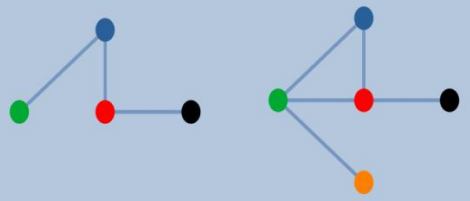
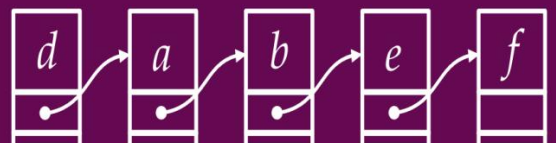
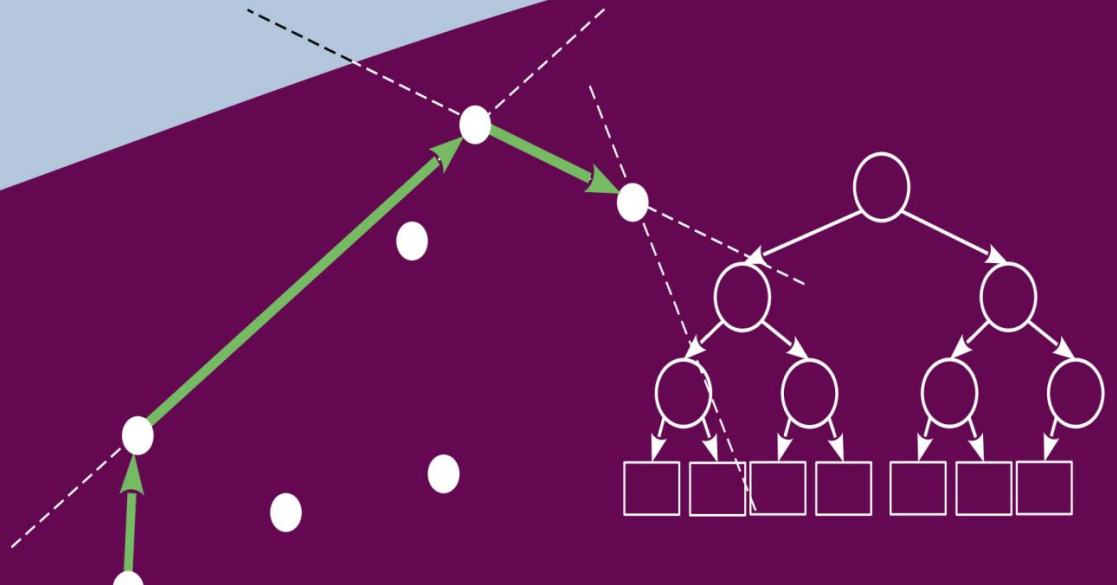
Advanced algorithms and data structures

Week 3: Data structures for characters strings

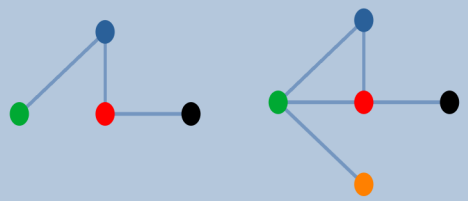


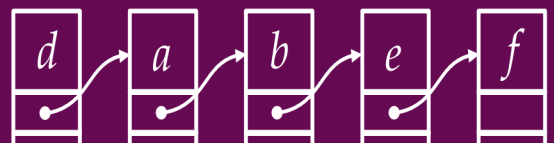


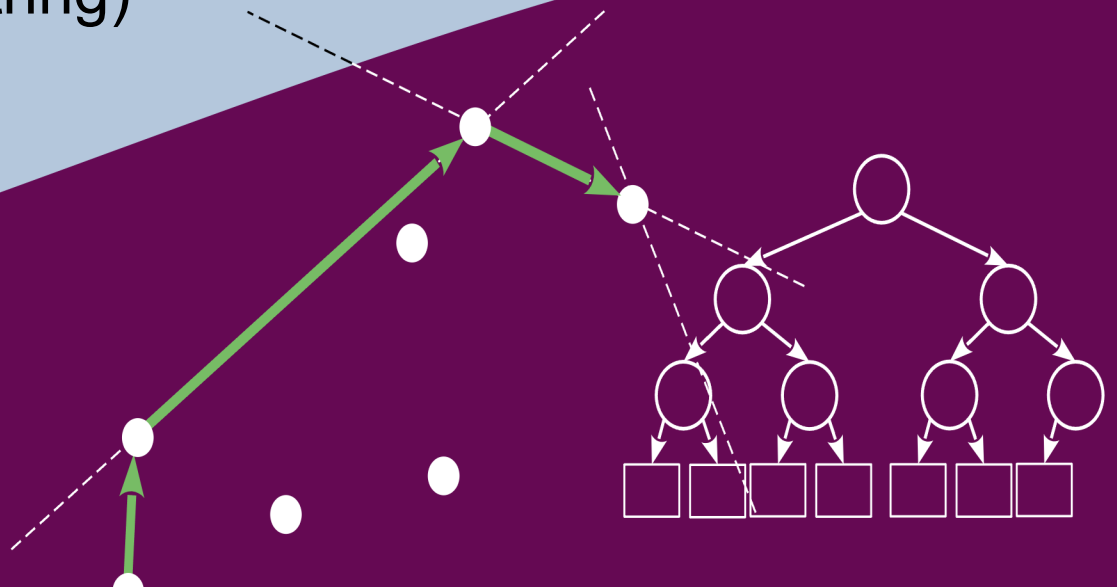


Napredni algoritmi i strukture podataka

Tjedan 3: Strukture podataka za znakovne nizove (string)







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.

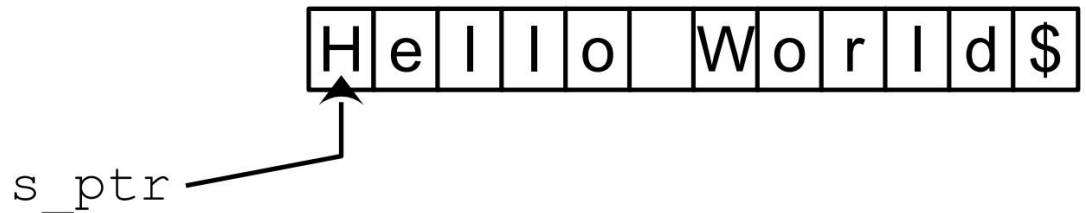
Creative Commons

- slobodno smijete:
  - dijeliti — umnožavati, distribuirati i javnosti priopćavati djelo
  - prerađivati djelo
- pod sljedećim uvjetima:
  - imenovanje: morate priznati i označiti autorstvo djela na način kako je specificirao autor ili davatelj licence (ali ne način koji bi sugerirao da Vi ili Vaše korištenje njegova djela imate njegovu izravnu podršku).
  - nekomercijalno: ovo djelo ne smijete koristiti u komercijalne svrhe.
  - dijeli pod istim uvjetima: ako ovo djelo izmijenite, preoblikujete ili stvarate koristeći ga, preradu možete distribuirati samo pod licencom koja je ista ili slična ovoj.

U slučaju daljnjeg korištenja ili distribuiranja morate drugima jasno dati do znanja licencne uvjete ovog djela.
Od svakog od gornjih uvjeta moguće je odstupiti, ako dobijete dopuštenje nositelja autorskog prava.
Ništa u ovoj licenci ne narušava ili ograničava autorova moralna prava. *Tekst licence preuzet je s http://creativecommons.org/*

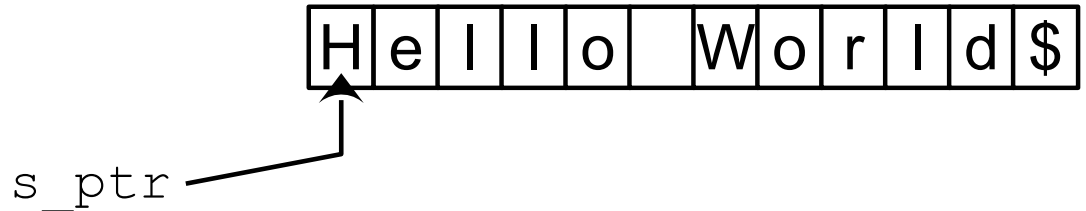
Character string representation (1)

- A string of characters is characterized by its alphabet ÿ - it defines all the characters that can be found in the string
  - We have the mapping : ÿ ÿ ÿ, 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 $



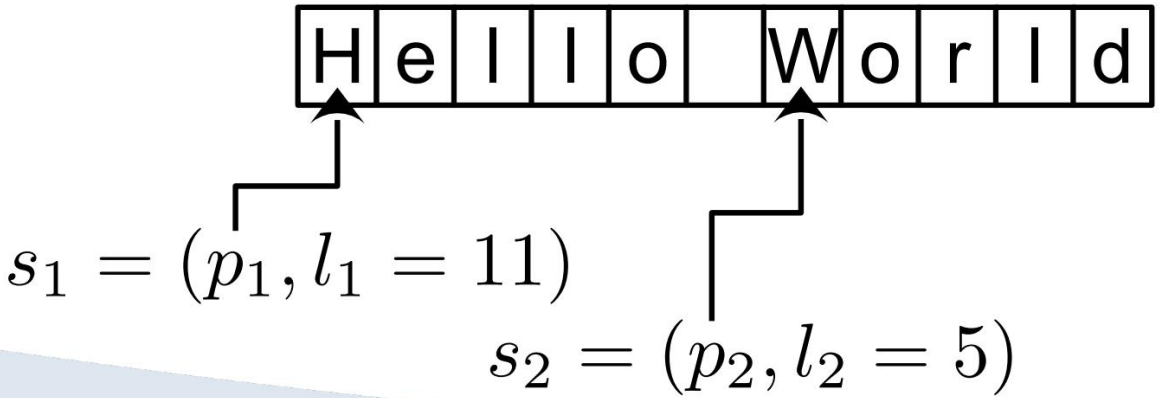
Reprezentacija znakovnog niza (1)

- Znakovni niz je okarakteriziran svojim alfabetom Σ – definira sve znakove koji se u nizu mogu naći
  - Imamo mapiranje $\mu: \Sigma \rightarrow \mathbb{N}$, kojim se znakovima alfabeta pridodaje cijeli broj
  - Tako definiramo znakovne tablice, poput ASCII (1 bajt – 8 bitova) ili Unicode (2 bajta – 16 bitova)
- Osnovna reprezentacija – korištenjem ASCII tablice znakovni niz definiramo kao slijed bajtova koji je završen s NULL terminatorom (vrijednost 0) – a koji u predavanju označavamo kao $



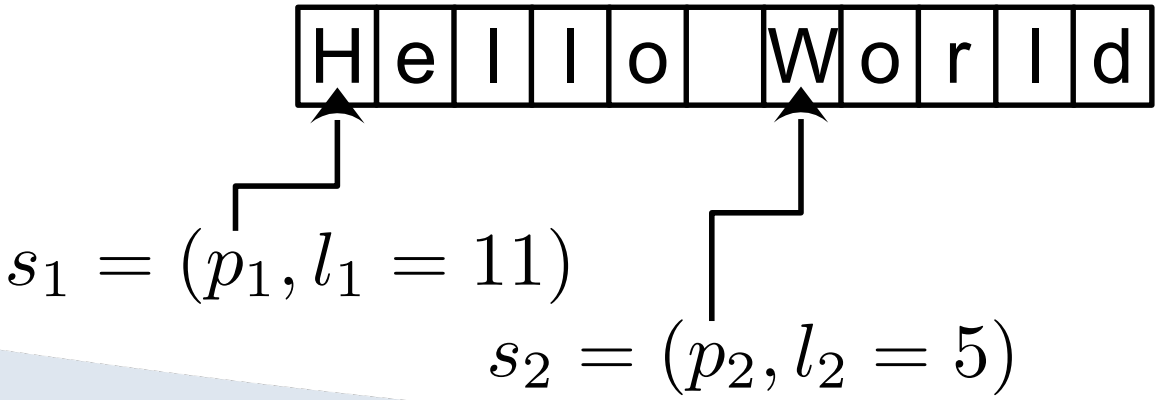
Character string representation (2)

