Data Structures and Algorithms – W13

String matching algorithms

Unification algorithm

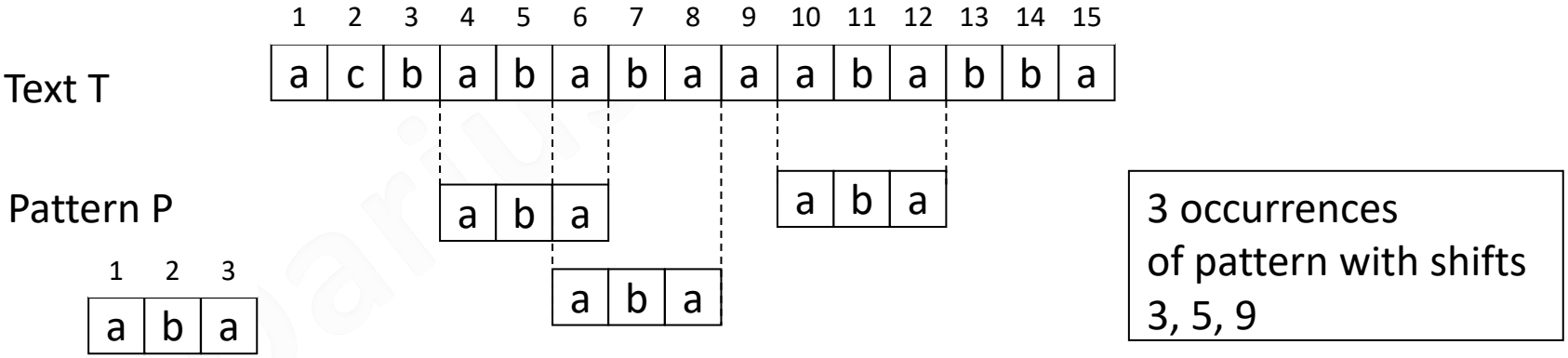
Contents

- String matching in the text:
  - Definitions
  - Naive algorithm – analysis
  - The Rabbin-Karp algorithm
  - An algorithm using finite automata
  - Knuth-Morris-Pratt (KMP) algorithm
- Unification:
  - Definitions
  - Unification algorithm

ε – epsilon
δ – delta
ϕ – fi
σ – sigma
π - pi

String matching problem

- We assume that :
  - The text is an array T [1..n] of length n
  - The pattern is an array P [1..m] with length m ≤ n..
  - Elements in P and T are symbols from a certain finite alphabet Σ.
    - E.g. $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$.
- The P and T texts are often called **words**.
- We say that pattern *P* ***occurs with shift s*** in text *T* (or, equivalently, that pattern *P* ***occurs beginning at position s + 1*** in text *T*) if $0 \le s \le n - m$ and $T[s + 1..s + m] = P[1..m]$
  - that is, if $T[s + j] = P[j]$, for $1 \le j \le m$.
- If *P* occurs with shift *s* in *T* , then we call *s* a ***valid shift***; otherwise, we call *s* an ***invalid shift***. The string-matching problem is the problem of finding **all** valid shifts with which a given pattern *P* occurs in a given text *T.*



Notation and terminology 1/2

- By Σ^* (pronounced "sigma-star") we mean a set of all texts (words) created from the symbols of the alphabet Σ. We consider only words of a finite length.
- A word with a length of zero, called an empty word and denoted by ε, also belongs to Σ^*.
- The length of the word *x* is designated as $|x|$.
- The concatenation (submission) of two words x and y, denoted as *xy*, is a word of length $|x| + |y|$ consisting of *x* symbols followed by *y* symbols.
- The word *w* is the prefix of the word *x* (what we denote as $w \sqsubset x$) when $x=wy$ for a certain word $y \in \Sigma^*$.
  - If $w \sqsubset x$, then $|w| \leq |x|$.
- The word w is the suffix of the word *x* (what we denote as $w \sqsupset x$), when x=yw for a certain word $y \in \Sigma^*$.
  - If $w \sqsupset x$, then $|w| \leq |x|$.
- The empty word ε is both the prefix and the suffix of each word.
- Example: ab $\sqsubset$ abcca and cca $\sqsupset$ abcca.
- For any strings *x* and *y* and any character *a*, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$.
- Relations $\sqsubset$ and $\sqsupset$ are transitive relations.

