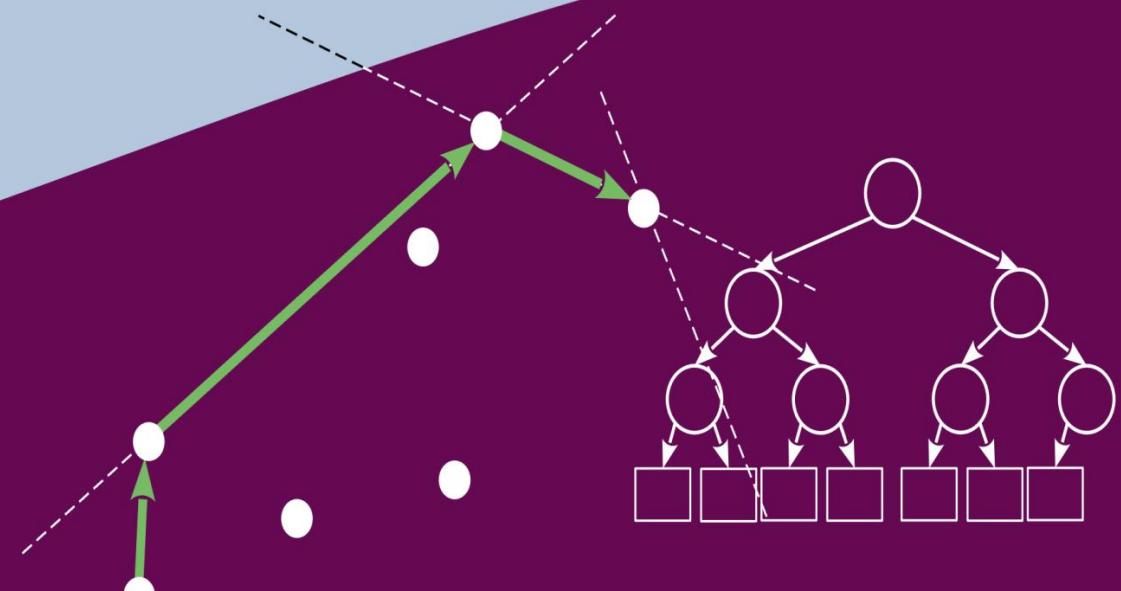
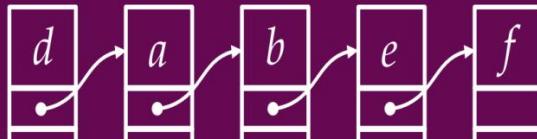
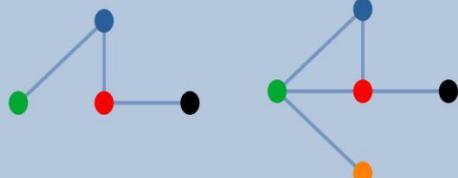


Advanced algorithms and data structures

Week 3: Data structures for characters
strings



Creative Commons

- you are free to: • share
 - duplicate, distribute and communicate the work to the public •
 - process the work
- under the following conditions: •
 - designation: you must acknowledge and mark the authorship of the work as it is specified by the author or licensor (but not in a way that suggests that you or your use of his work has his direct endorsement).
 - non-commercial: you may not use this work for commercial purposes. •
 - distribute under the same conditions: if you modify, transform, or create using this work, you may distribute the adaptation only under a license that is the same or similar to this one.



In the case of further use or distribution, you must make clear to others the license terms of this work.

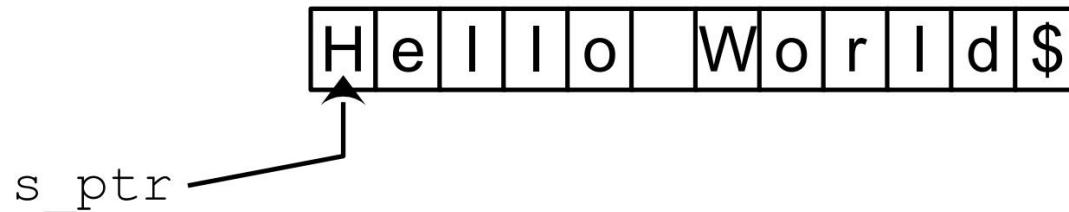
Any of the above conditions may be waived with the permission of the copyright holder.

Nothing in this license infringes or limits the author's moral rights.

The text of the license is taken from <http://creativecommons.org/>

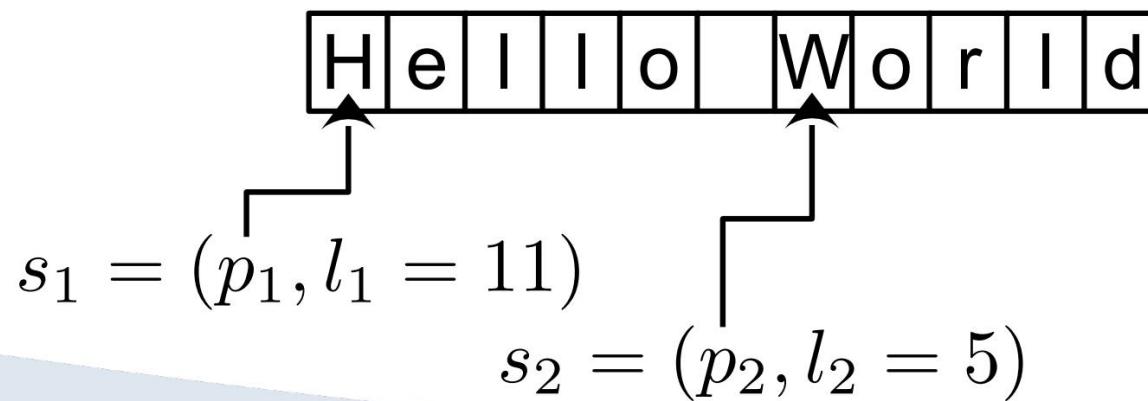
Character string representation (1)

- A string of characters is characterized by its alphabet Σ - it defines all the characters that can be found in the string
 - We have the mapping : $\Sigma \rightarrow \mathbb{Z}$, which adds an integer to the characters of the alphabet
 - This is how we define character tables, such as ASCII (1 byte – 8 bits) or Unicode (2 bytes – 16 bits)
- Basic representation - using the ASCII table, we define a character string as a sequence of bytes that ends with a NULL terminator (value 0) - and which we denote in the lecture as \$



Character string representation (2)

- Representation with an arranged pair, (\quad) , where is
 - – a pointer to the first character of a character string
 - – string length
- This kind of representation is interesting when we need to isolate a character substring in a simple way without creating a new instance
 - A good solution when we have a larger text in memory, so it needs to be in it map individual parts
 - For this type of representation, it is not necessary to use the NULL terminator



Prefix Tree (Trie) (1)

- Let's imagine a larger text in the computer's memory, and a set of keywords (or phrases) of interest in that text
- We define a set of key character strings (words, phrases, parts of text) as

$$= !:\{0 < \ddot{y} \bullet \text{where } \quad \}$$

N is the number of character strings

- We want to know if the character string is contained in the text
 - We can verify this by testing the hypothesis $\ddot{y} \bullet$ A naive implementation would loop through all the characters of the character strings in the set S, resulting in s

(-!l \ddot{y} # \$ l) l

Prefix Tree (Trie) (2)

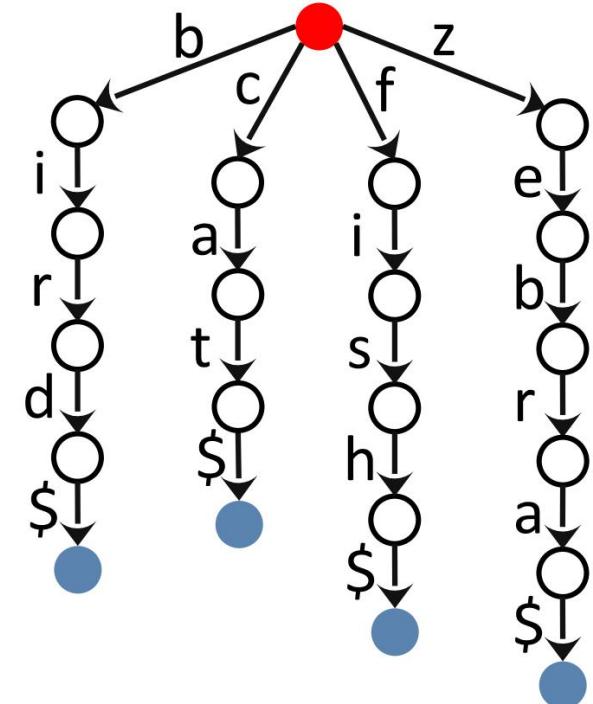
- Trie is an ordered triple = (, • ,) which represents an m-tree
Mapping function : ÿ ÿ maps the edges of the tree to the alphabet
 - The root node represents an empty character string
 - Each node of a Trie can have at most ~~c~~ children
 - Each output edge of a node must be mapped to a different character of the alphabet - two different outputs edges of a node cannot be mapped to the same character of the alphabet
 - Leaves are terminating nodes whose input edges are mapped to \$ (NULL terminator)
 - Trie is actually a finite deterministic automaton that enables parsing character string

Prefix Tree (Trie) (3)