- Representation with an arranged pair, ( ), 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



Reprezentacija znakovnog niza (2)

- Reprezentacija uređenim parom (p, l), gdje je
  - p – pokazivač na prvi znak znakovnog niza
  - l – duljina niza
- Ovakva reprezentacija je zanimljiva kada na jednostavan način trebamo izolirati znakovni podniz bez stvaranja nove instance
  - Dobro rješenje kada imamo veći tekst u memoriji, pa u njemu treba mapirati pojedine dijelove
  - Za ovakvu reprezentaciju nije nužno korištenje NULL terminatora



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

$$= !:\{0 < ÿ \bullet \text{where} \}$$

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 ÿ • A naive implementation would loop through all the characters of the character strings in the set S, resulting in s

$$(-!!ÿ\#\ \$^{|})^{|}$$

Prefiksno stablo (Trie) (1)

- Zamislimo veći tekst u memoriji računala, te skup ključnih riječi (ili izraza) od interesa u tom tekstu
- Skup ključnih znakovnih nizova (riječi, izrazi, dijelovi teksta) definiramo kao

$$S = \{s_i : 0 < i \leq N\}$$

  - gdje je N broj znakovnih nizova
- Želimo znati da li je znakovni niz q sadržan u tekstu
  - To možemo provjeriti tako da testiramo hipotezu $q \in S$
  - Naivna implementacija bi prolazila kroz sve znakove znakovnih nizova u skupu S, što rezultira s

$$O(\sum_{s_i \in S} |s_i|)$$

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 ÿ 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

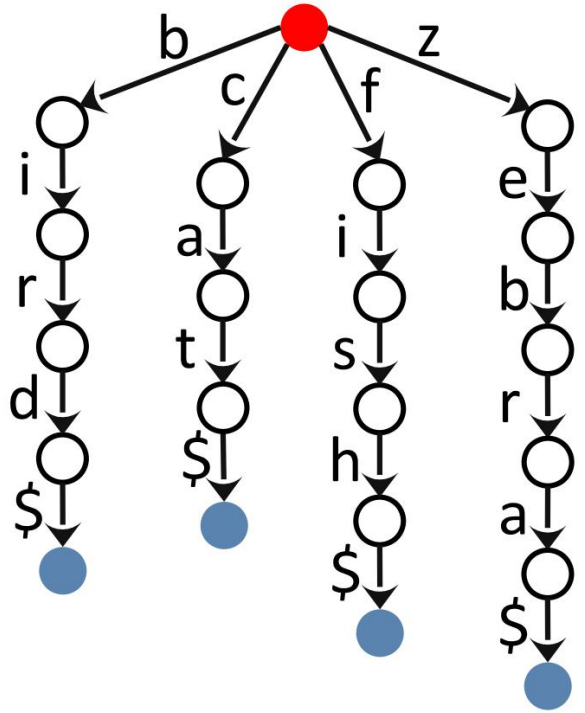
Prefiksno stablo (Trie) (2)

- Trie je uređena trojka $T = (N, E, \mu)$ koja predstavlja m-stablo
  - Funkcija mapiranja $\mu: E \rightarrow \Sigma$ mapira bridove stabla na alfabet
  - Korijenski čvor predstavlja prazni znakovni niz
  - Svaki čvor Trie-a može imati najviše $|\Sigma|$ djece
    - Svaki izlazni brid čvora mora se mapirati na drugi znak alfabeta – dva različita izlazna brida jednog čvora ne mogu biti mapirani na isti znak alfabeta
  - Listovi su terminirajući čvorovi čiji su ulazni bridovi mapirani na $ (NULL terminator)
  - Trie je zapravo konačni deterministički automat koji omogućava parsiranje znakovnog niza

Prefix Tree (Trie) (3)

- An example

% = { " $”” $”,” ÿ$”,” , ÿ, , $"}
ÿ = $(, , , , , , , , }



- 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
    (|ÿ ÿ) | |
  - Is there a better way than that?

Prefiksno stablo (Trie) (3)

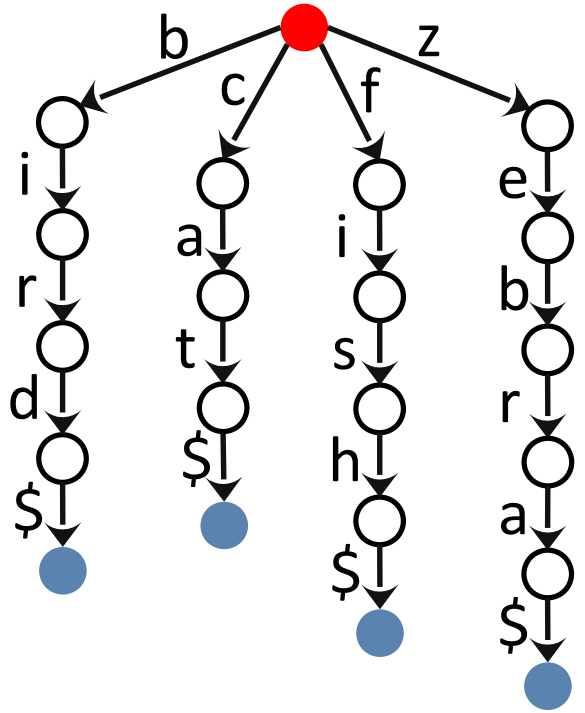
- Primjer

$$S_1 = \{"bird\$", "cat\$", "fish\$", "zebra\$"\}$$
$$\Sigma = \{\$, a, b, c, d, e, f, h, i, r, s, t, z\}$$

- Problem u ovakvom stablu je pretraživanje izlaznih bridova
  - Kako ustanoviti da li imamo izlazni brid koji je istovjetan našem sljedećem znaku u nizu q
  - Naivna implementacija bi podrazumijevala iteraciju po svim izlaznim bridovima

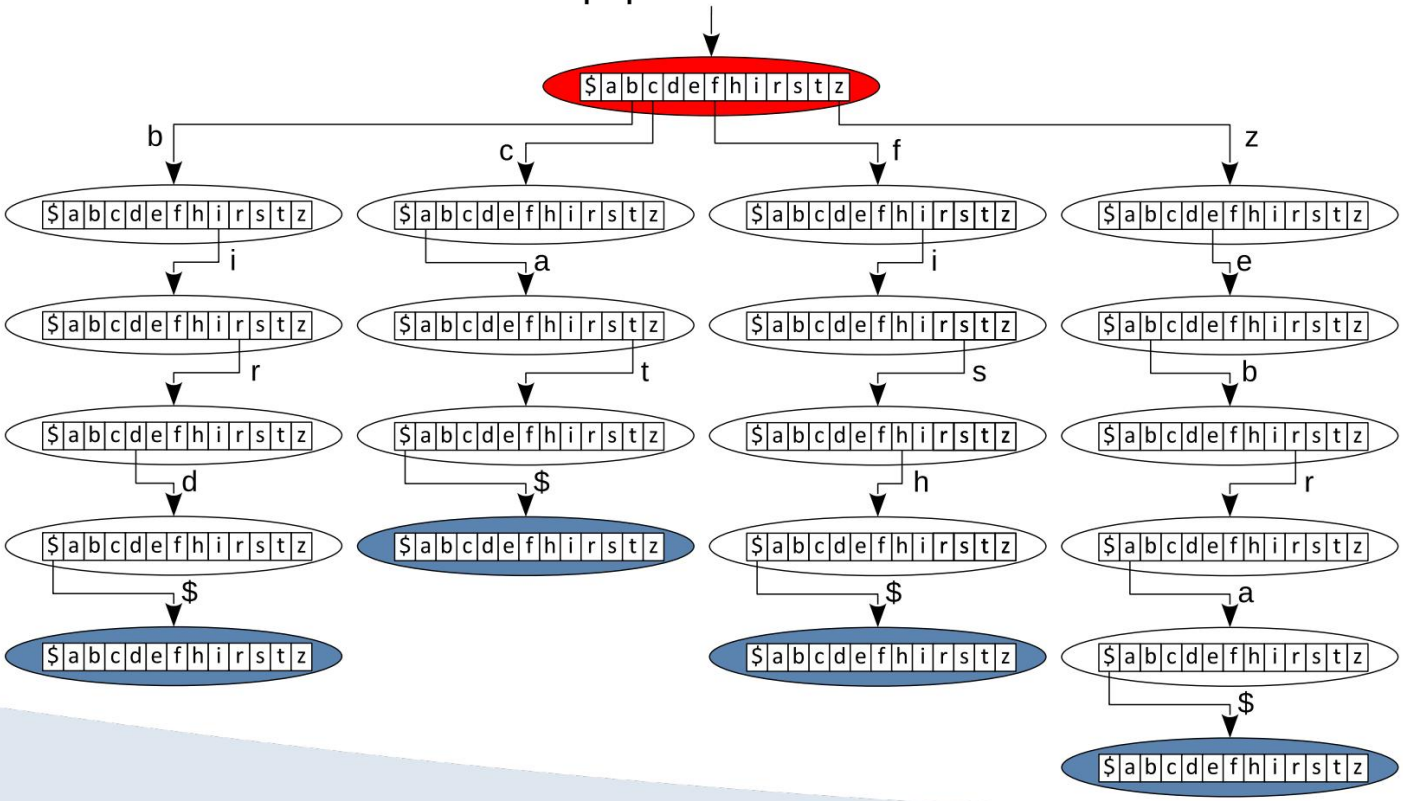
$$O(|q| * |\Sigma|)$$

  - Postoji li bolji način od toga?



