - 'a'], s + 1);
    } else {
        nd->is_end = true;
    }
}
```

Tries

```
bool contains(node* nd, char *s) {
    if (*s) {
        if (!nd->children[*s - 'a'])
            return false;

        return contains(nd->children[*s - 'a'], s + 1);
    } else {
        return nd->is_end;
    }
}
```

Tries

```
node *trie = new node();

insert(trie, "banani");

if (contains(trie, "banani")) {
    // ...
}
```

Tries

- Time complexity?

- Let k be the length of the string we're inserting/looking for
- Lookup is $\mathcal{O}(k)$ and insertion is both $\mathcal{O}(k|\Sigma|)$
- The insertion takes this time because we might have to make k nodes, each needing $|\Sigma|$ pointers initialized
- This can be improved by using a map/dict for children instead, but that does make lookup slower, tradeoffs as usual

Sets and languages

- One way to store sets of strings is using a BST or hashmap based on a string data type.
- Tries allow us to store a set of strings more efficiently.
- Only works if we have a finite set of strings we're interested in.
- What if our set is: strings that contain "abba"?
- Can use Aho-Corasick algorithm to construct a modified trie
- Lets look at a more general approach

Sets and languages

Sets and languages

- A set of symbols is commonly known as an alphabet, or Σ.
- A formal language $\mathcal{L}$ over an alphabet Σ is a subset of $\Sigma*$.

Sets and languages

- A set of symbols is commonly known as an alphabet, or Σ.
- A formal language $\mathcal{L}$ over an alphabet Σ is a subset of $\Sigma*$.
- Many types exist.
    - regular,
    - context-free,
    - indexed,
    - context-sensitive,
    - recursive,
    - recursively enumerable

Regular languages

- A formal language that can be defined by a regular expression
- Words have the form $xy^n z$
- Recognized by Deterministic Finite Automata (DFA)
- Recognized by Non-Deterministic Finite Automata (NFA)
- Closed under union, intersection, concatenation, Kleene star and more.
- Can represent any finite or infinite regular languages using DFA/NFA

Deterministic Finite Automata

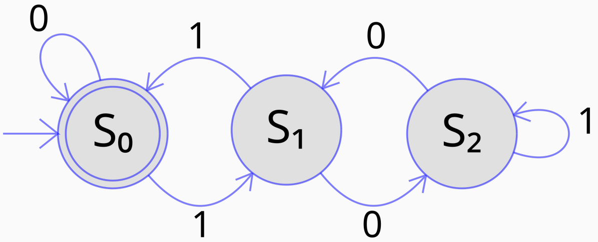
DFA

- Formally defined as the tuple $(Q, \Sigma, \delta, q_0, F)$ consisting of
  - a finite set of states Q,
  - a finite set of input symbols called the alphbet Σ,
  - a transition function $\delta : Q \times \Sigma \to Q$,
  - an initial state $q_0 \in Q$,
  - a set of accepting states $F \subseteq Q$.
- A word is accepted by the automaton if there exists a path starting in the initial state, following the transitions for the symbols in the word, finally ending in an accepting state.
- Can be complete or incomplete/partial.

DFA Example

- $Q = \{S_0, S_1, S_2\}$
- $\Sigma = \{0, 1\}$
- $q_0 = S_0$
- $F = \{S_0\}$
- δ given by the table

	0	1
S_0	S_0	S_1
S_1	S_2	S_0
S_2	S_1	S_2



Applications

- Regular Expression matching
- Parsing
- Natural Language Processing
- Video Game Character Behavior
- Password Generation (see Lykilorð on Kattis)

Applications

- Regular Expression matching
- Parsing
- Natural Language Processing
- Video Game Character Behavior
- Password Generation (see Lykilorð on Kattis)
- Either use a library such as automata-lib for Python...
- ...or implement yourself! (See assignments)

DFA Read

- Input is a string $w = w_1 w_2 \dots w_k$
- Set current state q as initial state q_0
- For each symbol w_i in w, set $q = \delta\left(q, w_i\right)$)
- Return `accept` if $q \in F$
- Otherwise return `reject`
- Time complexity is $O(k)$.

DFA Complement

- We have automata M recognizing language $\mathcal{L}$.
- We want automata M' recognizing language $\overline{\mathcal{L}}$.
- If M accepts w then M' must reject w, and vice versa.
- How to construct M'?

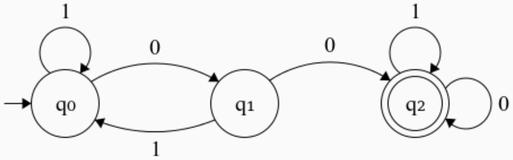
DFA Set Operations

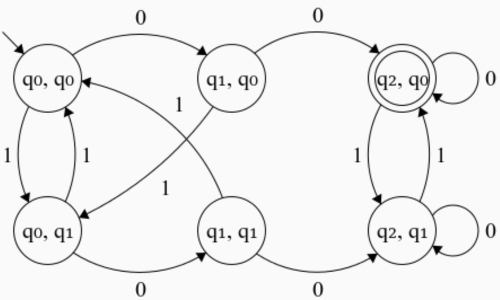
- We have automata M_1 recognizing language $\mathcal{L}_1$.
- We have automata M_2 recognizing language $\mathcal{L}_2$.
- We want automata M' recognizing language $\mathcal{L}_1 \cap \mathcal{L}_2$.
- If M_1 and M_2 accept w then M' must accept w, otherwise reject.
- How to construct M'?

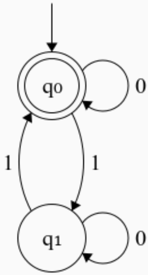
DFA Set Operations

- We have automata M_1 recognizing language $\mathcal{L}_1$.
- We have automata M_2 recognizing language $\mathcal{L}_2$.
- We want automata M' recognizing language $\mathcal{L}_1 \cap \mathcal{L}_2$.
- If M_1 and M_2 accept w then M' must accept w, otherwise reject.
- How to construct M'?
- Start with the cross product of M_1 and M_2.

DFA Cross Product







DFA Set Operations

- Union, Difference and Symmetric Difference handled similarly.
- Construction the same except for final states
- Other operations like Concatenation and Kleene Star handled differently