- An example

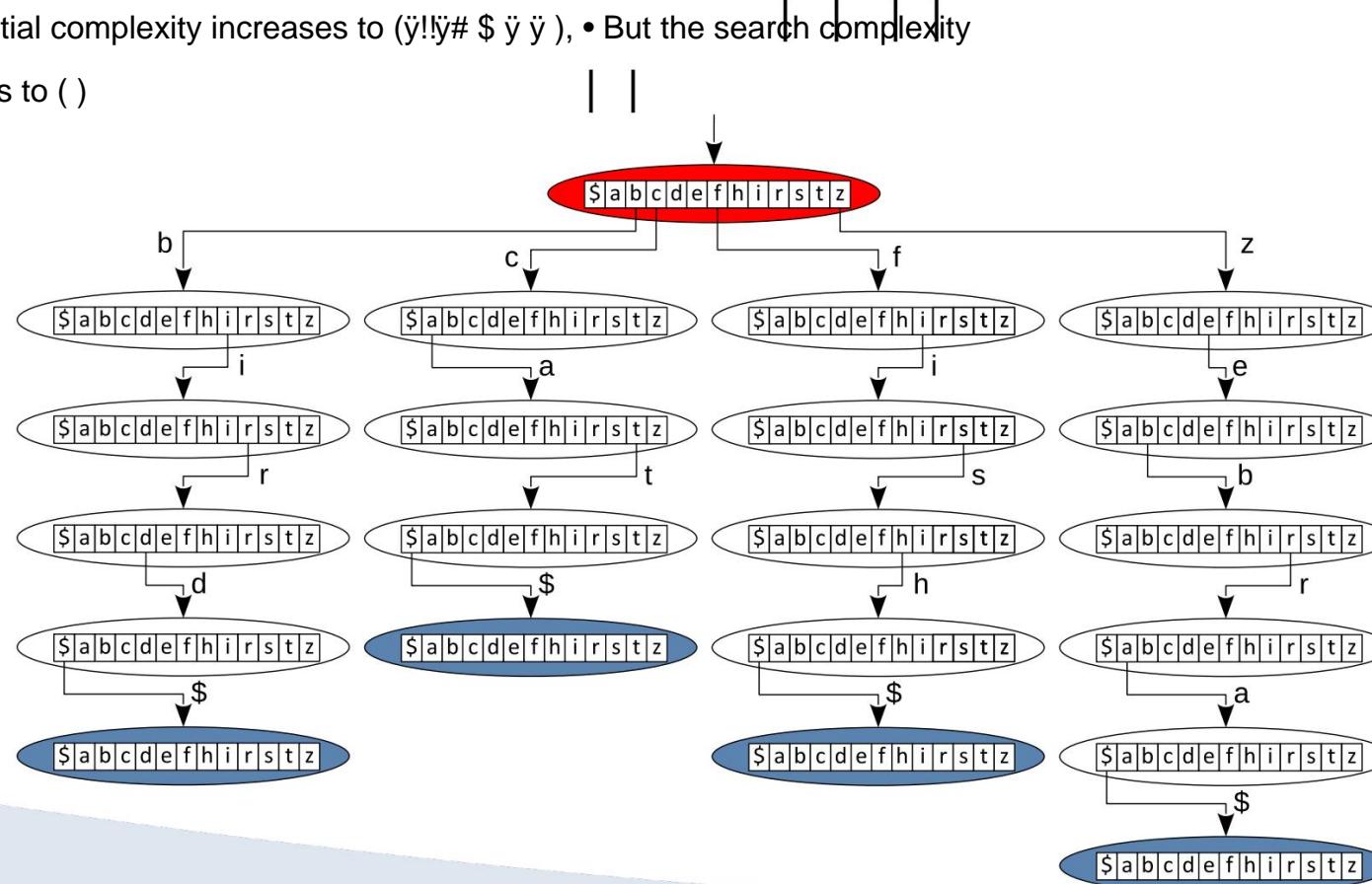
- The problem in such a tree is searching output edges

- How to determine if we have an exit edge that is identical to our next character in the sequence
 - A naive implementation would imply iterating over all exit edges
($\ddot{y} \ddot{y}$) | |
 - Is there a better way than that?



Prefix Tree (Trie) (4)

- In each node we put an alphabet mapping field with pointers to children
 - From the mapping : $\{b, c, f, z\}$ we know which field element we are looking at
 - If there is a pointer in that element, then that transition exists
 - The spatial complexity increases to $(\log n \cdot \log m)$, • But the search complexity decreases to $(\log m)$



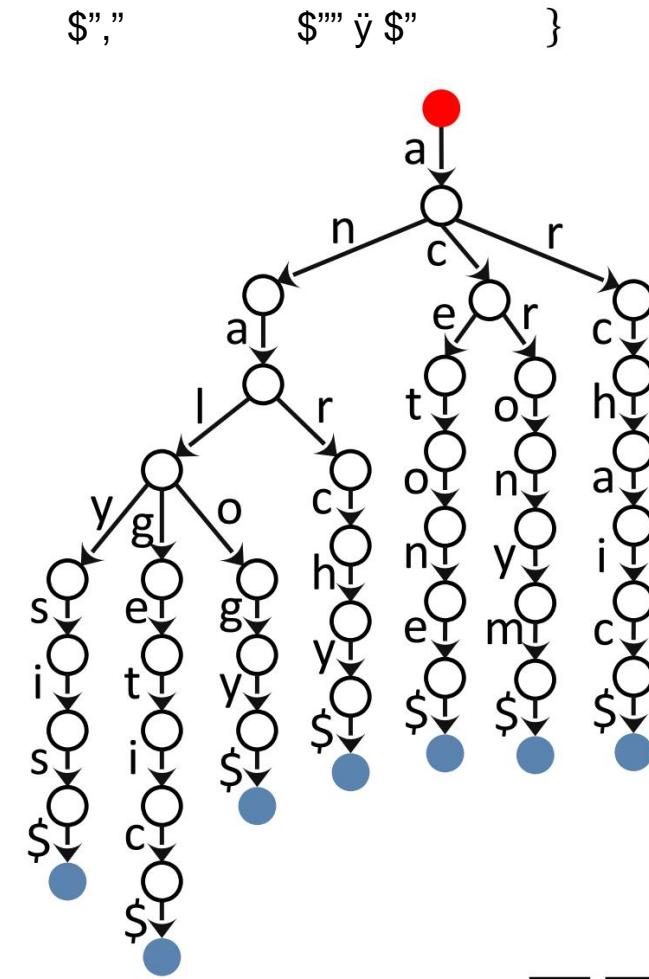
Prefix Tree (Trie) (5)

- Many character strings in can share a common prefix

$\{ \text{"an"}, \text{"al"}, \text{"log"}, \text{"ar"}, \text{"ace"}, \text{"aron"}, \text{"arc"}, \text{"an"}, \text{"et"}, \text{"on"}, \text{"y"}, \text{"m"}, \text{"ai"} \}$

a	n	a	l	y	s	i	s	
a	n	a	l	g	e	t	i	c
a	n	a	l	o	g	y		
a	n	a	r	c	h	y		
a	c	e	t	o	n	e		
a	c	r	o	n	y	m		
a	r	c	h	a	i	c		

- The very definition of Trie, which restricts that multiple output edges of a single node are mapped to the same character of the alphabet, forces us that common prefixes of character strings share the tree structure



Prefix Tree (Trie) (6)

- ## Search

- We take character by character from the character string and follow the transitions in the Trie
- If after \$ we reached a leaf, then (and the Trie) contains what led us
 - We have used all the characters in , to the leaf of the Trie
 - In any other case it was not found in
- What happens when we search for = "
- \$" in the previous Trie ?

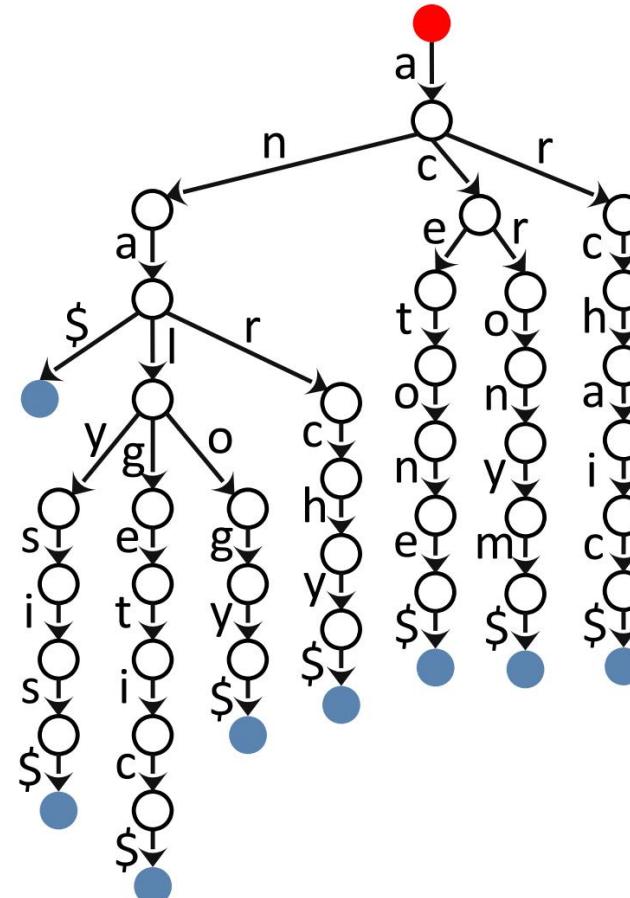
```

function SEARCHTRIE( $T, q$ )
   $cn \leftarrow root(T)$ 
  for  $ch \in q$  do
    if there is transition from  $cn$  for character  $ch$  to  $cn_{child}$  then
       $\triangleright array[ch] \neq NULL$  in  $cn$ 
       $cn \leftarrow cn_{child}$ 
    else
      return ( $false, cn$ )
    if  $cn$  is a leaf then
      return ( $true, cn$ )
  return ( $false, cn$ )

```

Prefix Tree (Trie) (7)

- An example $) = * \ddot{y} \{ " \$" \}$



Prefix Tree (Trie) (8)

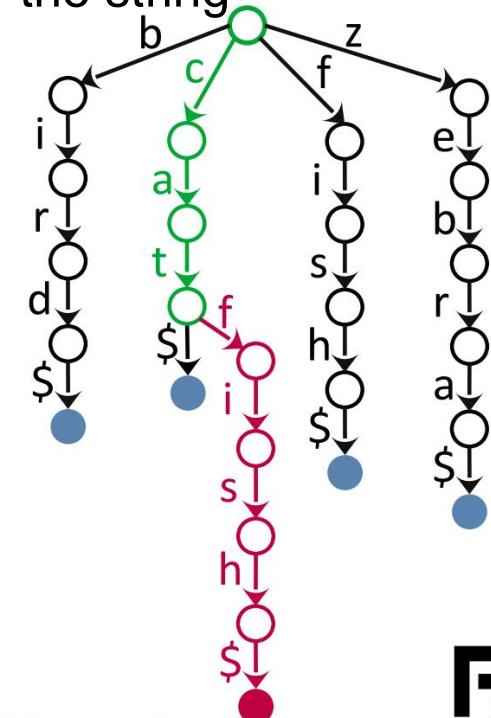
- Entering a new character string (• We insert)
 - start the search = • If we find the whole in the Trie • Otherwise we stop at the first character for which we do not have a transition in the Trie – it can also be \$

- Let's create the following Trie structure for the rest of the string

• Example: u ▪ we add = " ü\$"

```

procedure INSERTTRIE( $T, s$ )
   $cn \leftarrow \text{root}(T)$ 
  for  $ch \in s$  do
    if there is transition from  $cn$  for character  $ch$  to  $cn_{child}$  then
       $\triangleright \text{array}[ch] \neq \text{NULL}$  in  $cn$ 
       $cn \leftarrow cn_{child}$ 
    else
      create new node  $cn_{new}$ 
      add transition for character  $ch$  from  $cn$  to  $cn_{new}$ 
       $cn \leftarrow cn_{new}$ 
  
```



Prefix Tree (Trie) (9)

- Delete character string (remove)

- We start the search = • If we find

the whole in the Trie • Otherwise, then δ and deletion is possible
we stop with the deletion