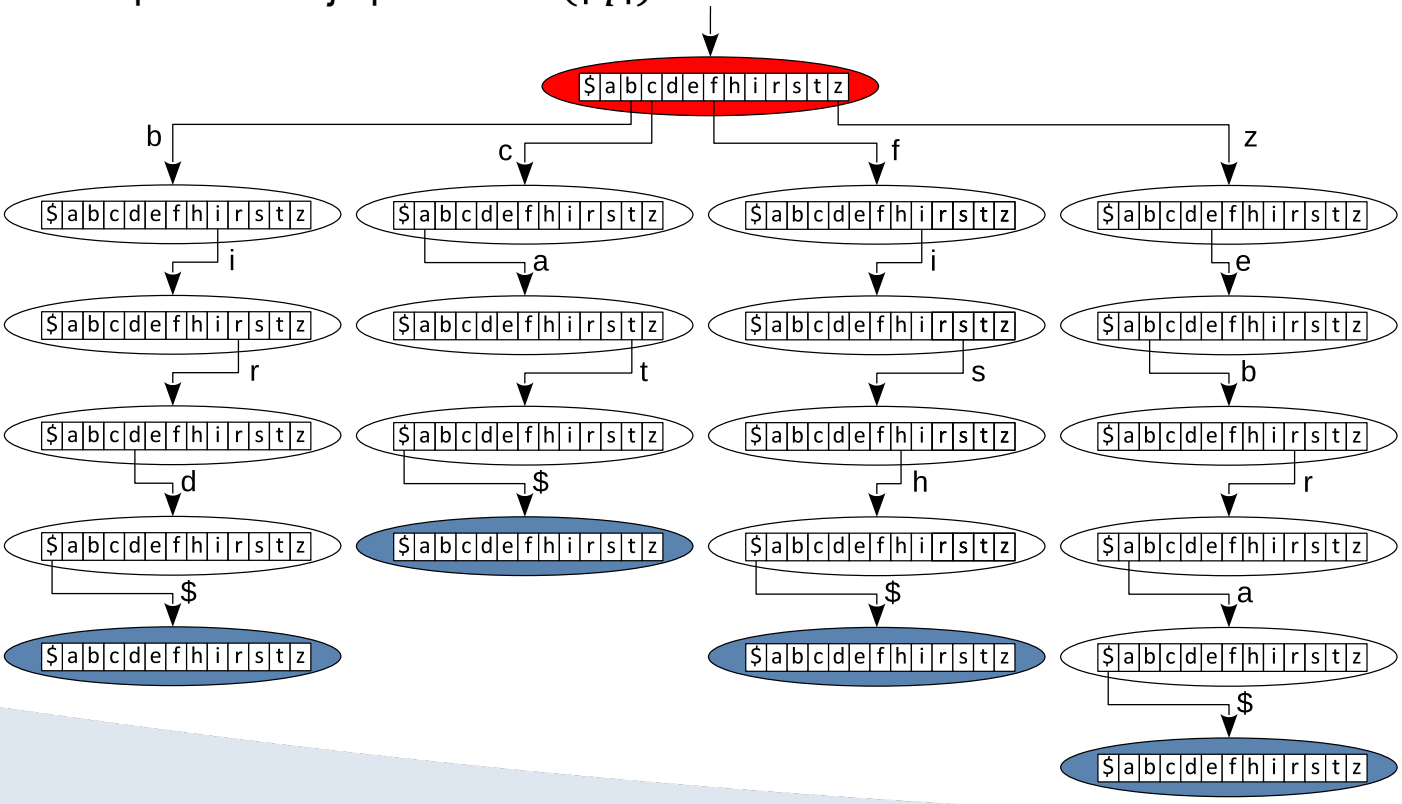
Prefix Tree (Trie) (4)

- In each node we put an alphabet mapping field with pointers to children
  - From the mapping : ÿ ÿ ÿ 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 (ÿ!ÿ# $ ÿ ÿ), • But the search complexity decreases to ( )



Prefiksno stablo (Trie) (4)

- U svaki čvor stavimo mapirajuće polje alfabeta s pokazivačima na djecu
  - Iz mapiranja $\mu: \Sigma \rightarrow \mathbb{N}$ znamo koji element polja gledamo
  - Ako u tom elementu ima pokazivač, tada ta tranzicija postoji
  - Prostorna kompleksnost se povećava na $O(\sum_{s_i \in S} |s_i| * |\Sigma|)$,
  - No složenost pretraživanja pada na $O(|q|)$

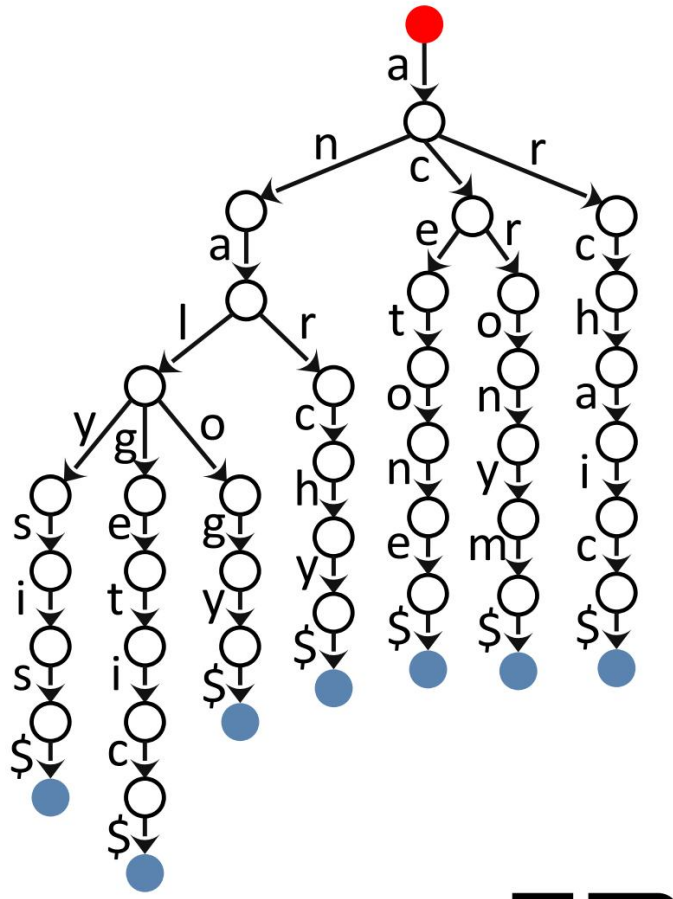


Prefix Tree (Trie) (5)

- Many character strings in can share a common prefix

! = { " $”,” $”,” $”,” ÿ $”,” $”,” $”” ÿ $” }

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

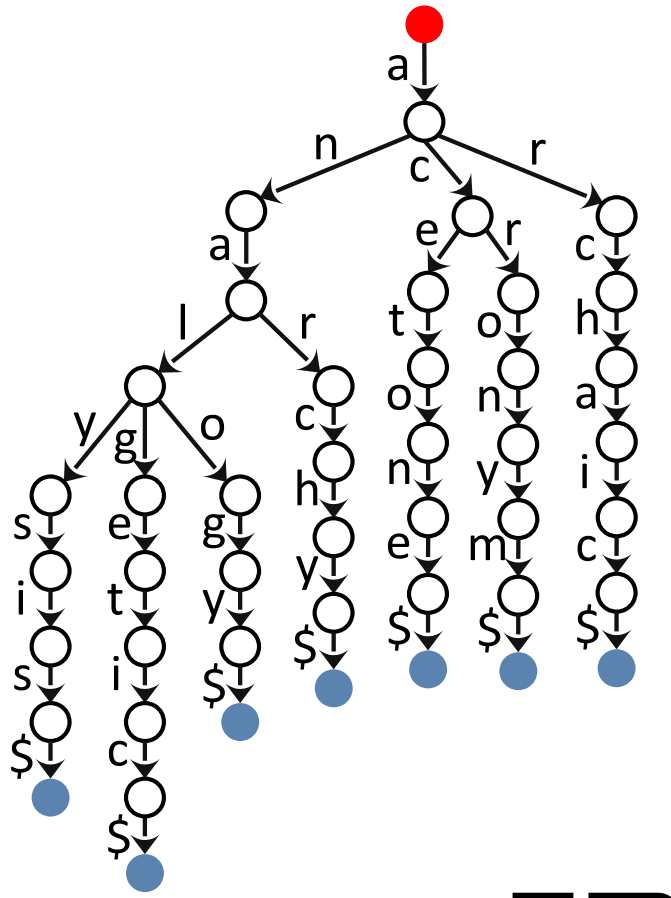
Prefiksno stablo (Trie) (5)

- Mnogo znakovnih nizova u S može dijeliti zajednički prefiks

$S_2 = \{"analysis\$", "analgetic\$", "analogy\$", "anarchy\$", "acetone\$", "acronim\$", "archaic\$"\}$

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		

- Sama definicija Trie-a, koja ograničava da više se izlaznih bridova jednog čvora mapira na isti znak alfabeta, nas prisiljava da zajednički prefiksi znakovnih nizova dijele strukturu stabla



Prefix Tree (Trie) (6)

- Search
  - We take character by character from the character string and follow the transitions in the Trie • If after $ in 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 ← root(T)
    for ch ∈ q do
        if there is transition from cn for character ch to cn_child then
                                                ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            return (false, cn)
    if cn is a leaf then
        return (true, cn)
    return (false, cn)
```

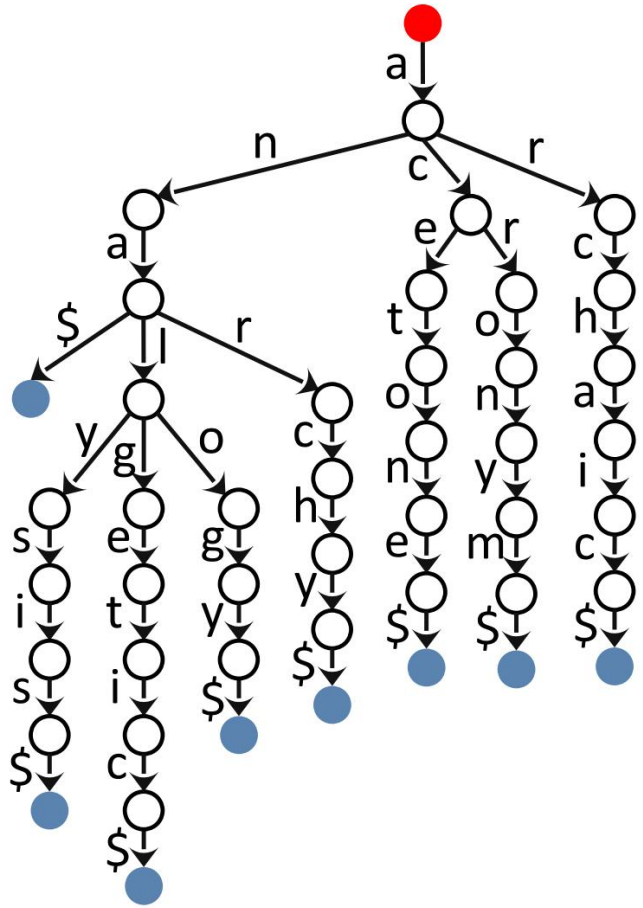
Prefiksno stablo (Trie) (6)

- Pretraživanje
  - Uzimamo znak po znak iz znakovnog niza q i slijedimo tranzicije u Trie-u T
  - Ako smo nakon \$ u q došli do lista, tada S (i Trie T) sadrži q
    - Iskoristili smo sve znakove u q, što nas je dovelo do lista Trie-a T
  - U svakom drugom slučaju q nije pronađen u S
    - Što se desi kada tražimo $q = "ana\$"$ u prethodnom Trie-u ?