Notation and terminology 1/2

- **Overlapping-suffix lemma**

  Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

For brevity of notation:

- we denote the k-character prefix $P[1..k]$ of the pattern $P[1..m]$ by P_k.
  - Thus, $P_0 = \varepsilon$ and $P_m = P = P[1..m]$.
- we denote the k-character prefix of the text T as T_k.

Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \leq s \leq n-m$ such that $P \sqsupset T_{s+m}$.

The test "$x = y$" is assumed to take time $\Omega(t + 1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.

The problem of string matching occurs when looking for genes from the DNA sequence is a matching in the text of about $3*10^8$ symbols (alkalis A / T / G / C).

Naive algorithm - code

```
     Naive_String_Matcher(T,P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} for s:=0 to n-m do
{ 4}   if P[1..m]=T[s+1..s+m] then
{ 5}     print „patter occurs with shift” s
```

$T=a^n$, $P=a^m$ ⟶ (n-m+1)m character comparisons

$T=a^n$, $P=a^{m-1}b$

↓

Worse-case complexity: O((n-m+1)m)

T=„random”, P=„random” ⟶ average ≤2(n-m+1) character comparisons

↓

Average complexity: O(n-m+1)

The Rabin-Karp algorithm - idea

- For expository purposes, let us assume that S = {0, 1, 2, . . . , 9}
  - In the general case, we can assume that each character is a digit in radix-d notation, where d = |S|. We can then view a string of k consecutive characters as representing a length-k decimal number.
  - The character string „31415" thus corresponds to the decimal number 31,415.
- Given a pattern P[1..m], we let p denote its corresponding decimal value.
- In a similar manner, given a text T[1..n], we let t_s denote the decimal value of the length-m substring T[s + 1..s + m], for s = 0, 1, . . . , n-m.
- Certainly, t_s = p if and only if T[s+1..s+m] = P[1..m]; thus, s is a valid shift if and only if t_s = p. If we could compute p in time $\Omega(m)$ and all the t_s values in a total of $\Omega(n - m + 1)$ time, then we could determine all valid shifts s in time $\Omega(m) + \Omega(n-m+1) = \Omega(n)$ by comparing p with each of the t_s's.
- We can compute p in time $\Omega(m)$ using Horner's rule:
- p = P[m] + 10 (P[m - 1] + 10(P[m - 2] + · · · + 10(P[2] + 10P[1]))).
- The value t_0 can be similarly computed from T[1..m] in time $\Omega(m)$. To compute the remaining values $t_1, t_2, \ldots, t_{n-m}$ in time $\Omega(n - m)$, it suffices to observe that t_{s+1} can be computed from t_s in constant time, since

$$t_{s+1} = 10\left(t_s - 10^{m-1}T[s+1]\right) + T[s+m+1]$$

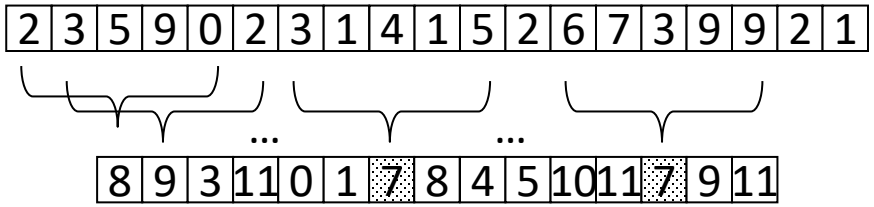
T=..314152.., m=5, t_s=31415

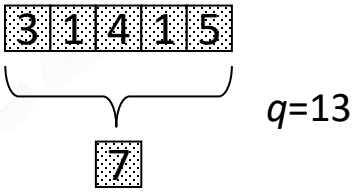
t_{s+1}=10(31415-10000∗3)+2=14152

Problem! : *p* and t_s may be **too large** to work with conveniently.

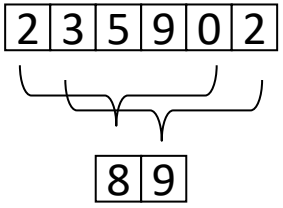
The Rabin-Karp algorithm

- We can compute p and the t_s's modulo a suitable modulus q.





q=13



35902 ≡ (23590 – 3*10000)*10+2 (mod 13)
≡ (8 – 2*3)*10 + 2 (mod 13)
≡ 9

$$t_{s+1} = \left(d\left(t_s - T[s+1]h\right) + T[s+m+1]\right) \bmod q \qquad \text{where} \qquad h \equiv d^{m-1} \left(\bmod q\right)$$