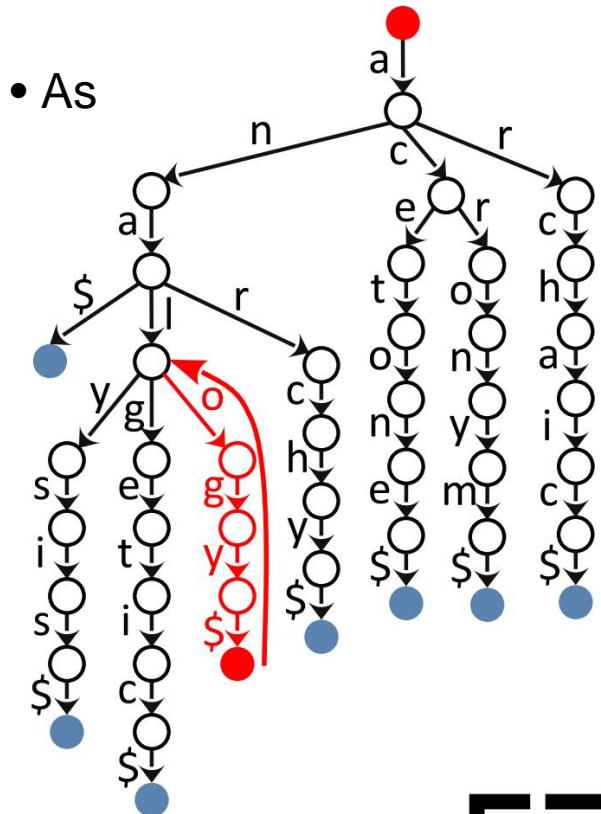
- We return from the leaf to which the search brought us • As long as the parent node has only one child

- Example: `u , we delete = "` \$

```

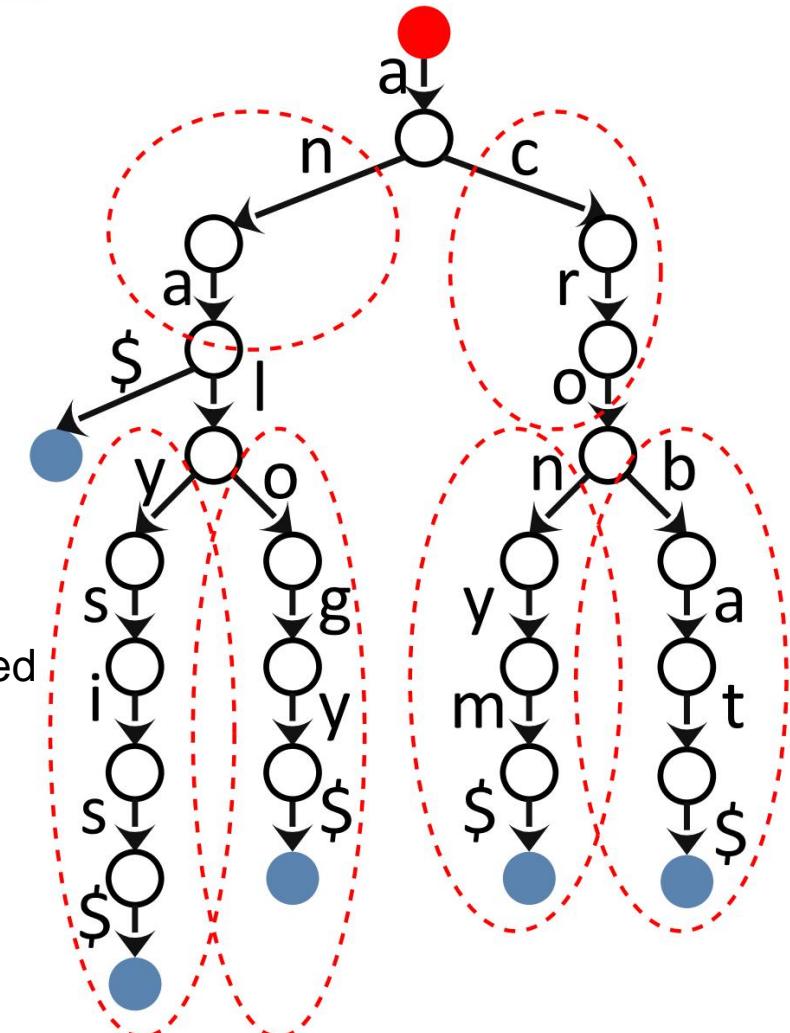
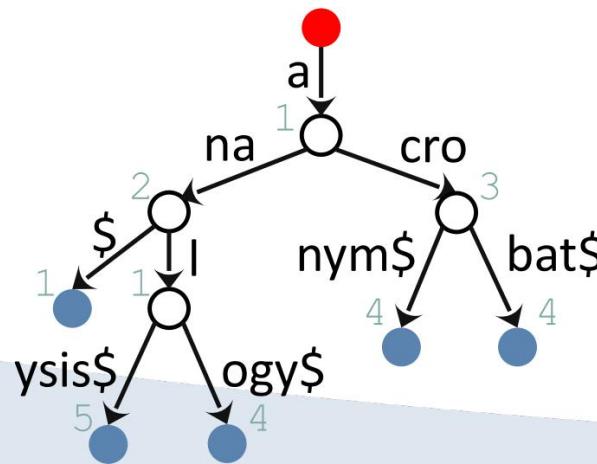
procedure REMOVETRIE( $T, s$ )
  ( $succ, cn$ )  $\leftarrow$  SEARCHTRIE( $T, s$ )
  if not succ then
    return  $\triangleright s$  not found in Trie
   $s \leftarrow \text{reversed}(s)$ 
   $cn \leftarrow \text{parent}(cn)$ 
  for  $ch \in s$  do
    remove transition from  $cn$  for character  $ch$ 
    if  $cn$  has no children and there is a parent of  $cn$  then
       $cn \leftarrow \text{parent}(cn)$ 
    else
      return

```



Patricia tree (1)

- The Trie structure is not compact
 - Remember all fields in nodes where we have only one single pointer
 - The spatial complexity of the Trie can be reduced by turning the sequence of nodes that have only one transition into one transition
 - We get a compressed Trie or Patricia tree
(Practical Algorithm To Retrieve Information Coded In Alphanumeric)



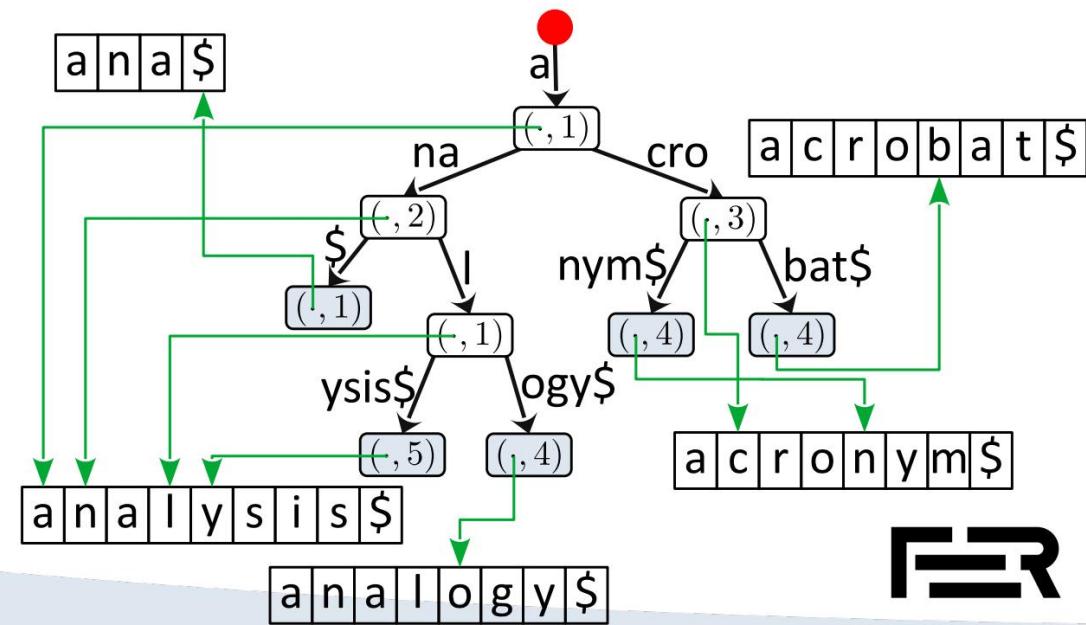
Patricia tree (2)

- Some essential items of the Patricia tree

- All internal nodes of the Patricia tree have at least two children - otherwise we return to the Trie structure
- Character string representation based on ordered pair (,)
it is shown to be better in Patricia trees - less memory consumption
 - We have a set of character strings in memory
 - Nodes point only to specific character substrings in the set

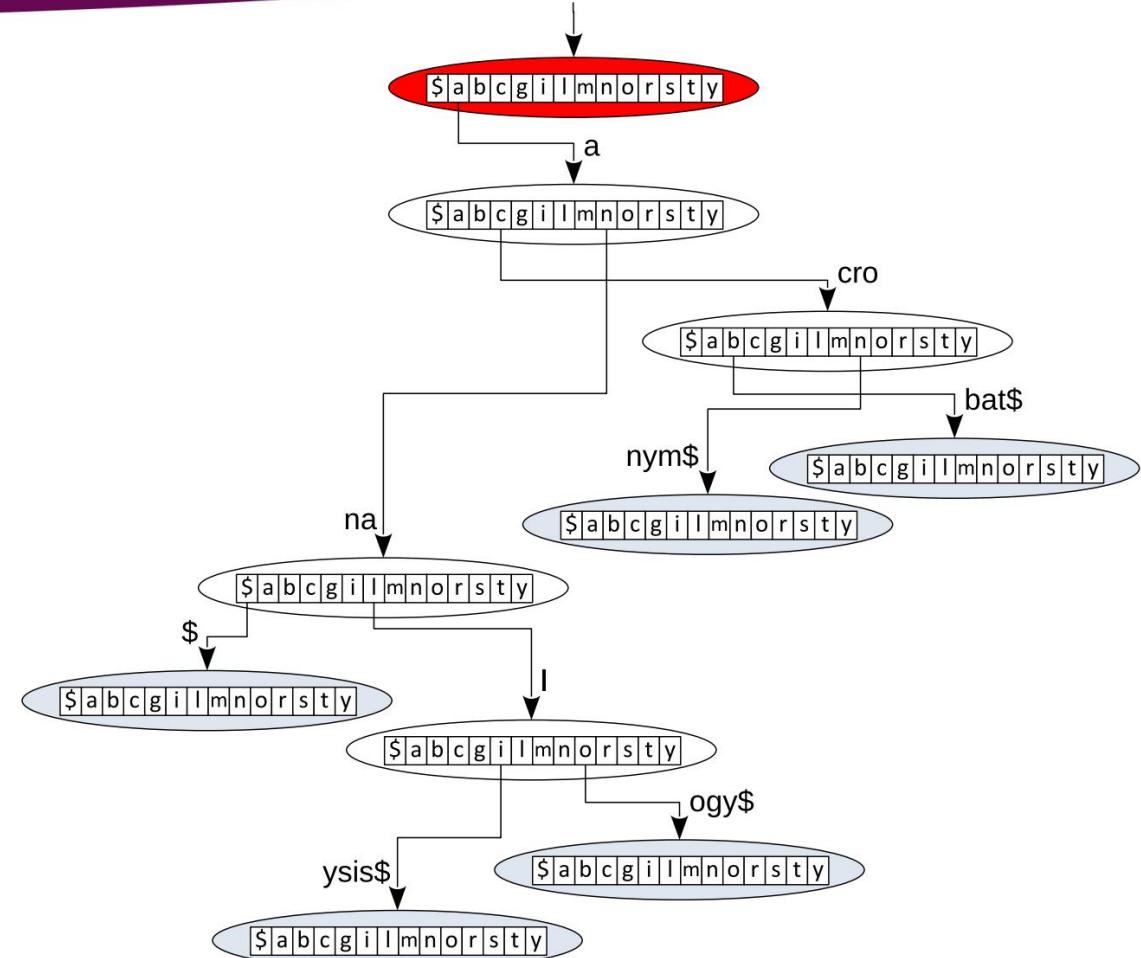
• Let's remember that in each node Patricia trees we have only one entrance edge