```
function SearchTrie(T, q)
    cn ← root(T)
    for ch ∈ q do
        if there is transition from cn for character ch to cn_child then
                                                ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            return (false, cn)
    if cn is a leaf then
        return (true, cn)
    return (false, cn)
```

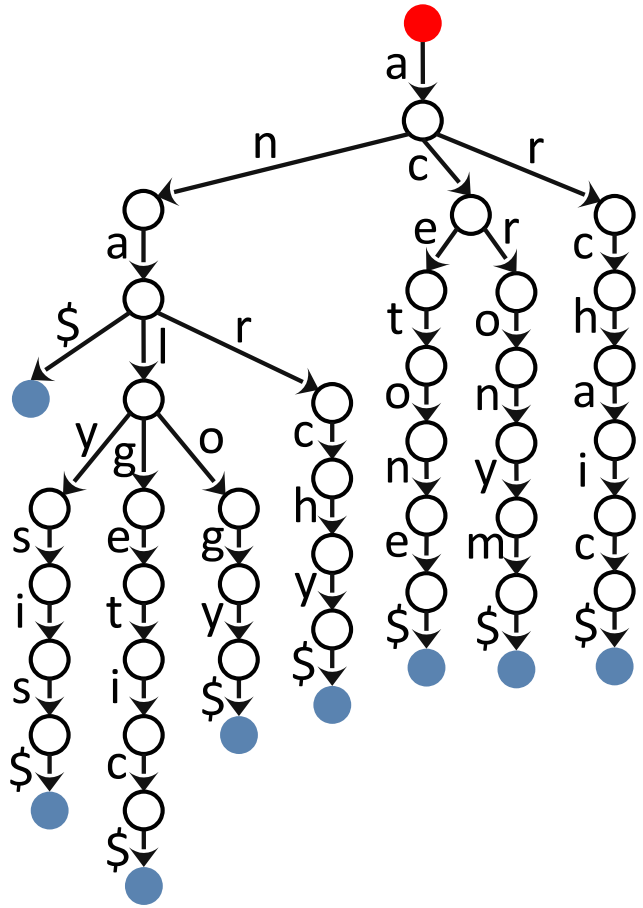
Prefix Tree (Trie) (7)

- An example) = * ÿ { " $”}



Prefiksno stablo (Trie) (7)

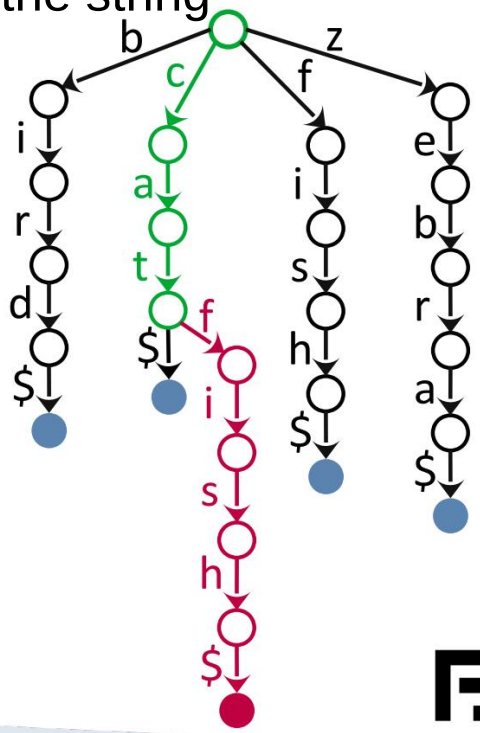
- Primjer $S_3 = S_2 \cup \{"ana\$"\}$



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 , then ÿ and entry is not required
  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 = " ÿ$”

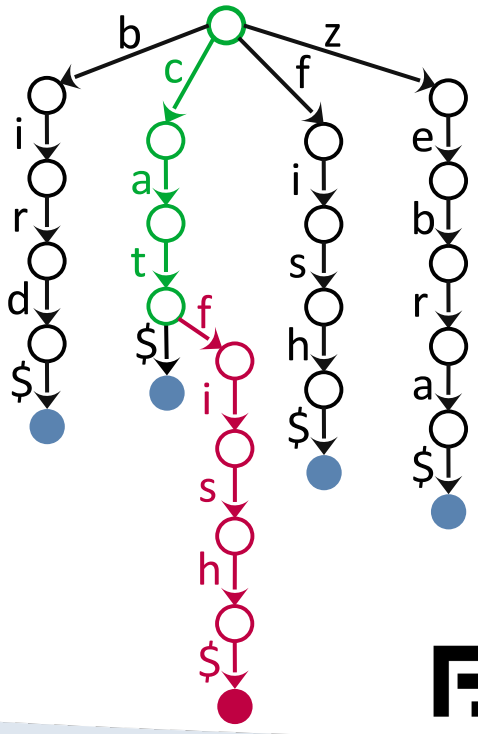
```
procedure InsertTrie(T, s)
    cn ← root(T)
    for ch ∈ s do
        if there is transition from cn for character ch to cn_child then
                                                ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            create new node cn_new
            add transition for character ch from cn to cn_new
            cn ← cn_new
```



Prefiksno stablo (Trie) (8)

- Upis novog znakovnog niza s (*insert*)
  - Krenemo s pretraživanjem $q = s$
    - Ako pronađemo cijeli q u Trie-u T, tada je $s \in S$ i upis nije potreban
    - Inače stanemo na prvom znaku za kojeg nemamo tranziciju u Trie-u – to može biti i $
  - Stvorimo slijednu strukturu Trie-a za ostatak znakovnog niza s
- Primjer: u S_1 dodamo $s =$ „$catfish\$$”

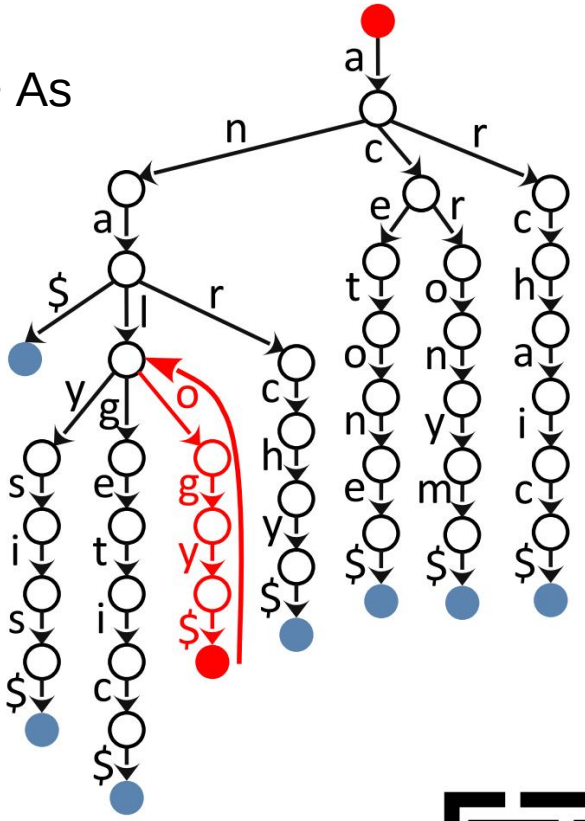
```
procedure InsertTrie(T, s)
    cn ← root(T)
    for ch ∈ s do
        if there is transition from cn for character ch to cn_child then
                                                ▷ array[ch] ≠ NULL in cn
            cn ← cn_child
        else
            create new node cn_new
            add transition for character ch from cn to cn_new
            cn ← cn_new
```



Prefix Tree (Trie) (9)

- Delete character string (remove)
  - We start the search = • If we find
    the whole in the Trie • Otherwise, then ÿ 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 = " $"

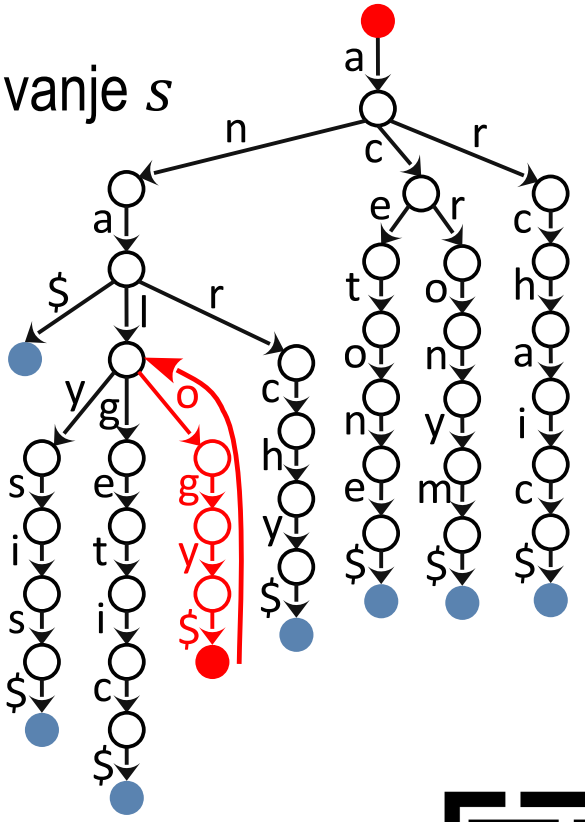