The Rabin-Karp algorithm - code

```
     Rabin_Karp_Matcher(T,P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} h := d^(m-1) mod q
{ 4} p := 0
{ 5} t_0 := 0
{ 6} for i:=1 to m do
{ 7}   p := (d*p+P[i]) mod q
{ 8}   t_o := (d*t_0+T[i]) mod q
{ 9} for s:=0 to n-m do
{10}   if p=t_s then
{11}     if P[1..m]=T[s+1..s+m] then
{12}       print „patter occurs with shift” s
{13}   if s < n-m then
{14}     t_s+1:=(d*(t_s-T[s+1]*h)+T[s+m+1]) mod q
```

The modulus *q* is typically chosen as a prime such that *dq* just fits within one computer word (double word)

The Rabin-Karp algorithm - analysis

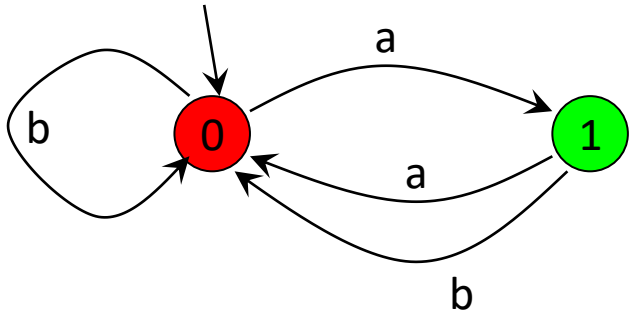
- The analysis of the complexity of algorithms is often divided into two phases:
  - **Preprocessing**: performed to prepare data for operation. If some data is repeated in the next instance of the problem, you do not have to do it (here are the calculations on the pattern).
  - **The right algorithm**: uses data from preprocessing and other data.
- The procedure RABIN-KARP-MATCHER needs $\Omega(m)$ time for preprocessing (calculation of p values).
- In the worse case, the comparison will take the time $\Omega((n-m+1)m)$
  - For example, for $P = a^m$ and $T = a^n$ all offsets are correct and you always have to do a string comparison over time $\Omega(m)$
- In many applications, we expect that the number of instances of the pattern is small, it can be assumed that the order O(1). Then, after the heuristic analysis, it turns out that the time of analyzing the whole text will be in the order O(n), which will give a total time equal to O(n).

Finite automaton 1/2

- A finite automaton M is a 5-tuple (Q, q_0, A, Σ, δ), where:
  - Q is a finite **set of states**,
  - $q_0 \in Q$ is the **start state**,
  - $A \subseteq Q$ is a distinguished set of **accepting states**,
  - Σ is a finite **input alphabet**,
  - δ is a function $Q \times \Sigma : \rightarrow Q$, called the **transition function** of M.
- The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q and reads input character a, it moves ("makes a transition") from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M is said to have **accepted** the string read so far. An input that is not accepted is said to be **rejected**.

Q = {0, 1}, q_o=0, A={1} Σ = { a, b},

δ	symbol a	b
state 0	1	0
1	0	0



Finite automaton 1/2

- A finite automaton *M* induces a function ϕ , called the ***final-state function***, from Σ^* to *Q* such that $\phi(w)$ is the state *M* ends up in after scanning the string *w*. Thus, *M* accepts a string *w* if and only if $\phi(w) \in A$. The function ϕ is defined by the recursive relation:

$$\phi(\varepsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a), \quad for\ w \in \Sigma^*, a \in \Sigma$$

We define an auxiliary function σ , called the ***suffix function*** corresponding to *P*. The function σ is a mapping from Σ^* to {0, 1, . . . , *m*} such that $\sigma(x)$ is the length of the longest prefix of *P* that is a suffix of *x*:

$\sigma(x) = \max \{ k: P_k \sqsupset x\}$

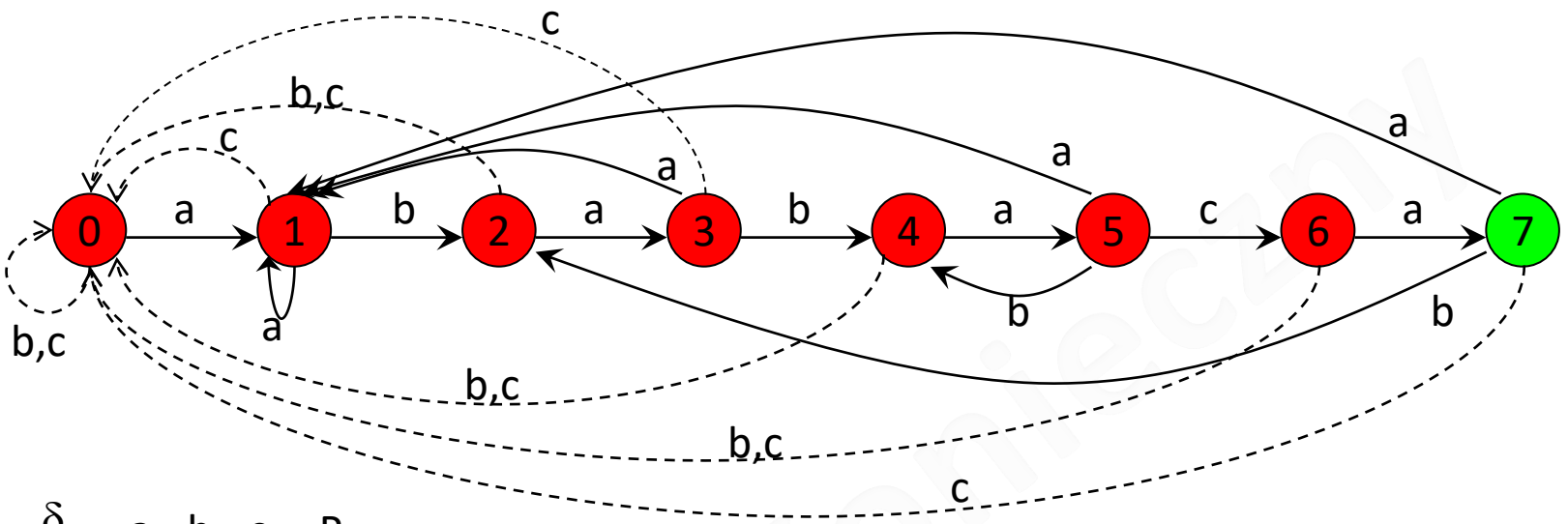
We define the string-matching automaton that corresponds to a given pattern *P*[1..*m*] as follows:

- The state set *Q* is {0, 1, . . . , *m*}. The start state q_0 is state 0, and state *m* is the only accepting state.

- The transition function δ is defined by the following equation, for any state *q* and character *a*:

$$\delta(q, a) = \sigma(P_q a)$$

Automaton - example



δ	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	-	1	2	3	4	5	6	7	8	9	10	11
T[i]	-	a	b	a	b	a	b	a	c	a	b	a
state ϕ(T[i])	0	1	2	3	4	5	4	5	6	7	2	3

Finite automaton matcher - code

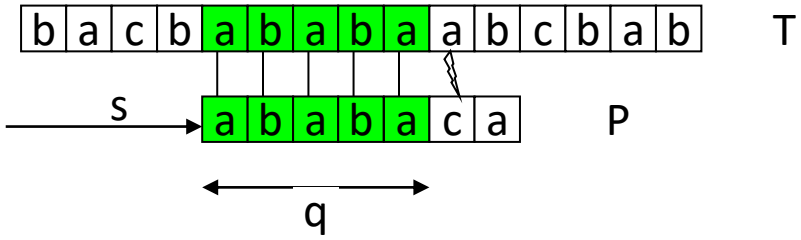
```
     Finite_Automaton_Matcher(T,P)
{ 1} n := length[T]
{ 2} q := 0
{ 3} for i:=1 to n do
{ 4}   q := δ(q,T[i])
{ 5}   if q = m then
{ 6}     s := i - m
{ 7}     print „patter occurs with shift” s
```

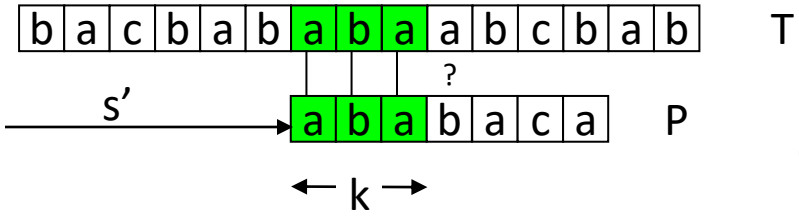
```
     Compute_Transition_Function(P,Σ)
{ 1} m := length[P]
{ 2} for q:=0 to m do
{ 3}   for a∈Σ do
{ 4}     k := min(m+1,q+2)
{ 5}     repeat
{ 6}       k := k - 1
{ 7}     until P_k ⊐ P_q a
{ 8}     δ(q,a) := k
{ 9} return δ
```