- In the node we can store the ordered pair for that input transition



Patricia tree (3)

- Pointers in nodes i
they still function as code
Trie with a little supplement
 - Transitions in the Patricia tree represent a character substring
 - The pointer in the node corresponds to the first character of the character substring of the transition
 - It is not possible for two transitions, two output edges from a node, to have a character substring which starts with the same character
 - They cannot share the same prefix because that would mean that prefix is part of a previous transition



Patricia tree (4)

- The search requires a comparison of the character substring contained in the transition with an adequate character substring of
 - During the search, we need to remember where we are in the character string (variable)

```

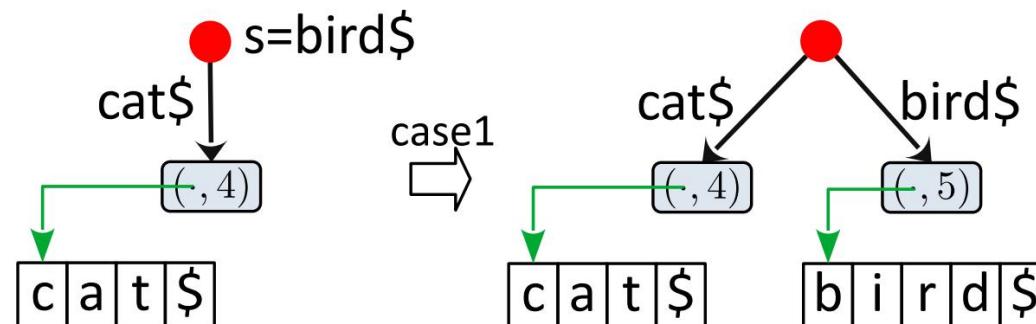
function SEARCHPATRICIA( $P, q = (q_p, q_l)$ )
   $sc \leftarrow 0$ 
   $cn \leftarrow root(P)$ 
  while  $cn$  not leaf do
    if there is transition from  $cn$  for character  $q_p[sc]$  to  $cn_c$  then
       $\triangleright array[q_p[sc]] \neq NULL$  in  $cn$ 
       $(t_p, t_l) \leftarrow$  string representation in  $cn_c$ 
      if  $q_p[sc : sc + t_l] = t_p[0 : t_l]$  then
         $\triangleright$  substring matching (Python notation)
         $sc \leftarrow sc + t_l$ 
         $cn \leftarrow cn_c$ 
      else
        return false
    else
      return false
  return  $sc = q_l$ 

```

Patricia tree (5)

- Entering a new character string is somewhat more complicated than in Trie
 - We start the search
 - We look from the perspective of the current node – we start from the root node

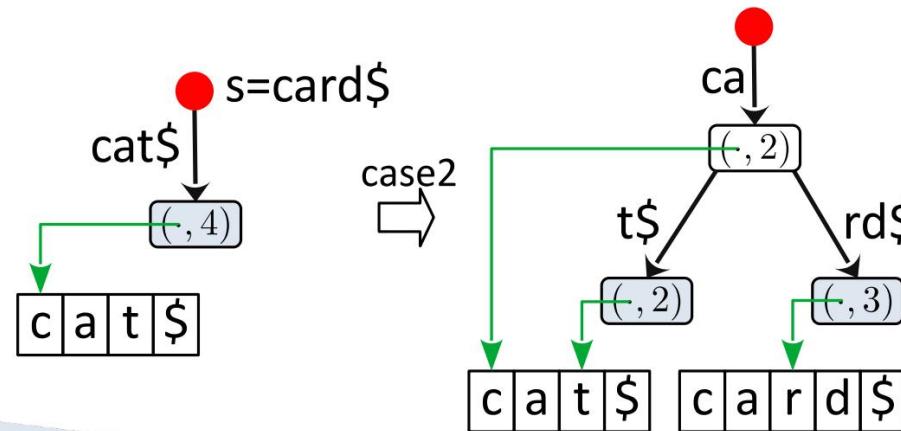
1. It is not possible to find an output transition from the node that would at least partially match the rest of the character string
 - We add a new leaf to the structure and a transition that is identical to the rest of the character string



Patricia tree (6)

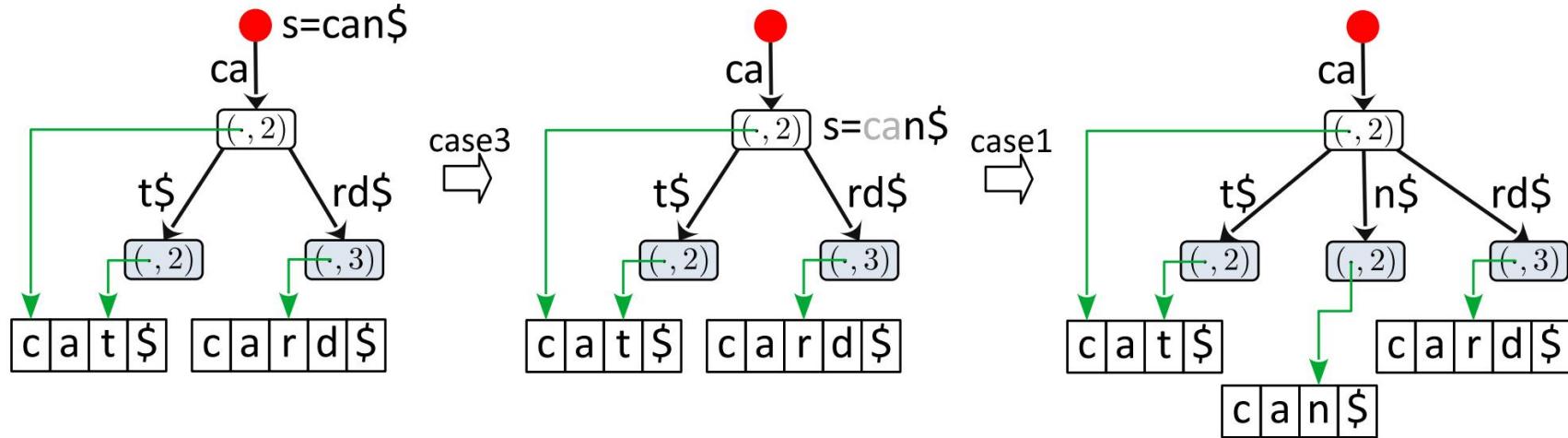
2. It is possible to find an output transition from a node that partially matches part of the rest of the character string \$2! which separates

- We add a new node, the old transition into two parts, the first part that matches the rest of the character string and the second part that does not match
- We add a new leaf and a transition between 3456 that does not respond to the split transition in the previous step



Patricia tree (7)

3. It is possible to find an output transition from a node that completely matches the rest of the character string
- We go through the transition and position ourselves in a new node
 - We pay attention to the rest of the character string



- Evolution of the Patricia tree – the sequence of writing and deleting character strings in : each step can be visualized

Patricia tree (8)

- The complexity of adding a string is $O(|s|)$

```

procedure INSERTPATRICIA( $P, s = (s_p, s_l)$ )
     $sc \leftarrow 0$ 
     $cn \leftarrow root(P)$ 
    while  $cn$  is root or not leaf do  $\triangleright O(|s|)$ 
        if there is transition from  $cn$  for character  $s_p[sc]$  to  $cn_c$  then
             $\triangleright array[s_p[sc]] \neq NULL$  in  $cn$ 
             $(t_p, t_l) \leftarrow$  string representation in  $cn_c$ 
            if  $s_p[sc : sc + t_l] \subset t_p[0 : t_l]$  then  $\triangleright$  case 2
                 $\triangleright$  we know for sure that the first character is matched
                 $i \leftarrow$  first unmatched character from  $t_p$   $\triangleright i > 0$ 
                remove transition between  $cn$  and  $cn_c$ 
                add node  $cn_{ins}$  with  $(t_p, i)$   $\triangleright O(|\Sigma|)$ 
                update node  $cn_c$  with  $(t_p + i, t_l - i)$ 
                add transition between  $cn$  and  $cn_{ins}$ 
                add transition between  $cn_{ins}$  and  $cn_c$ 
                add leaf node  $cn_{leaf}$  with  $(s_p + (sc + i), s_l - (sc + i))$   $\triangleright O(|\Sigma|)$ 
                add transition between  $cn_{ins}$  and  $cn_{leaf}$ 
            return
            else if  $s_p[sc : sc + t_l] = t_p[0 : t_l]$  then  $\triangleright$  case 3
                 $sc \leftarrow sc + t_l$ 
                 $cn \leftarrow cn_c$ 
            else  $\triangleright$  case 1
                add leaf node  $cn_{leaf}$  with  $(s_p + sc, s_l - sc)$   $\triangleright O(|\Sigma|)$ 
                add transition between  $cn$  and  $cn_{leaf}$ 
            return

```

Patricia tree (9)