```
procedure RemoveTrie(T, s)
    (succ, cn) ← SearchTrie(T, s)
    if not succ then
        return                                  ▷ s not found in Trie
    s ← 'reversed(s)
    cn ← parent(cn)
    for ch ∈ s do
        remove transition from cn for character ch
        if cn has no children and there is a parent of cn then
            cn ← parent(cn)
        else
            return
```



Prefiksno stablo (Trie) (9)

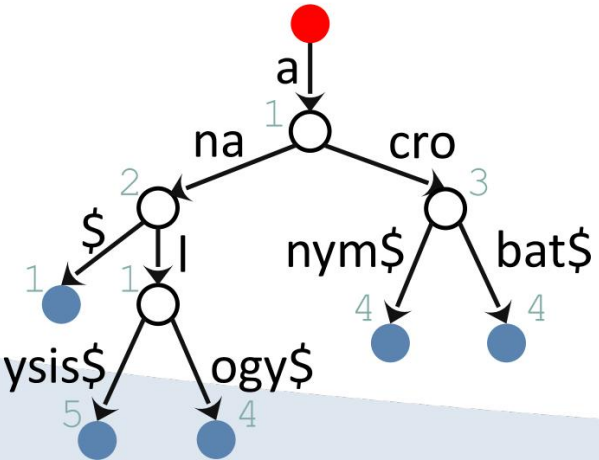
- Brisanje znakovnog niza s (*remove*)
  - Krenemo s pretraživanjem $q = s$
    - Ako pronađemo cijeli q u Trie-u T, tada je $s \in S$ i brisanje je moguće
    - Inače stanemo s brisanjem
  - Vraćamo se od lista do kojeg nas je dovelo pretraživanje s
    - Tako dugo do dok roditeljski čvor ima samo jedno dijete
- Primjer: u S_3 brišemo $s =$ „$analogy\$$”

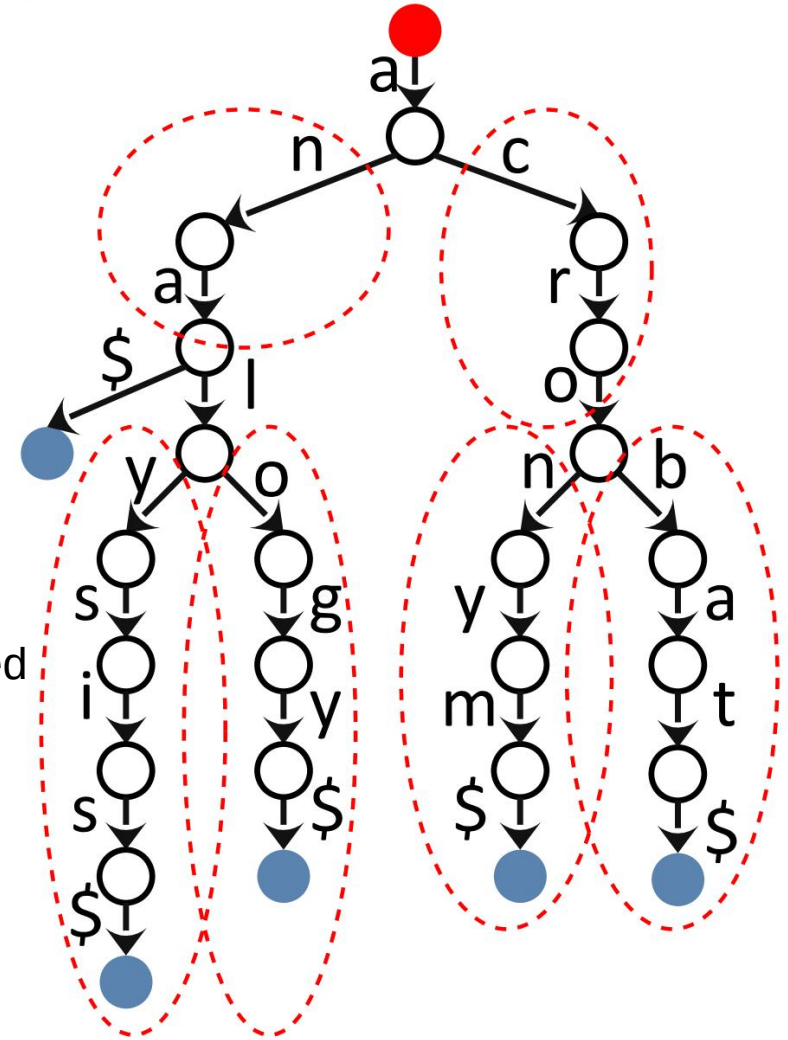
```
procedure RemoveTrie(T, s)
    (succ, cn) ← SearchTrie(T, s)
    if not succ then
        return                                  ▷ s not found in Trie
    s ← 'reversed(s)
    cn ← parent(cn)
    for ch ∈ s do
        remove transition from cn for character ch
        if cn has no children and there is a parent of cn then
            cn ← parent(cn)
        else
            return
```



Patricia tree (1)

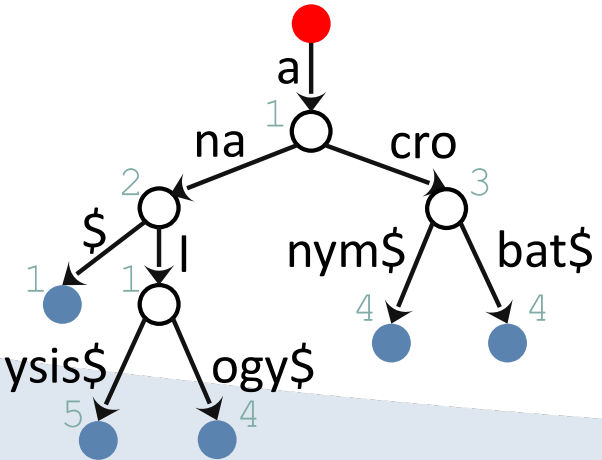
- The Trie structure is not compact
  - Remember all fields in nodes where we have only one single pointer
  - The spatial complexity of the Trie can be reduced by turning the sequence of nodes that have only one transition into one transition
  - We get a compressed Trie or Patricia tree (Practical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

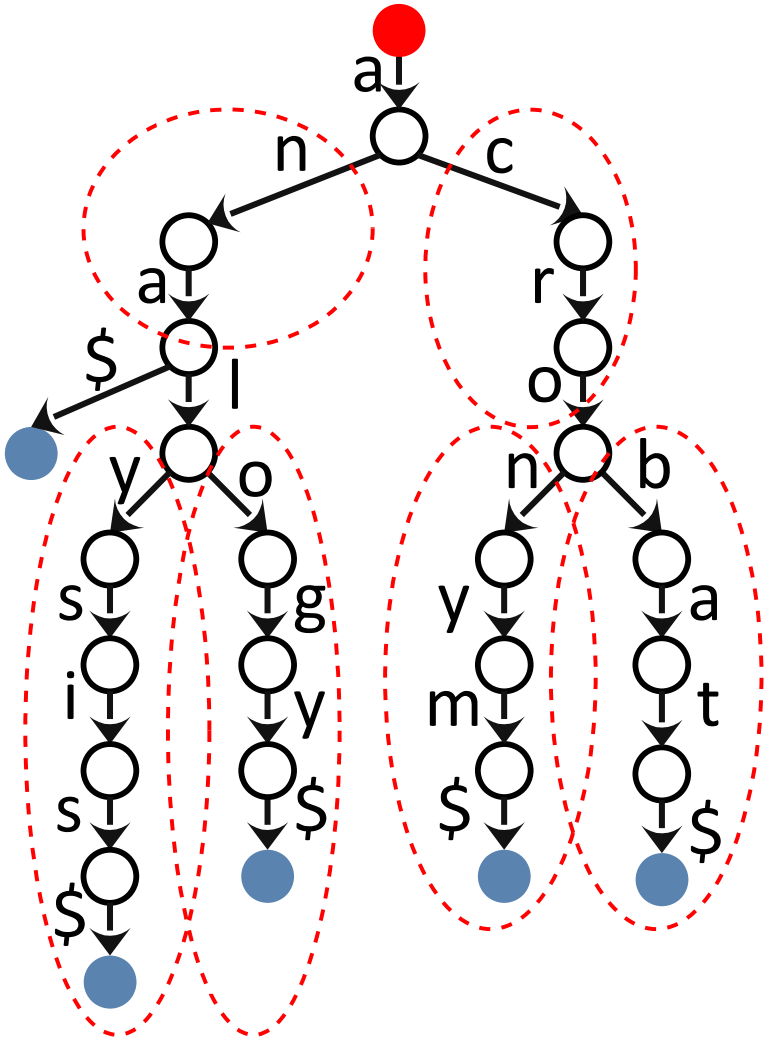




Patricia stablo (1)

- Struktura Trie-a nije kompaktna
  - Sjetite se svih polja u čvorovima u kojima imamo samo jedan jedini pokazivač
  - Prostorna kompleksnost Trie-a se može smanjiti tako da slijed čvorova koji imaju samo jednu tranziciju pretvorimo u jednu tranziciju
  - Dobivamo kompresirani Trie ili Patricia stablo (Practical Algorithm To Retrieve Information Coded In Alphanumeric)





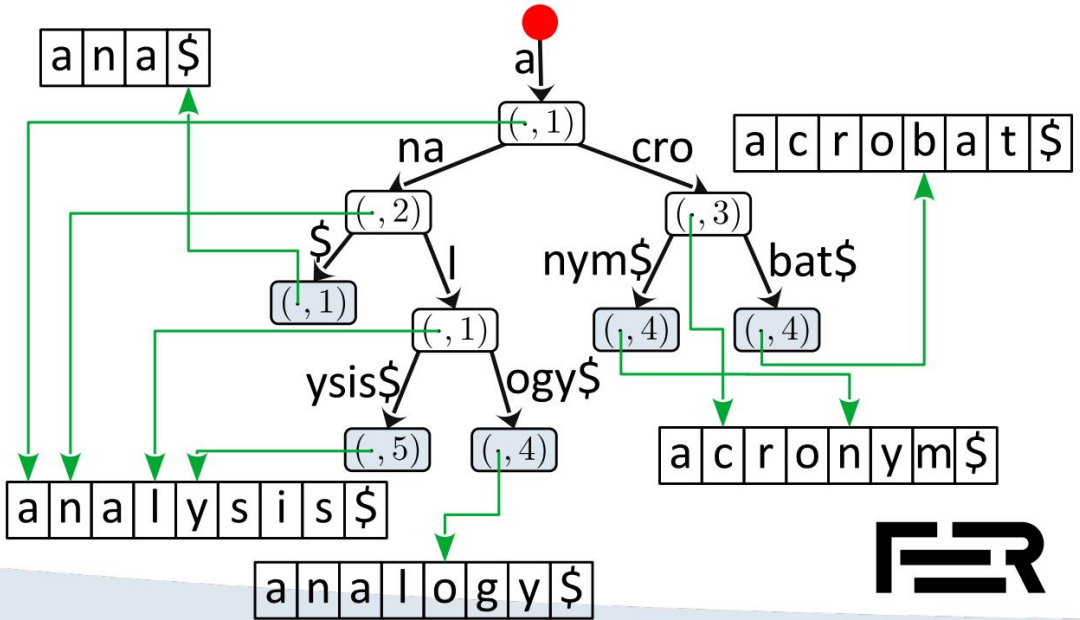