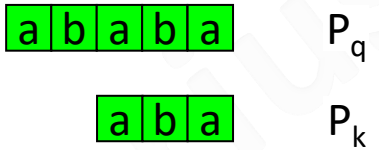
Finite automaton matcher - analysis

- The running time of COMPUTE-TRANSITION-FUNCTION is $O(m^3|\Sigma|)$:
  - the outer loops contribute a factor of $m|\Sigma|$
  - the inner **repeat** loop can run at most $m + 1$ times
  - the test $P_k \sqsupset P_q a$ on line 6 can require comparing up to m characters.
- Much faster procedures exist; the time required to compute δ from P can be improved to $O(m|\Sigma|)$ by utilizing some cleverly computed information about the pattern P.
- With this improved procedure for computing δ, we can find all occurrences of a length-m pattern in a length-n text over an alphabet Σ with $O(m|\Sigma|)$ preprocessing time and $\Omega(n)$ matching time.

Knuth-Morris-Pratt's idea







- Given a pattern $P[1..m]$, the ***prefix function*** for the pattern P is the function $\pi : \{1, 2, \ldots, m\} \rightarrow \{0, 1, \ldots, m - 1\}$ such that

$\pi[q]=\max\{k: k<q \text{ and } P_k \sqsupset P_q\}$

- That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

KMP - code

```
     KMP_Matcher(T,P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} π := Compute_Prefix_Function(P)
{ 4} q :=0
{ 5} for i:=1 to n do
{ 6}   while q>0 and P[q+1]≠T[i] do
{ 7}     q := π[q]
{ 8}   if P[q+1] = T[i] then
{ 9}     q := q+1
{10}   if q=m then
{11}     print „patter occurs with shift” i-m
{12}     q := π[q]
```

```
     Compute_Prefix_Function(P)
{ 1} m := length[P]
{ 2} π[1] := 0
{ 3} k := 0
{ 4} for q:=2 to m do
{ 5}   while k>0 and P[k+1]≠P[q] do
{ 6}     k := π[k]
{ 7}   if P[k+1]=P[q] then
{ 8}     k := k+1
{ 9}   π[q] := k
{10} return π
```

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

KMP - analysis

- Using the potential method in the analysis of the amortized time, the COMPUTE-PREFIX-FUNCTION procedure is performed at the time $\Omega(m)$:
  - In the **for** loop (line 4) we can only raise the value of k by one in line 8
  - The **while** loop (line 5) can only lower the value of k, but never below 0. That means it can be made as many times as the number of times k will be increased.
  - The remaining instructions are fixed-time $O(1)$.
  - The outer loops are executed $\Omega(m)$ times, hence the overall time the COMPUTE-PREFIX-FUNCTION procedure has a complexity of $\Omega(m)$.
- Analogous analysis of the operation of the KMP-MATCHER procedure will lead to the conclusion that its complexity is $\Omega(n)$.

Complexity - comparison

Algorithm	Preprocessing time	Worse case Matching time
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$\Omega(m)$	$O((n-m+1)m)$
Finite automaton	$O(m\|\Sigma\|)$	$\Omega(n)$
Knutt-Morris-Pratt	$\Omega(m)$	$\Omega(n)$

Unification algorithm - definitions

- Unification - Is it possible to do some substitution for variables so that some two expressions are equal?
  - E.g. p(f(x),g(y)) and p(f(f(a)),g(z)) ?
  - Substitution {x←f(a),y←f(g(a)),z←f(g(a))}
  - We get: p(f(f(a)),g(f(g(a)))) and p(f(f(a)),g(f(g(a)))), or the same terms
  - We can use substitution {x←f(a), y ←a, z ←a}
  - Or even {x ←f(a), z ←y}
  - The last substitution is the **most general unifier** - MGU
  - In some cases MGU can not be found, e.g. g(x) and f(y)

Unification - definitions

For simplicity, let us limit the definition of a term to four principles:

- term ::= x where x is a variable
- term ::= f(list_of_terms) where f is a function symbol
- list_of_terms ::= term
- list_of_terms ::= term, list_of_terms

Example:

- f(g(a,b),h(c),g(c,c))
- f,g,h – function symbols
- a,b,c - variables

Unification - a general algorithm

- Input - a set of equations of terms
- Output - a set of equations in the form of MGU or the answer that MGU can not be found.

Transformations (*x* - any variable):

Rule1. Convert the equation t = x, where t is not a variable, to x = t.