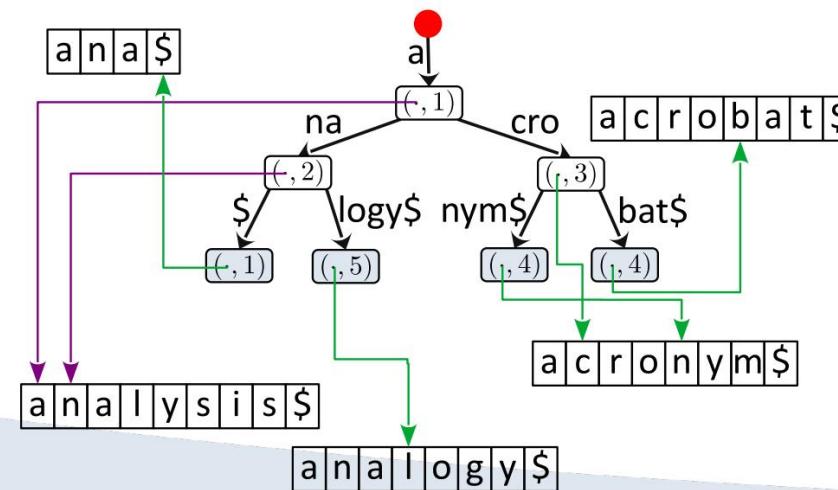
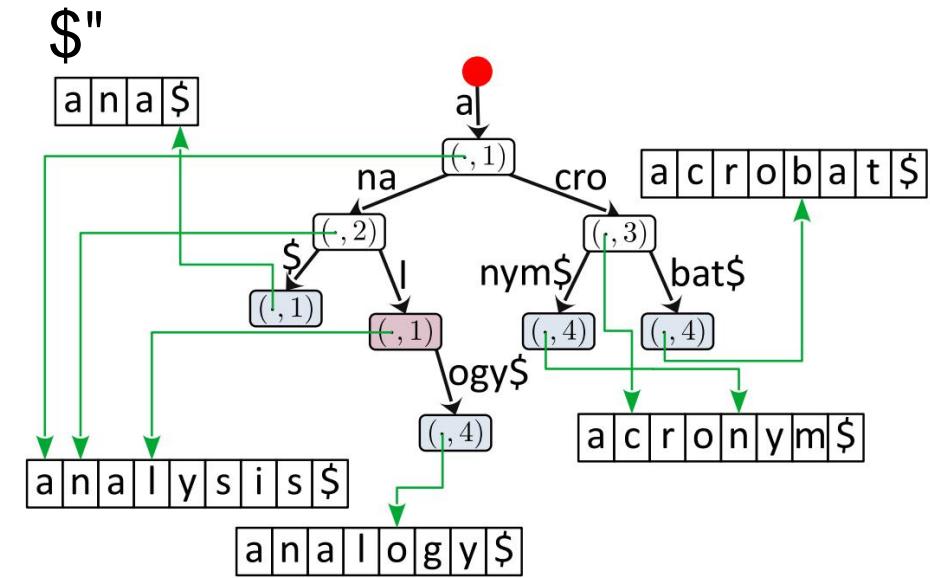
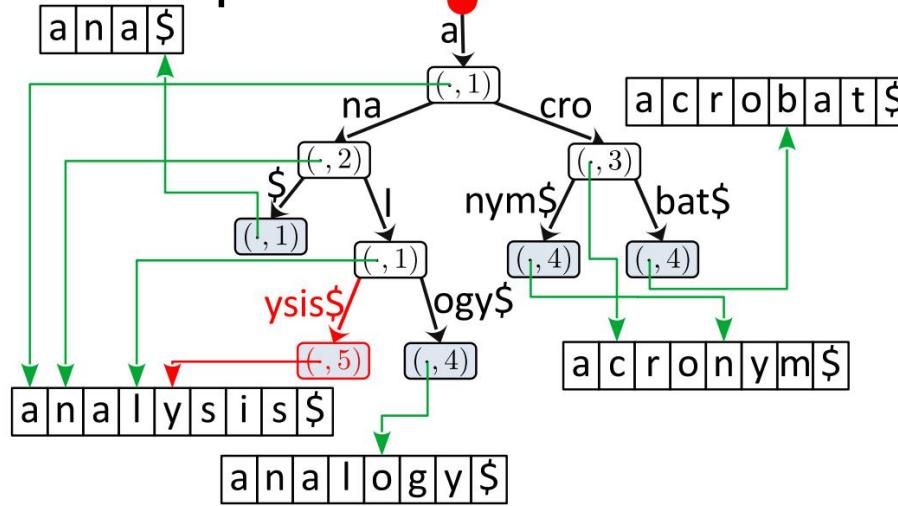
- Deleting a string from a set •

Starts with a search - must exist in the Patricia tree

- We start from the leaf that represents as in Trie
- We remove the leaf and its input transition
- If the parent of the removed leaf remains with one child, then that parent should be removed and its transitions merged - uncompressed case
- Concatenating transitions for character strings (,) is easy
 - Move the pointer to the beginning by a few characters towards the beginning of the character sequence string
 - Increase the length by that number of characters

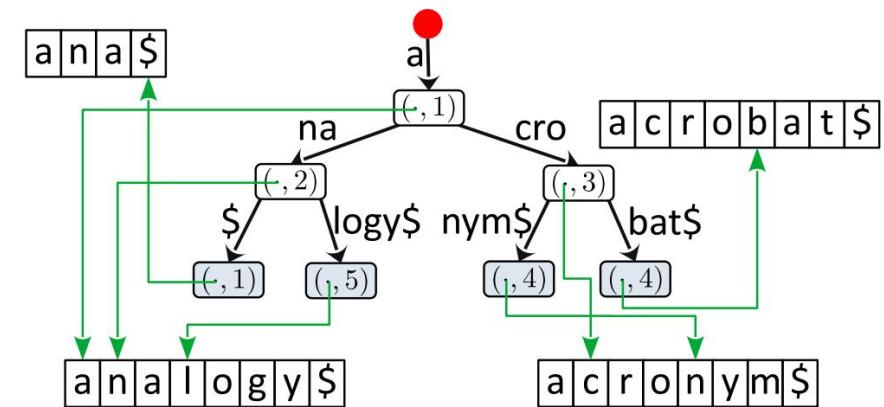
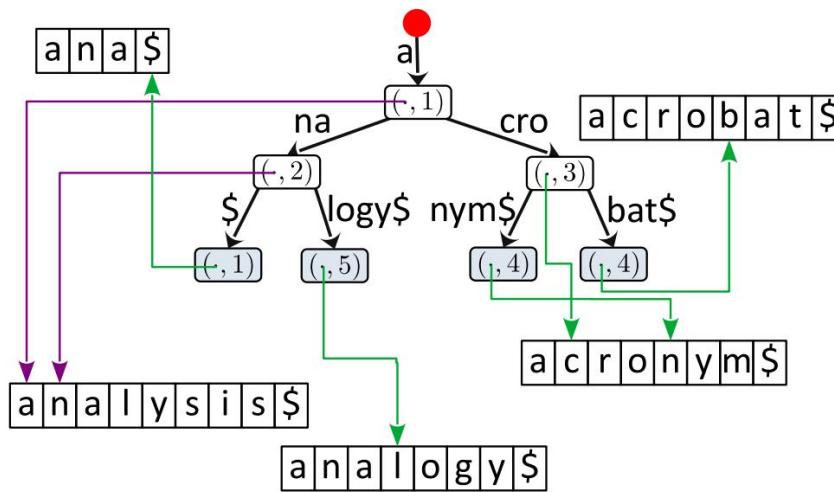
Patricia tree (10)

- Example: we remove = "



Patricia tree (11)

- Character string " \$" we need to remove from memory (destructor in Cu) • However, we have two more pointers to that instance • We transfer the pointers to the character string one of the remaining children
- We transfer the pointers to the character string one of the remaining children

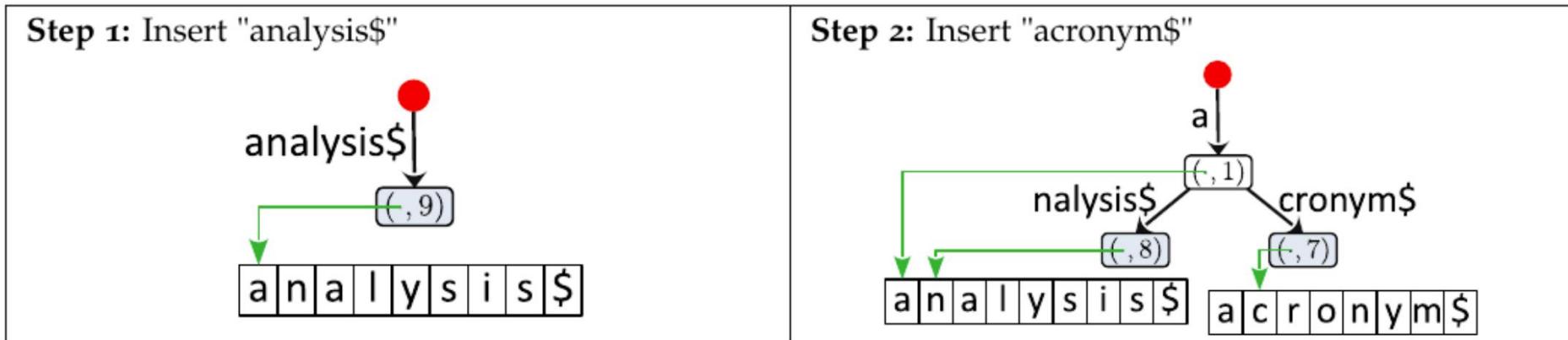


Patricia tree (12)

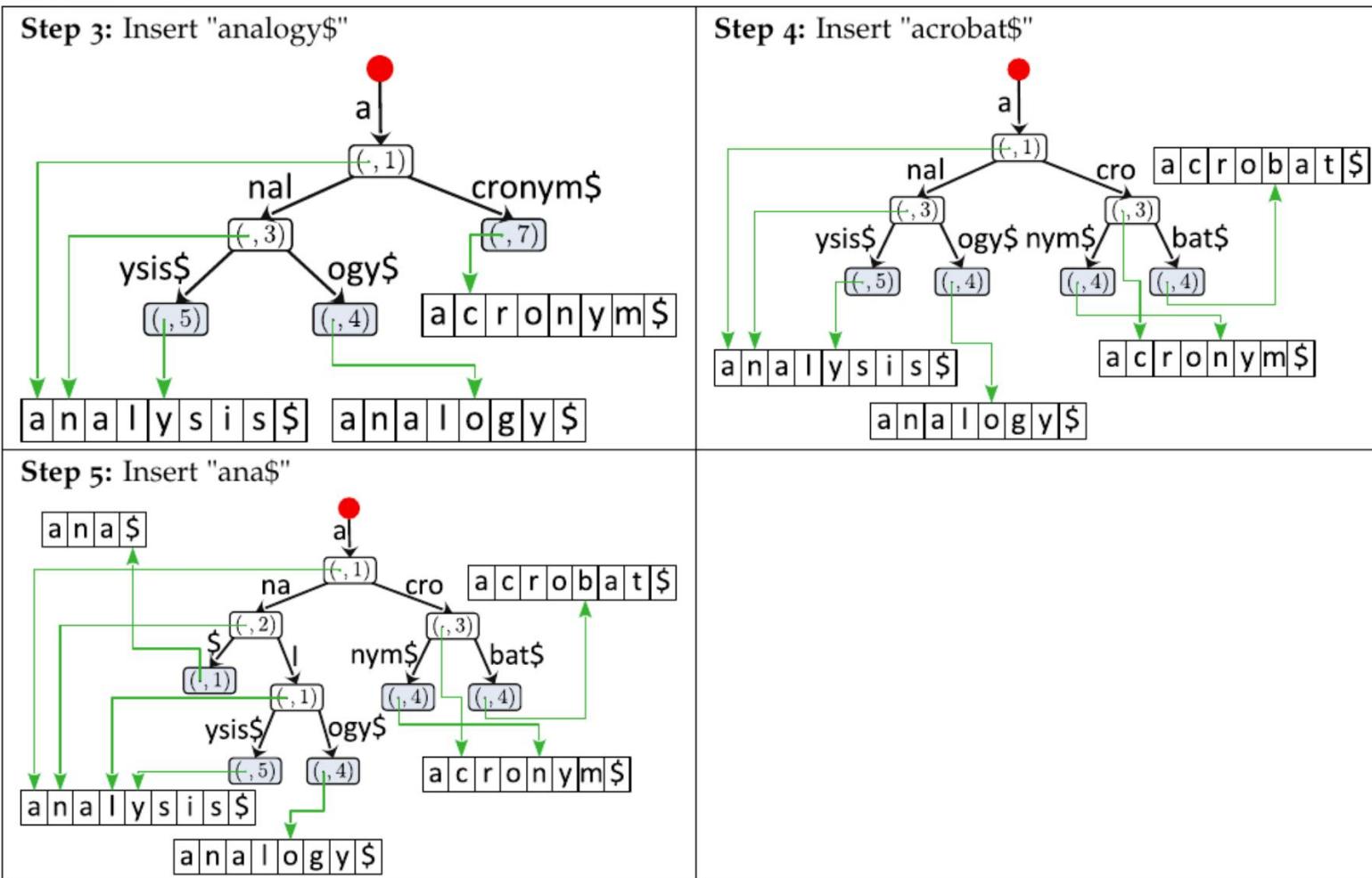
- An example

- Draw the evolution of the Patricia tree to write the next sequence of characters
"of strings