Patricia tree (2)

- Some essential items of the Patricia tree
  - All internal nodes of the Patricia tree have at least two children - otherwise they do 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

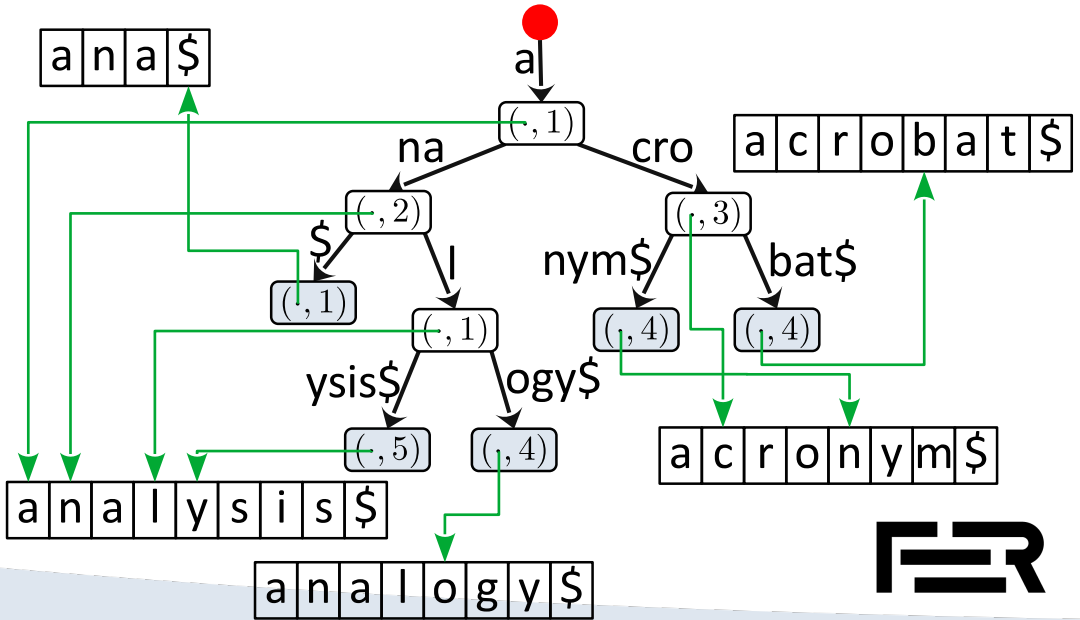
entrance edge

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



Patricia stablo (2)

- Neke bitne stavke Patricia stabla
  - Svi unutarnji čvorovi Patricia stabla imaju barem dva djeteta – inače se vraćamo na Trie strukturu
  - Reprezentacija znakovnog niza temeljena na uređenom paru (p, l) pokazuje se boljom u Patricia stablima – manja potrošnja memorije
    - U memoriji imamo skup znakovnih nizova S
    - Čvorovi pokazuju samo na određene znakovne podnizove u skup S
- Sjetimo se da u svaki čvor Patricia stabla imamo samo jedan ulazni brid
  - U čvoru možemo čuvati uređeni par za tu ulaznu tranziciju



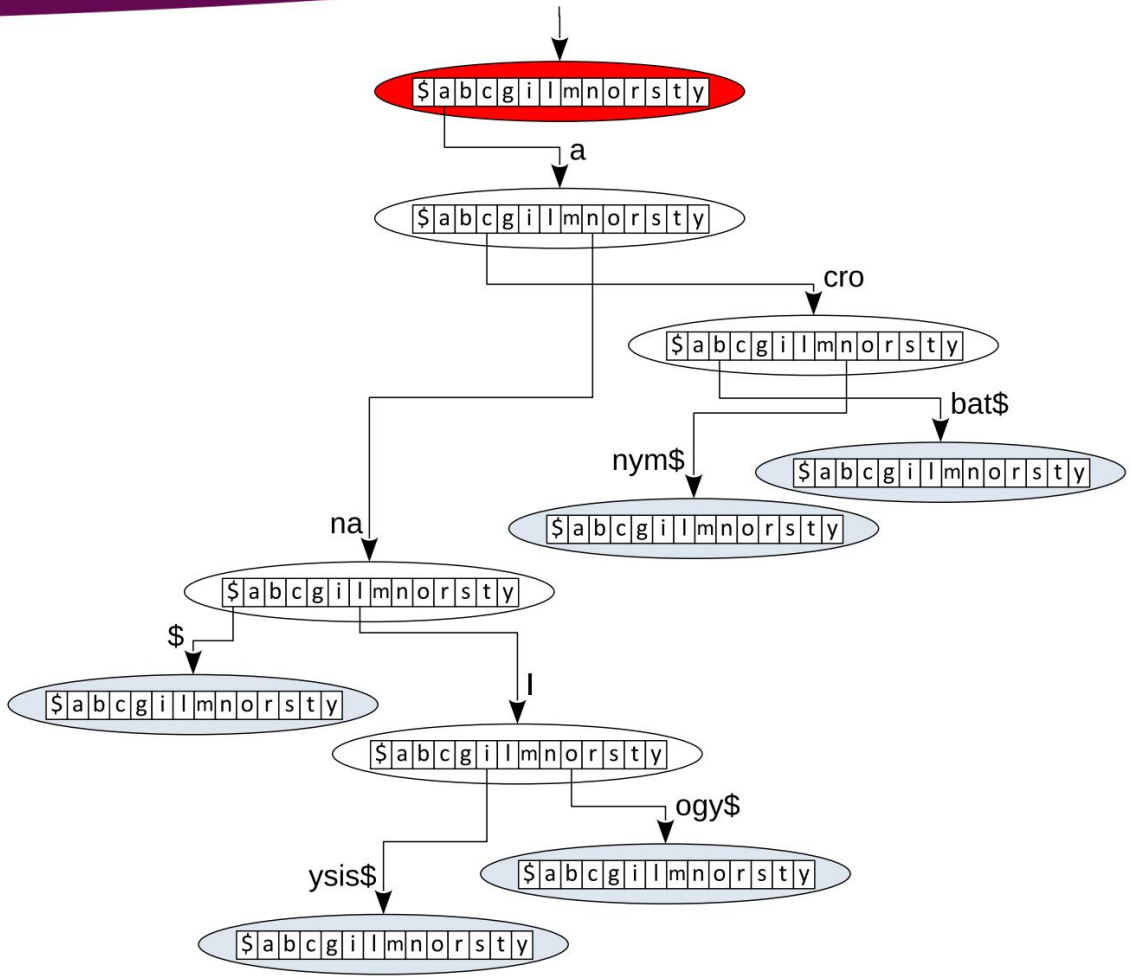
Patricia tree (3)

- Pointers in nodes i

  they still function as code

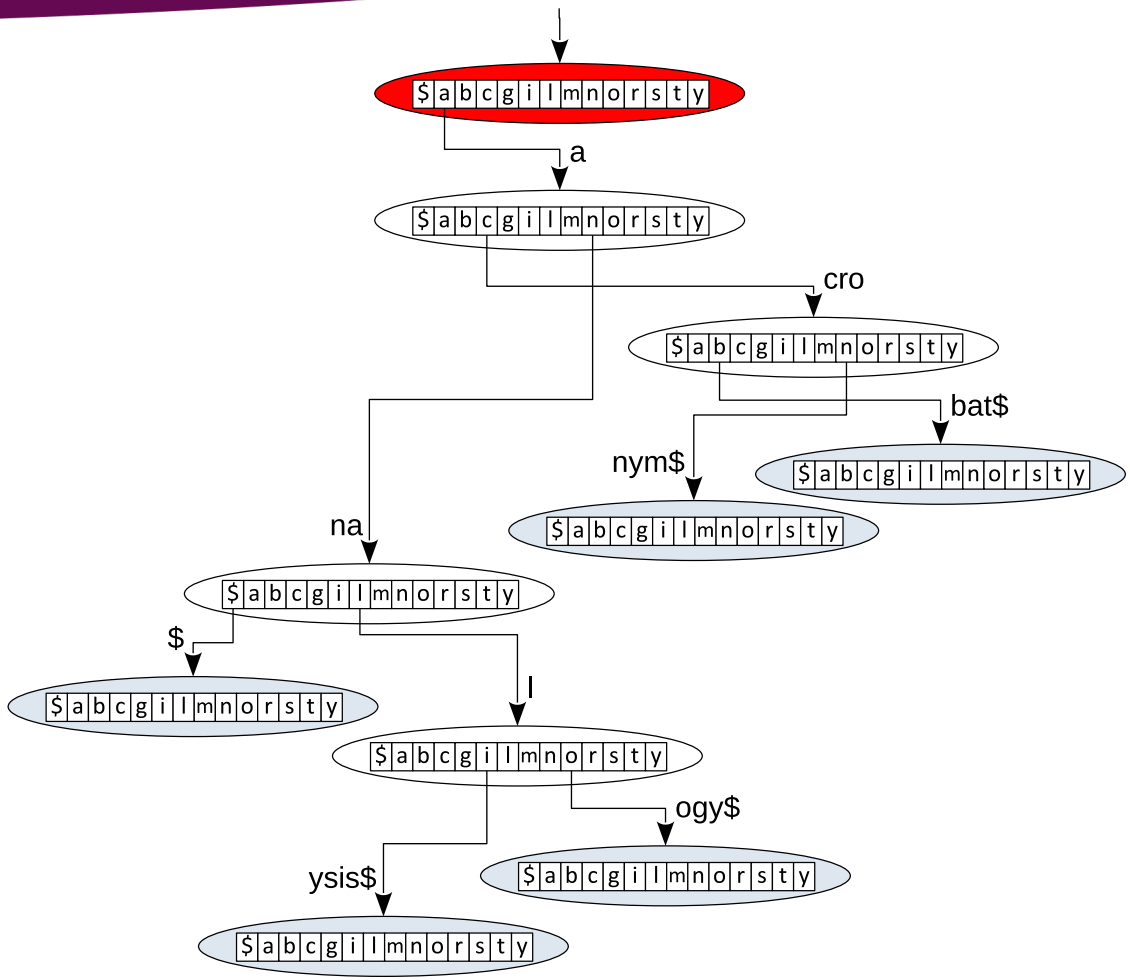
  Trie with a little supplement

  - 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 stablo (3)

- Pokazivači u čvorovima i dalje funkcioniraju kao i kod Trie-a uz malu nadopunu
  - Tranzicije u Patricia stablu predstavljaju znakovni podniz
  - Pokazivač u čvoru odgovara prvom znaku znakovnog podniza tranzicije
  - Nije moguće da dvije tranzicije, dva izlazna brida iz čvora, imaju znakovni podniz koji počinje s istim znakom
    - Ne mogu dijeliti isti prefiks jer bi to značilo da je taj prefiks dio prethodne tranzicije



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 ← 0
    cn ← root(P)
    while cn not leaf do
        if there is transition from cn for character q_p[sc] to cn_c then
                                        ▷ array[q_p[sc]] ≠ NULL in cn
            (t_p, t_l) ← string representation in cn_c
            if q_p[sc : sc + t_l] = t_p[0 : t_l] then
                                ▷ substring matching (Python notation)
                sc ← sc + t_l
                cn ← cn_c
            else
                return false
        else
            return false
    return sc = q_l
```