Rule2. Delete the equation of the form x=x

Rule3. Let t'=t" are not variables. If the main function symbols of the terms t' and t" are different - finish the algorithm with a **negative response**. Else replace the equation $f(t'_1,\ldots,t'_k)=f(t''_1,\ldots,t''_k)$ by k formulas $t'_1=t''_1$, …, $t'_k=t''_k$.

Rule4. Let x = t be an equation such that the variable x occurs in a set of equations not only on the left side of this equation. If the variable x occurs in t - terminate the algorithm with a **negative response**. Else replace all occurrences of the variable x in other equations with term t.

The non-deterministic algorithm performs these transformations as long as possible. If possible transformations are finished - the set of equations is MGU.

Unification - deterministic algorithm

1. Set the equations in a queue (list)
2. Get the equation from the queue
3. If you can convert to **Rule**2, go back to step 2
4. If you can make the **Rule1** transformation, do it and put the new equation at the beginning of the queue. Return to step 2.
5. If you can make the **Rule3** transformations, do it and them put the new equations at the beginning of the queue. Return to step 2.
6. If you can make the **Rule4** transformation, do it, put the equation at the end of the queue, return to step 2
7. If all the equations were already analyzed - the equations in the queue constitute MGU, the end of the algorithm. If not, put the equation at the end and go back to step 2.

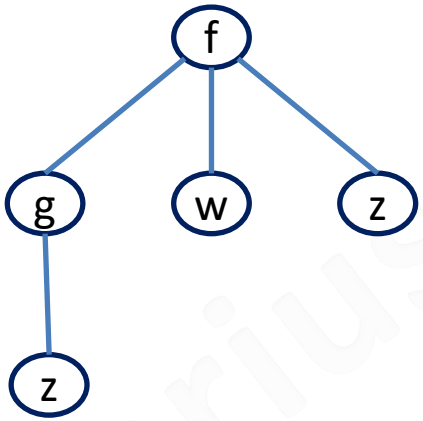
(in steps 5 and 6, the algorithm for **Rule3** and **Rule4** transformations may end in a negative response).

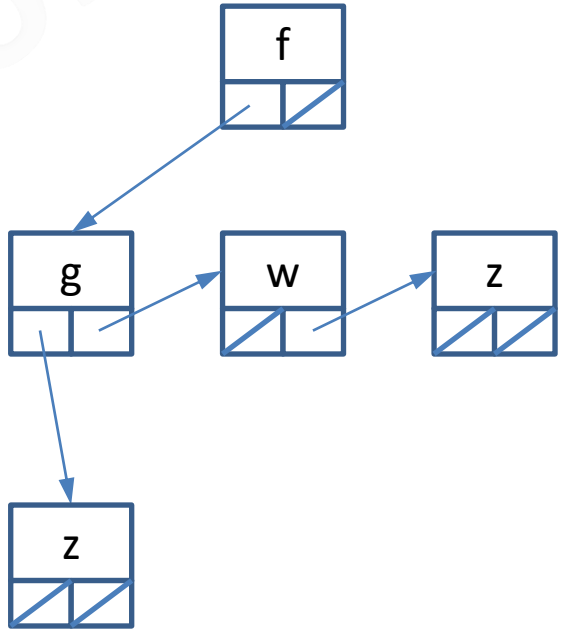
Unification - example

- { g(y)=x, f(x,h(x),y)=f(g(z),w,z) } Rule1
- {x=g(y), f(x,h(x),y)=f(g(z),w,z) } Rule4
- {f(g(y), h(g(y)),y)=f(g(z),w,z), x=g(y) } Rule3
- {g(y)=g(z), h(g(y))=w, y=z, x=g(y) } Rule3
- {y=z, h(g(y))=w, y=z, x=g(y) } Rule4
- {h(g(z))=w, z=z, x=g(z), y=z } Rule1
- {z=z, x=g(z), y=z, w=h(g(z)) } Rule2
- {x=g(z), y=z, w=h(g(z)) }
- End of the algorithm: the last set is the MGU

Term trees - implementation

- Remembering a term as a string of characters is uncomfortable, ineffective.
- Trees for terms can be implemented as trees with any number of children - just like in the binomial heap: reference to the leftmost child and siblings.
- For the unification algorithm, a reference to the parent remembered in the node is not needed.
- The internal nodes contain the function symbol
- The leaves contain the variable symbol





Terms trees in unification algorithm

- One equation is a pair of such trees.
- Easy analysis and execution of operations related to transformations Rule1, Rule2, Rule3, Rule4.