`< $", " $", " $", " $", " $">`



Patricia tree (13)



Patricia tree (14)

- Some significant applications: IP routing (in routers), IP filtering, firewall, IP lookup, ...
- If we take $\hat{y} = 0.1$ for the alphabet { }
- We can transform IPv4 addresses into character strings like

192.168.11.45/32

1100 0000. 1010 1000 . 0000 1011 . 0010 1101

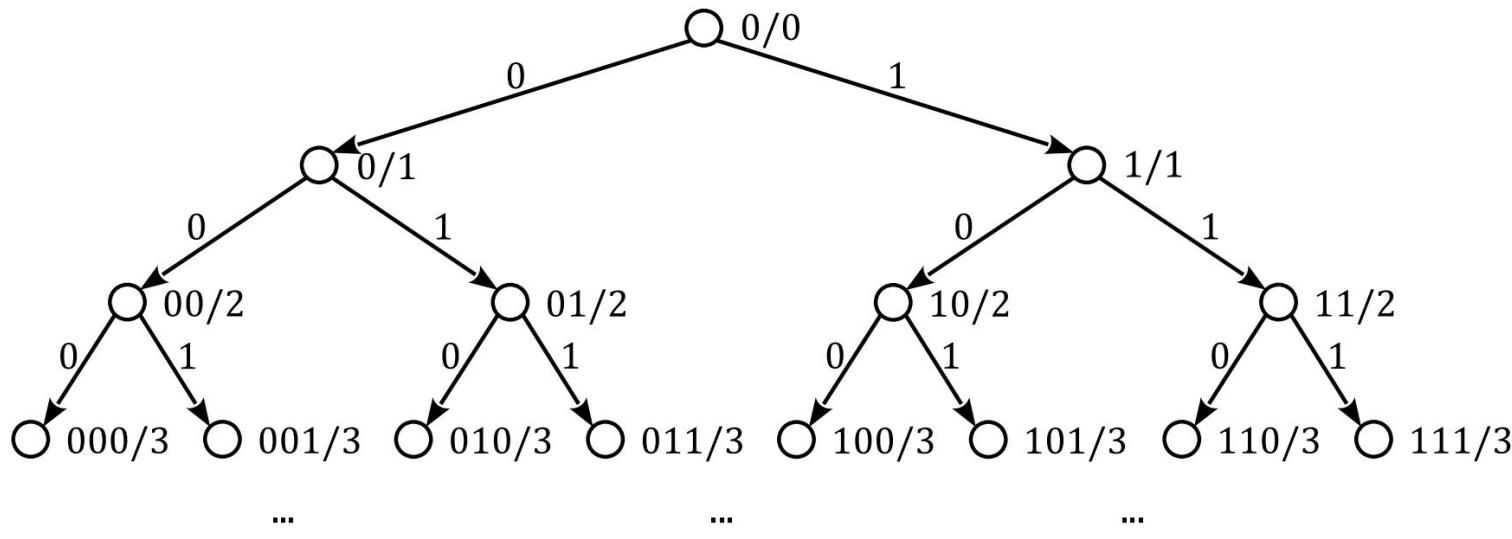
- Or a mask like

192.168.0.0/16

1100 0000. 1010 1000 . * . *

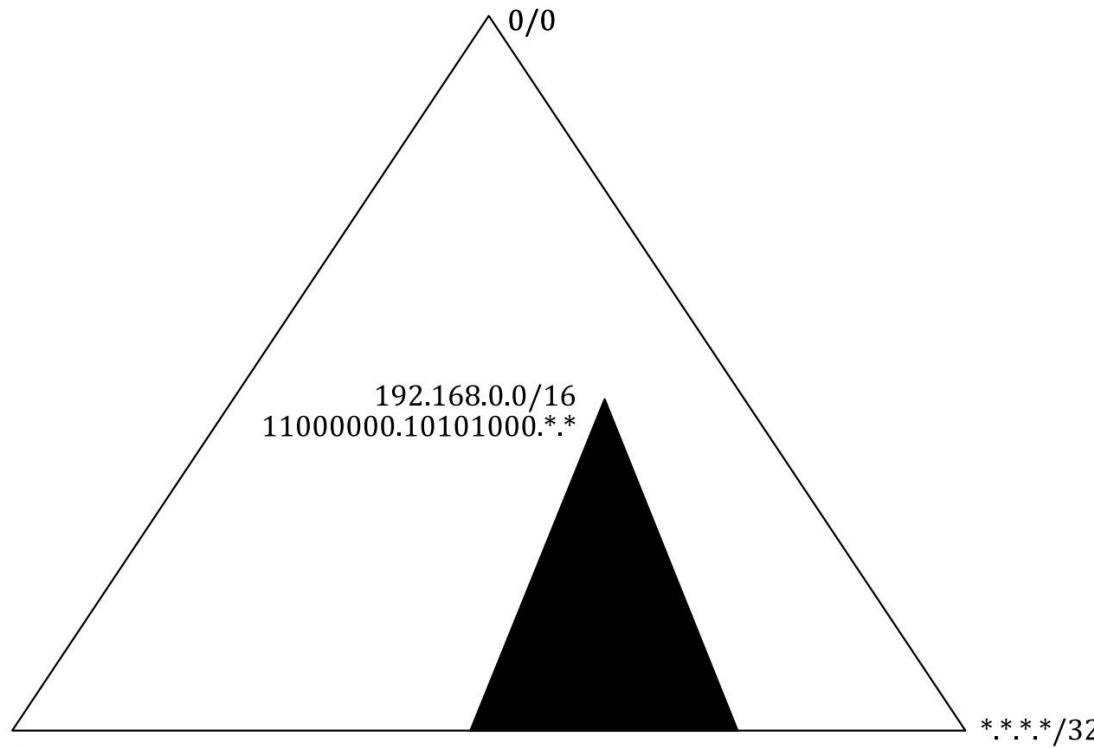
Radix trees (1)

- We build the radix tree for IP routing in the following way - binary tree



Radix trees (2)

- In such a Radix tree, a subnet is detected through its mask • All nodes of the subtree represent members of the subnet
192.168.0.0/16



IP routing

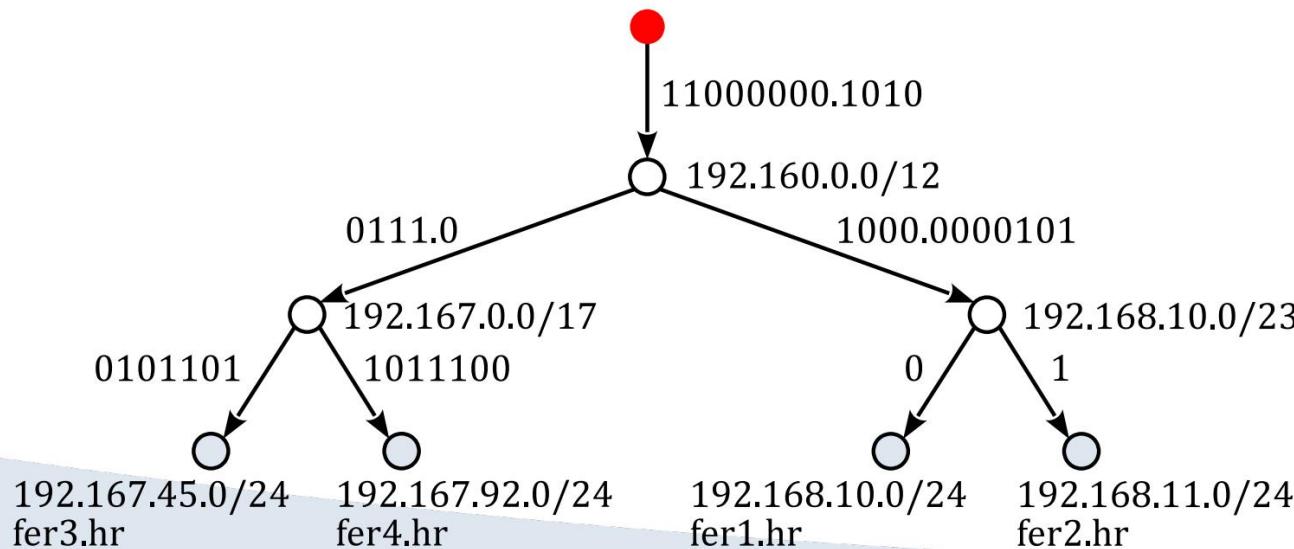
- We compress the Radix tree – we get a binary Patricia tree •

The following IP routing table

192.168.10.0/24	11000000.10101000.00001010	fer1.hr
192.168.11.0/24	11000000.10101000.00001011	fer2.hr
192.167.45.0/24	11000000.10100111.00101101	fer3.hr
192.167.92.0/24	11000000.10100111.01011100	fer4.hr

- turns into a Patricia tree •

leaves can have pointers to an interface definition (or something else)



Suffix fields Suffix trees

String suffix

- Let's imagine the string = " \$"
where $i = 1, [], [2], \dots, [], \dots, []$, and i is the i th character of the character string
- From a character string we can derive a set of all its suffixes, where the i -th suffix of the character string is defined as $[], \dots, [] = [], [+ 1]$

1	2	3	4	5	6	7	8	
p	r	o	g	r	a	m	\$	

1	p	r	o	g	r	a	m	\$
2	r	o	g	r	a	m	\$	
3	o	g	r	a	m	\$		
4	g	r	a	m	\$			
5	r	a	m	\$				
6	a	m	\$					
7	m	\$						
8	\$							

All character string suffixes

- An ordered pair representation of a character string gives us an advantage when creating a list of all character string suffixes
- Naive: if we wanted to create a new instance of the character string for the $i \dots i_n$ suffix [], we should copy the characters from i to n into that new instance.