Patricia stablo (4)

- Pretraživanje zahtijeva usporedbu znakovnog podniza sadržanog u tranziciji sa adekvatnim znakovnim podnizom od q
  - Tijekom pretraživanja trebamo pamtiti na kojem smo mjestu u znakovnom nizu q (varijabla sc)

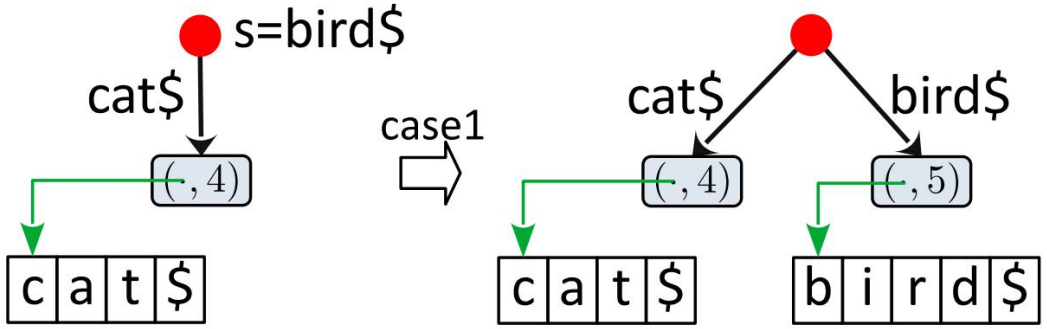
```
function SearchPatricia(P, q = (q_p, q_l))
    sc ← 0
    cn ← root(P)
    while cn not leaf do
        if there is transition from cn for character q_p[sc] to cn_c then
                                                ▷ array[q_p[sc]] ≠ NULL in cn
            (t_p, t_l) ← string representation in cn_c
            if q_p[sc : sc + t_l] = t_p[0 : t_l] then
                                        ▷ substring matching (Python notation)
                sc ← sc + t_l
                cn ← 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

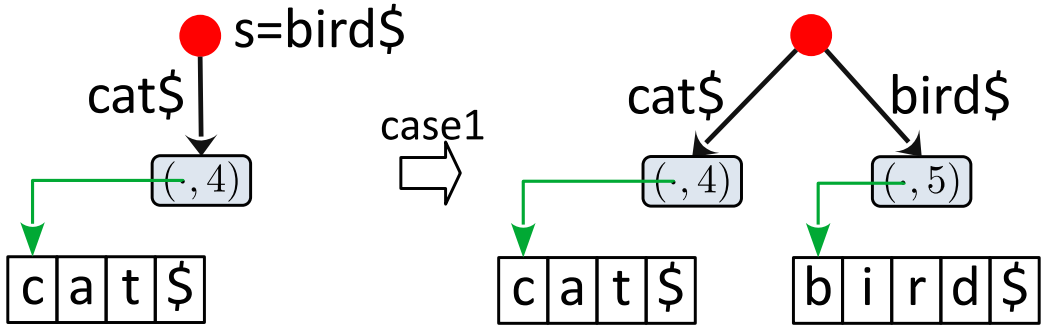
string



Patricia stablo (5)

- Upis novog znakovnog niza s je nešto kompliciraniji nego u Trie-u
  - Započinjemo s pretraživanjem
  - Gledamo s aspekta trenutnog čvora – počinjemo iz korijenskog čvora

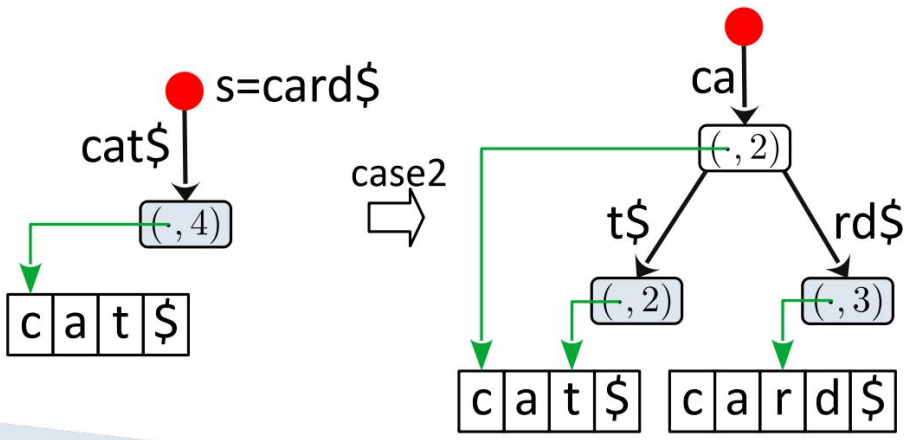
1. Nije moguće naći izlaznu tranziciju iz čvora koja bi barem djelomično podudarala ostatku znakovnog niza s
   - Dodamo novi list u strukturu i tranziciju koja je identična ostatku znakovnog niza s



Patricia tree (6)

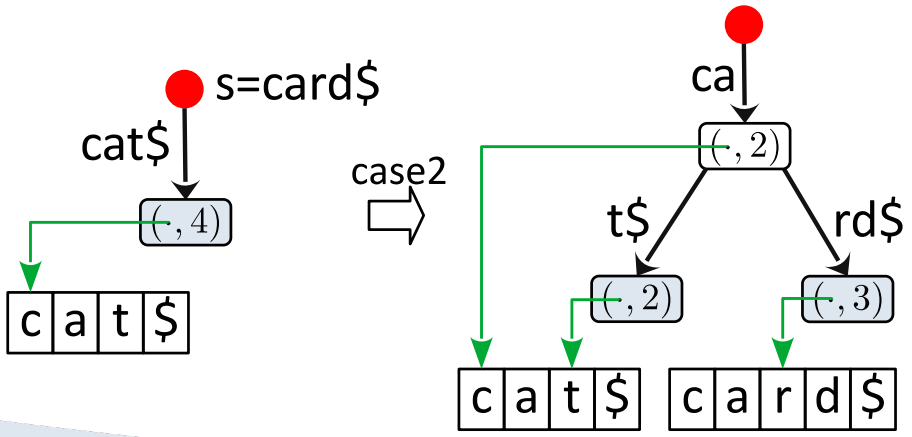
2. It is possible to find an output transition from a node that partially matches part of the rest of the character string $2! which separates

- 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 corresponds to the remainder of the string 3456 that corresponds to the 3456 of the split transition in the previous step



Patricia stablo (6)

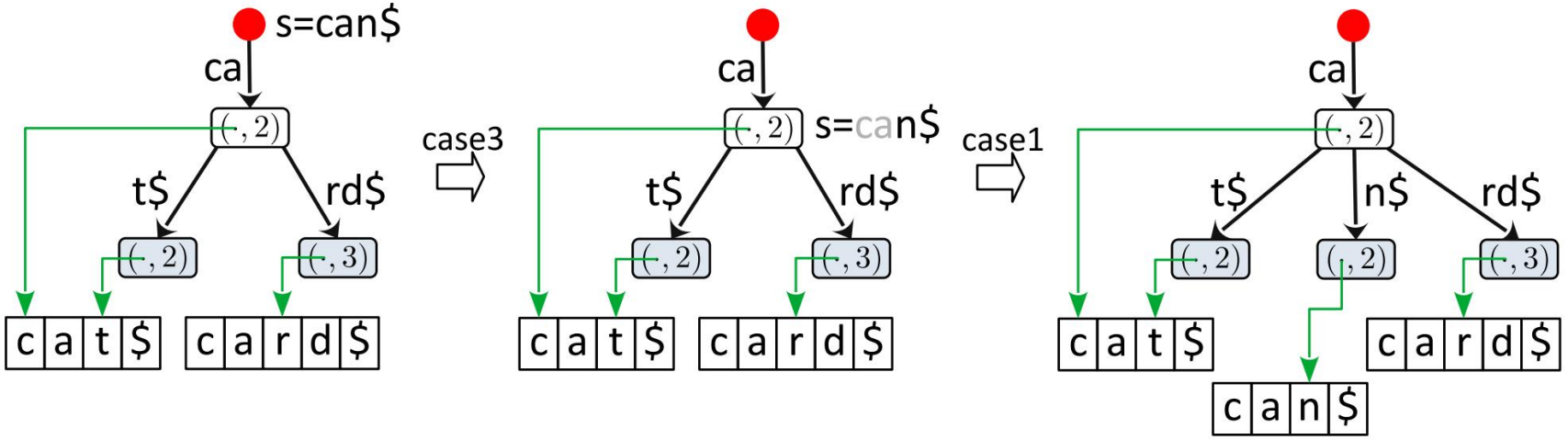
2. Moguće je naći izlaznu tranziciju iz čvora koja djelomično podudara dijelu ostatka znakovnog niza s
   - Dodamo novi čvor cn_{ins} kojim razdvajamo staru tranziciju na dva dijela, prvi dio koji se podudara s ostatkom znakovnog niza s i drugog dijela koji se ne podudara
   - Dodamo novi list cn_{leaf} i tranziciju između cn_{ins} i cn_{leaf} koja odgovara ostatku znakovnog niza s, a koji se nije podudarao s prvim dijelom razdvojene tranzicije u prethodnom koraku



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

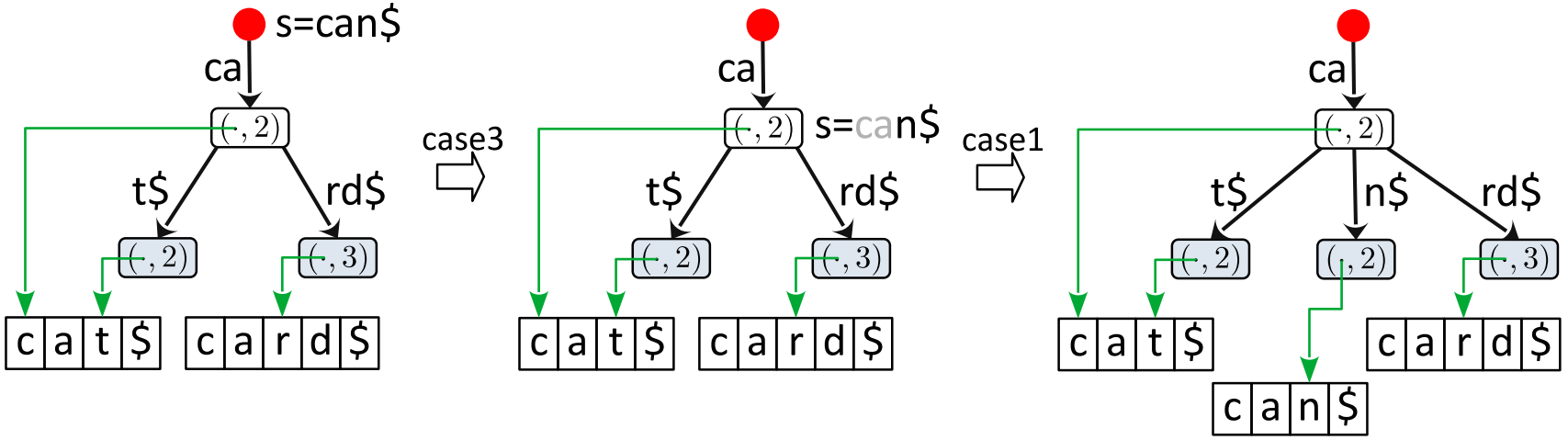
• We pay attention to the rest of the character string



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

Patricia stablo (7)

3. Moguće je naći izlaznu tranziciju iz čvora koja potpuno podudara dijelu ostatka znakovnog niza s
   - Prolazimo tranziciju i pozicioniramo se u novom čvoru
   - Pazimo na ostatak znakovnog niza s



- Evolucija Patricia stabla – slijed upisa i brisanja znakovnih nizova u S : svaki korak se može vizualizirati