- For all character string suffixes, this means the next copy number

$$(\dots \hat{i_1} \hat{i_2} \dots \hat{i_{j+1}}) \xrightarrow{\quad \quad \quad (\dots + 1) \quad \quad \quad }$$

- 2 • When we use an ordered pair (,) as a representation of a character string, we work in iterations
- We move the cursor one place forward
 - Let's reduce by one

Suffix field (1)

- We sort the previous list of suffixes alphabetically and get the next sorted list of suffixes (remember $\$=0$)

i	A_i	t_{A_i}						
1	8	\$						
2	6	a	m	\$				
3	4	g	r	a	m	\$		
4	7	m	\$					
5	3	o	g	r	a	m	\$	
6	1	p	r	o	g	r	a	m
7	5	r	a	m	\$			
8	2	r	o	g	r	a	m	\$

- `where = suffix < %, 7, ..., $, ... , 2 >` sorted array of suffixes that define field

Suffix field (2)

- The complexity of creating a suffix field is quite high
- We use a sorting algorithm which is $(n \log n)$, so when we add to that the comparison of character strings, it can be approximated as $(m * n \log n)$
- The idea of the suffix field is to check if the character string is in the character string • A naive search would be $(\frac{n}{m} * m)$

| |

- Suffix field search with binary search algorithm is $(\log n)$
- Binary search algorithm is similar to binary tree search
 - Let's find the middle, then check if it is smaller or larger than that middle
 - Considering the result, we reduce the search to half of the suffix field
 - We continue recursively until the final result:

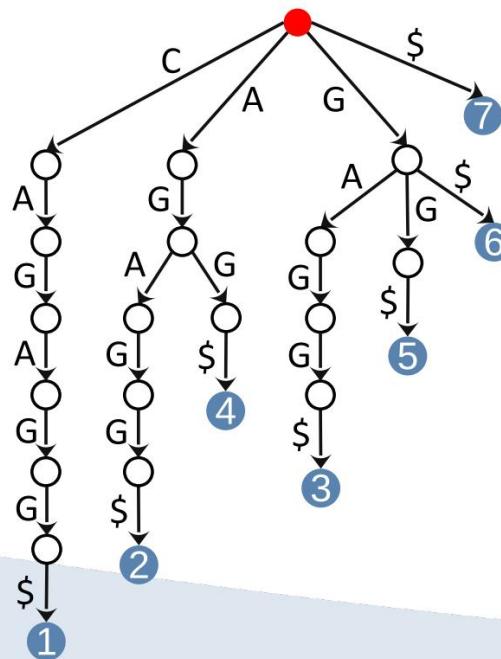
or

Suffix field (3) (info)

- The creation of the suffix field can be linear using special algorithms, such as algorithm skew
- Using the concepts of LCP and $\ddot{\gamma}$ -matrix, the search can be done accelerate to $(+ \log)$
- These algorithms and methods of creation are complex. Students who want to know more about this can find those topics in the NASP script.

Suffix Trie (1)

- We can transform the list of all suffixes of a certain character string in Trie
- For the alphabet $\Sigma = \{C, A, G, T, \$\}$ (DNA bases) and $=$
we have the following list of all suffixes
- Converted to the suffix Trie

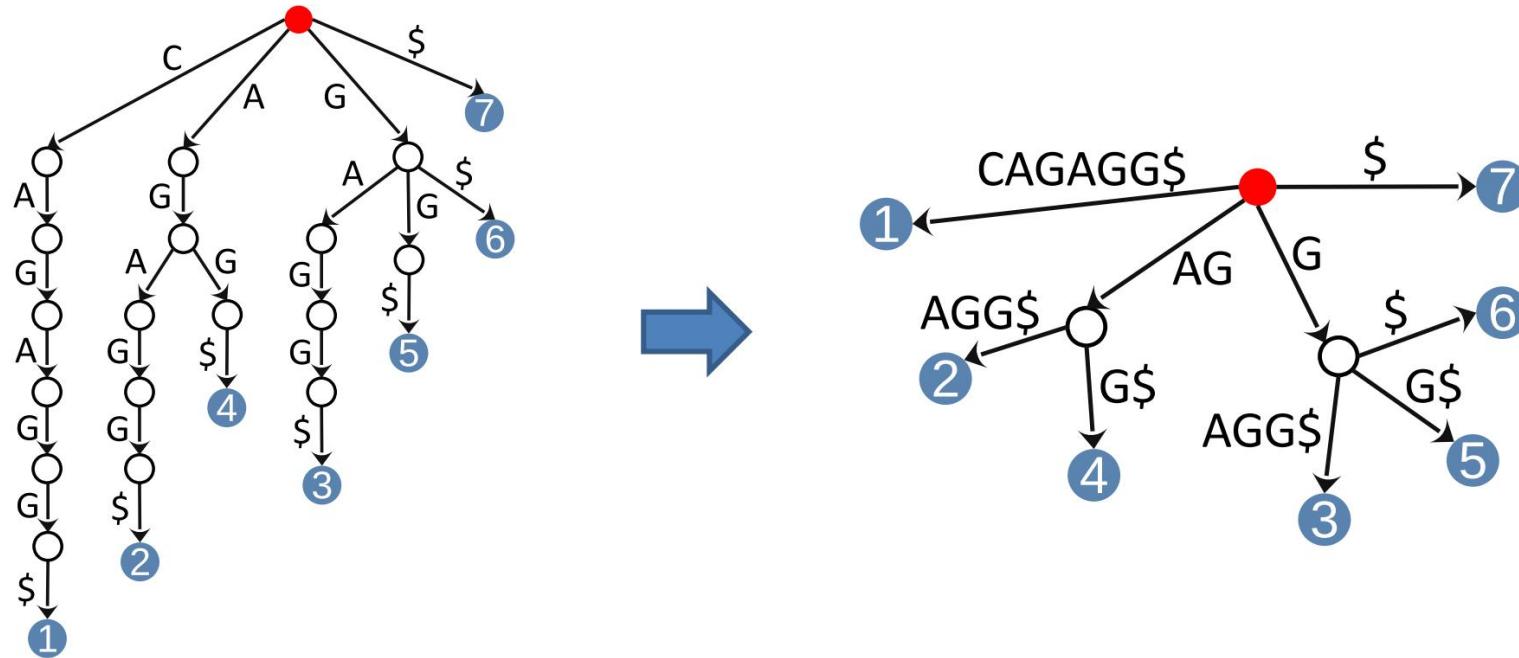


C	A	G	A	G	G	\$
A	G	A	G	G	\$	
G	A	G	G	\$		
A	G	G	\$			
G	G	\$				
G	\$					
\$						

- Each leaf of a suffix Trie contains the sequence number of that suffix or

Suffix tree (1)

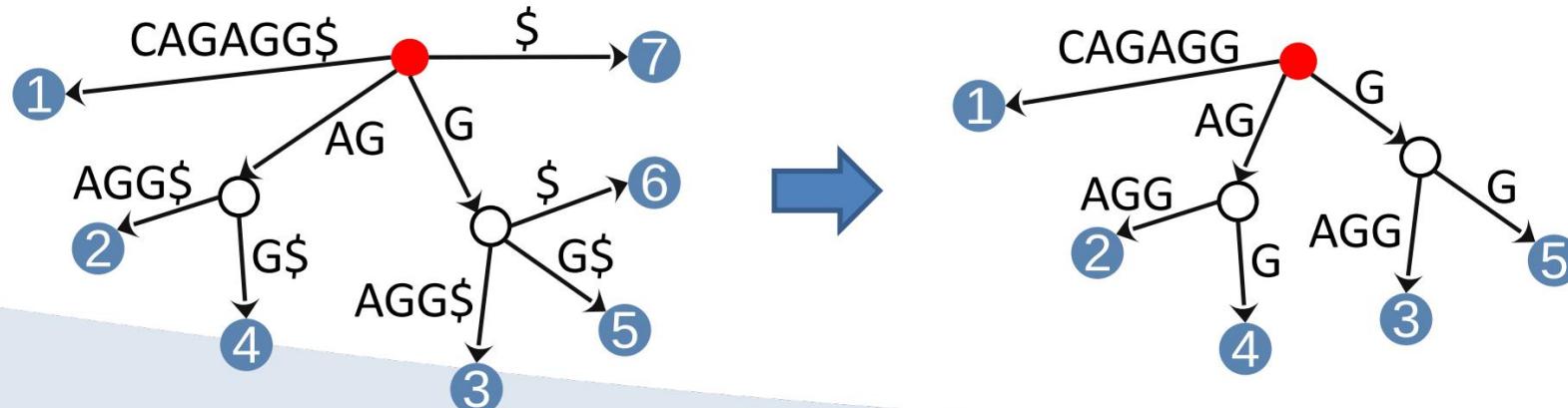
- A suffix tree is a compressed suffix Trie or Patricia tree of all suffixes of a given character string
- This is also called an **explicit suffix tree**



Suffix tree (2)

- The **implicit suffix tree** is obtained from the explicit one by the following transformation:

- Let's remove all NULL terminators (\$) for the edges of the explicit suffix trees
- Let's remove all edges that remain with empty character strings, including their subtrees
- We compress the suffix tree again: all nodes except the root node, which have less than two children are removed, and their transitions are concatenated



Ukkonen's algorithm (1)

- A naive attempt to create a suffix tree is (*)
 - This has been improved several times by various algorithms: McCreigh i Weiner
 - Ukkonen builds his algorithm on Weiner's algorithm
 - Ukkonen's algorithm is an algorithm that builds a suffix tree character by character from online
- We have an implicit suffix tree after the steps we mark !
 - We take the sign [+ 1] and execute the + 1 phase of the tree upgrade
 - + 1 phase is performed in + 1 steps
 - In each step, we add [+ 1] to all prefix suffixes [1,]
 - In + 1 step we add [+ 1] to the root node of the tree
 - The result is an implicit suffix tree

\$8%

Ukkonen's algorithm (2)

- For =
\$ we have the following
phases and steps

- The variable j defines the suffix of the prefix
 $t[1,i]$, or $t[j,i]$

- We traverse the tree for each [,] i
we add [+ 1], and we have three cases:

1. When passing [,] we reach the sheet.
The entire vertical path represents [,].
In the last transition before the leaf,
we just add [+ 1] to the end of the
transition.

Phase 1, $i + 1 = 1, t[i + 1] = A$						
Step 1	A					
Phase 2, $i + 1 = 2, t[i + 1] = B$						
Step 1, $j = 1$	A	B				
Step 2	B					
Phase 3, $i + 1 = 3, t[i + 1] = A$						
Step 1, $j = 1$	A	B	A			
Step 2, $j = 2$	B	A				
Step 3	A					
Phase 4, $i + 1 = 4, t[i + 1] = B$						
Step 1, $j = 1$	A	B	A	B		
Step 2, $j = 2$	B	A	B			
Step 3, $j = 3$	A	B				
Step 4,	B					
...						
Phase 7, $i + 1 = 7, t[i + 1] = C$						
Step 1, $j = 1$	A	B	A	B	A	B
Step 2, $j = 2$	B	A	B	A	B	C
...						
Step 6, $j = 6$	B	C				
Step 7	C					

Ukkonen's algorithm (3)

2. We passed all the characters [,] and stopped in the middle of the transition. The next character in the transition **is not** [+ 1]. We add an internal node to the tree and split the transition. We add a leaf and a transition [+ 1] from the newly added internal node to the newly added leaf. The same operation as with the Patricia tree.
3. We passed all the characters [,] and stopped in the middle of the transition. The next character in the transition **is** [+ 1]. We do nothing because [,] [+ 1] is already contained in the tree.

After we have finished building the implicit suffix tree,
we make it explicit by adding \$ to all transitions leading to the leaves of the tree.

Ukkonen's algorithm (4)

- **Suffix links** are an addition to the implicit suffix tree that enable faster traversal of the tree for [,]

- We have a node % we reached by passing [,]
- If we have another node, 7 is 7 which is reached by passing [,], then connected to % with a suffix connection
- This just means that for all stages \hat{y} we need to start from both nodes, which effectively skips all steps involving the substring
 - This means that we do not have to start $sj=1$, but immediately after
- 7 the substring there can be a root node when $= i =$

%

Example Ukkonen (1)

- We create an implicit suffix tree using Ukkonen's algorithm for = \$

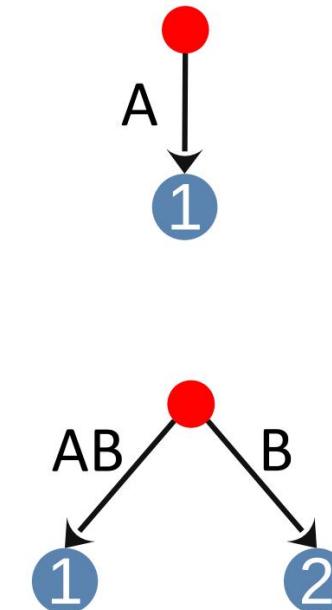
- Phase 1 : + 1 = 1, [+ 1] =**

- Step 1 : We only have the root node.
We add a new transition and leaf.

- Stage 2 : + 1 = 2, [+ 1] = •**

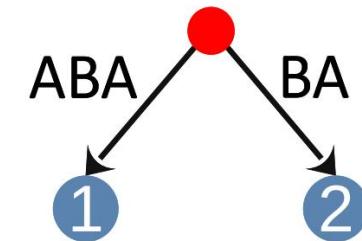
Step 1 : = 1, [,] = leads to ^{The passing of us} leaf 1, which is case 1. We add B to the last transition.

- Step 2: [,] = to root . We are adding a new transition node and associated leaf.



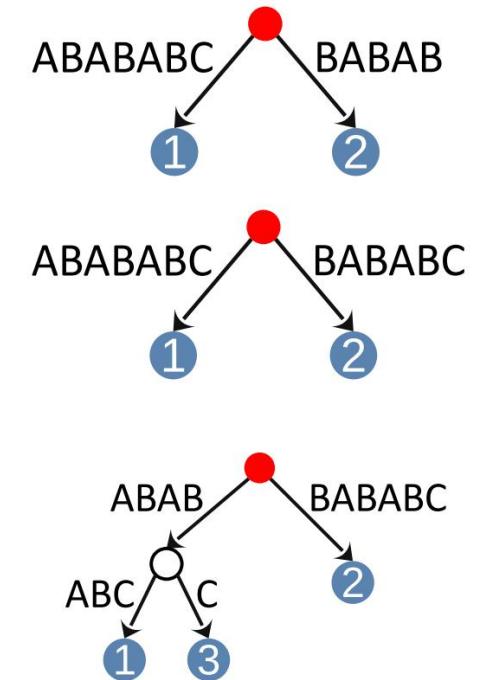
Example Ukkonen (2)

- **Phase 3 : + 1 = 3, [+ 1] =**
 - Step 1 : = 1, [,] = . The traversal brings us to leaf 1, which is case 1. We add A to the last transition.
 - Step 2 : = 2, [,] = leads . The passing of us to leaf 2, which is case 1. We add A to the last transition.
 - Step 3: [,] = . Considering , we remain in the root node, but passing [+ 1] leads us to transition, which is case 3. We do nothing.



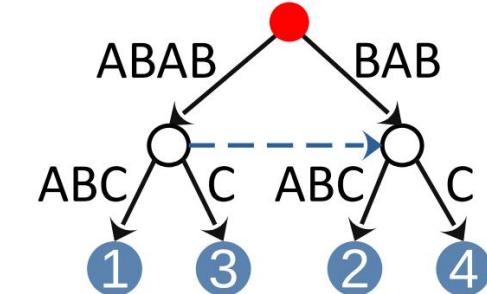
Example Ukkonen (3)

- We skip a series of the same stages
- **Stage 7 : + 1 = 7, [+ 1] = .** The traversal brings us to sheet 1, which is case 1. We add C to the last transition.
- Step 1 := 1, [,] = . Traversal brings us to sheet 2, which is case 1. We add C to the last transition.
- Step 2 := 2, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for

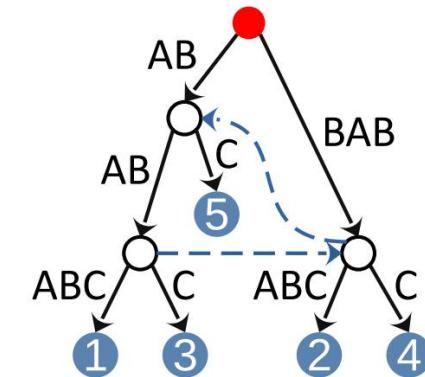


Example Ukkonen (4)

- Step 4 : = 4, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for and . In this step, ~~We have to cut the internal nodes~~ link.
- $\% = \quad \quad \quad 7 = \quad \quad \quad \cdot$



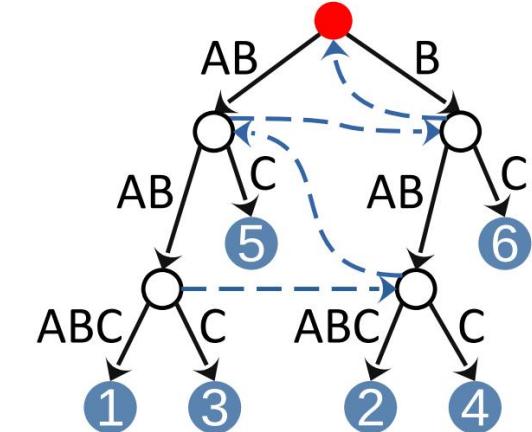
- Step 5 : = 5, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for { , , } and . In this step we internal nodes = i . We connect with a suffix connection.
- $\% = \quad \quad \quad 7 = \quad \quad \quad \cdot$



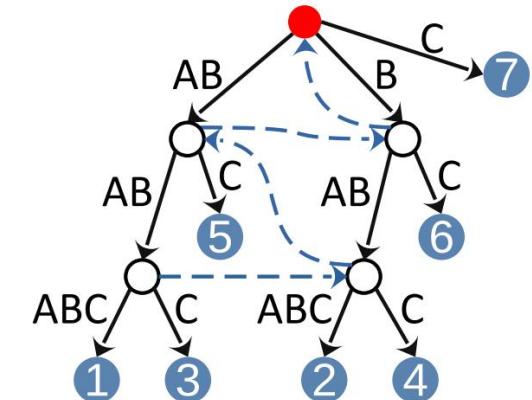
Example Ukkonen (5)

- Step 6 : = 6, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for . { , } . In this step we have two internal nodes = i . We connect them with a suffix link. We also add a suffix link between the = node and the root node.

7

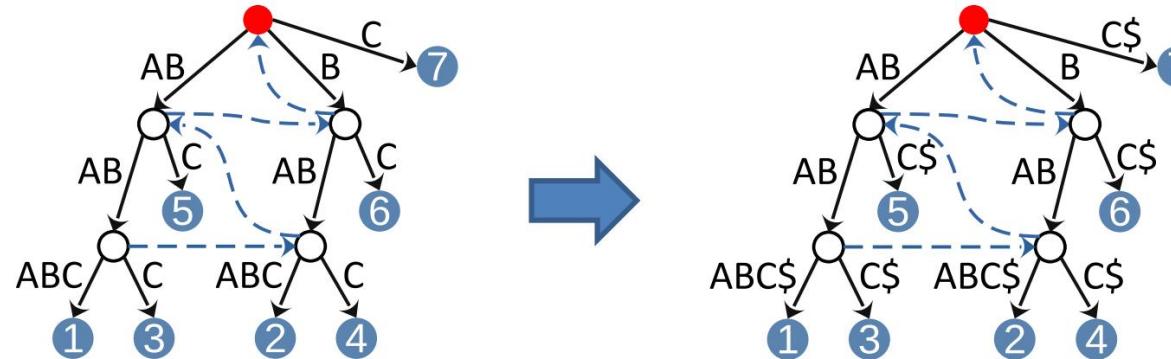


- Step 7: [,] = . Since no transition starts with C, we add a new sheet and a new transition.



Ukkonen's algorithm (5)

- We can now transform the implicit suffix tree into an explicit one



- Let's say that we now add the eighth character D (+

$1 = 8)$

- At the moment when we pass the prefix $[1,7] =$ to , we come across the suffix connections

- At • we have suffix links to 4 additional nodes

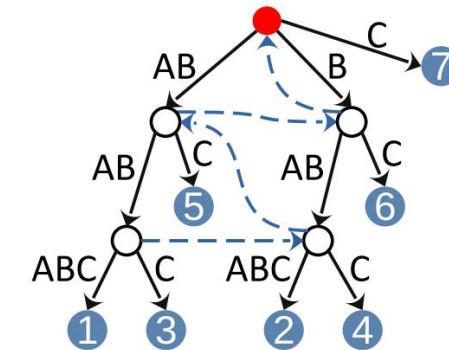
At transition we have 5 nodes from which we start with transitions

- We know that we can skip to $[6,7]$ because we had the , , suffixes in passing

Ukkonen's algorithm (6)

- At $t[6,7]$ we start from the beginning because we do not have a root node in our set of nodes

$t[1,7]$	A	B	A	B	A	B	C
$t[2,7]$		B	A	B	A	B	C
$t[3,7]$			A	B	A	B	C
$t[4,7]$				B	A	B	C
$t[5,7]$					A	B	C
$t[6,7]$						B	C
$t[7,7]$							C



- This reduces both the number of steps and the number of characters passed in a step
 - Reducing the number of steps - we actually parallelize the execution
 - The real gain is in reducing the number of characters passed in a certain step
 - In the previous example, we omitted crossing 6 characters

Pitanja ?