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** ▷ $O(|s|)$
    if there is transition from cn for character $s_p[sc]$ to cn_c **then**
      ▷ $array[s_p[sc]] \neq NULL$ 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** ▷ case 2
        ▷ we know for sure that the first character is matched
        $i \leftarrow$ first unmatched character from t_p ▷ $i > 0$
        remove transition between cn and cn_c
        add node cn_{ins} with (t_p, i) ▷ $O(|\Sigma|)$
        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 $(s_p + (sc + i), s_l - (sc + i))$
        ▷ $O(|\Sigma|)$
        add transition between cn_{ins} and cn_{leaf}
        return
      else if $s_p[sc : sc + t_l] = t_p[0 : t_l]$ **then** ▷ case 3
        $sc \leftarrow sc + t_l$
        $cn \leftarrow cn_c$
    else ▷ case 1
      add leaf node cn_{leaf} with $(s_p + sc, s_l - sc)$ ▷ $O(|\Sigma|)$
      add transition between cn and cn_{leaf}
      return

Patricia stablo (8)

- Kompleksnost dodavanja znakovnog niza je $O(|s| + |\Sigma|)$

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

Patricia tree (9)

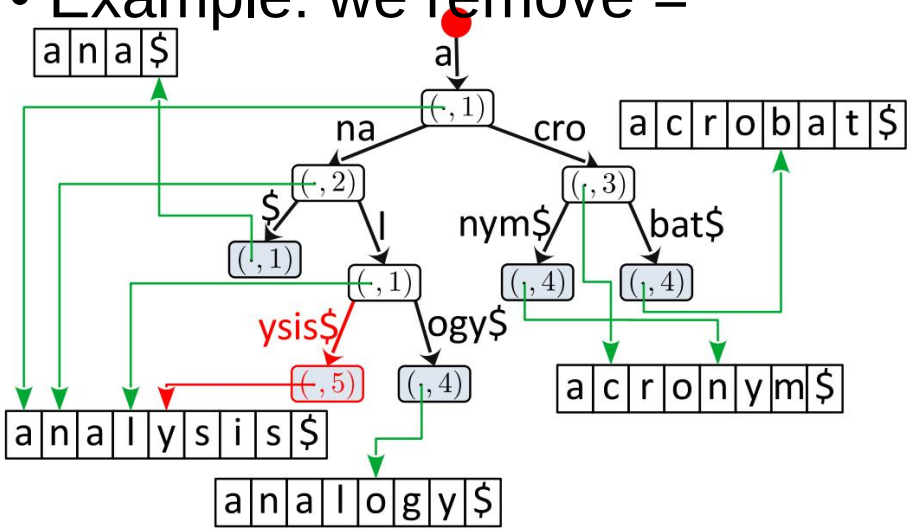
- Deleting a string from a set •
  Starts with a search - must exist in the Patricia tree
  - We start from the leaf that represents as in Trie
  - We remove the leaf and its input transition
  - If the parent of the removed leaf remains with one child, then that parent should be removed and its transitions 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 stablo (9)

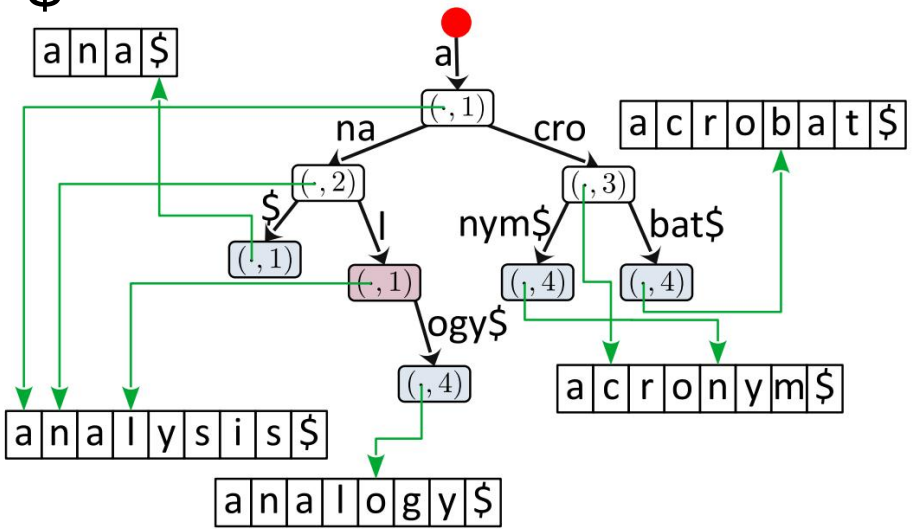
- Brisanje znakovnog niza s iz skupa S
  - Započinje pretraživanjem – s mora postojati u Patricia stablu
  - Krećemo od lista koji predstavlja s, kao i u Trie-u
  - Uklanjamo list i njegovu ulaznu tranziciju
  - Ako roditelj uklonjenog lista ostane s jednim djetetom, tada se taj roditelj treba ukloniti i njegove tranzicije spojiti – nekompresirani slučaj
  - Spajanje tranzicija za znakovne nizove (p, l) je jednostavno
    - Pokazivač p na početak prebacimo za nekoliko znakova prema početku znakovnog niza
    - Duljinu l povećamo za taj broj znakova

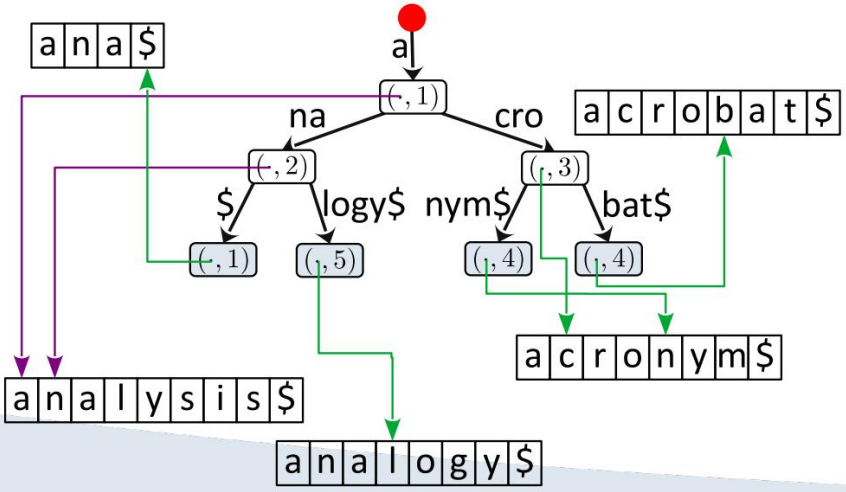
Patricia tree (10)

- Example: we remove = "



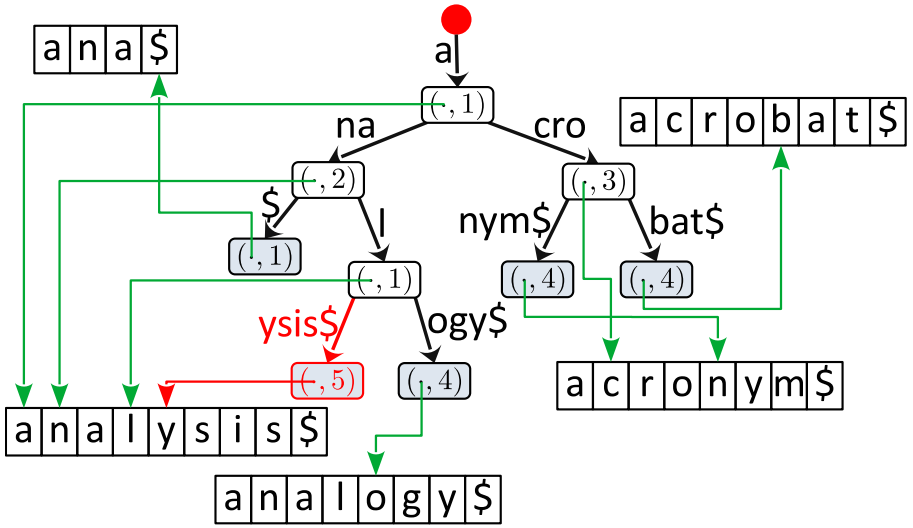
$"

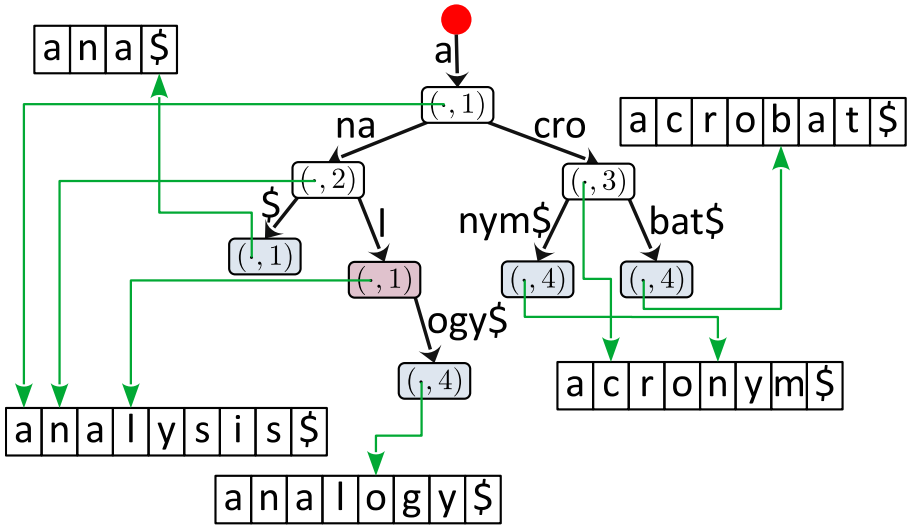


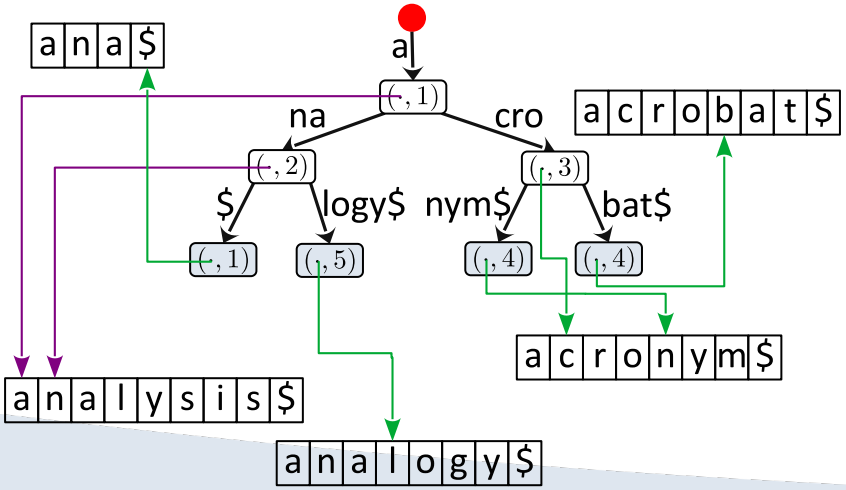


Patricia stablo (10)

- Primjer: uklanjamo $s = „analysis\$”$

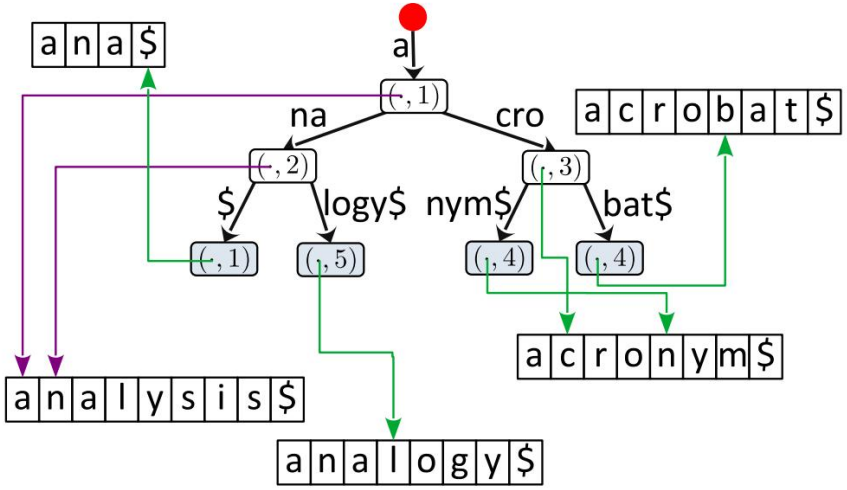


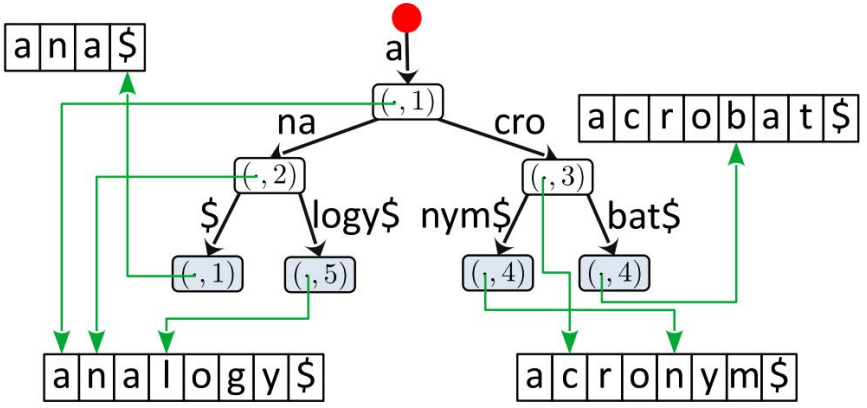




Patricia tree (11)

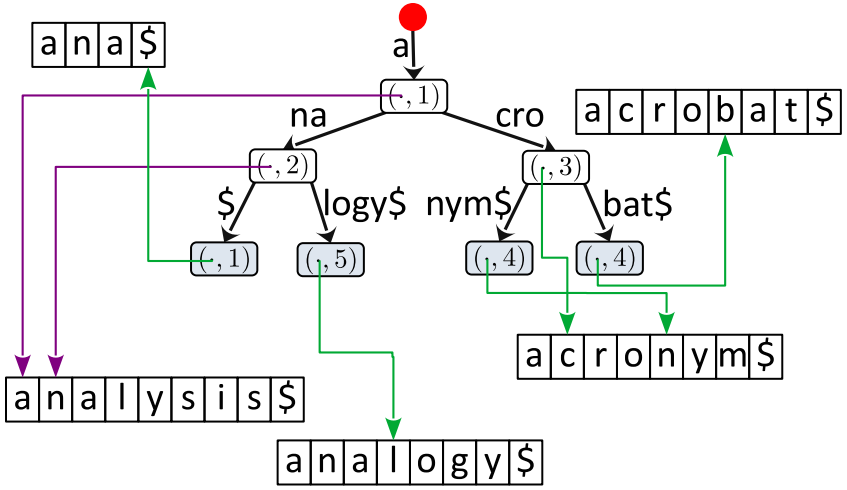
• Character string ” $" we need to remove from memory (destructor in Cu) • However, we have two more pointers to that instance • We transfer the pointers to the character string one of the remaining children

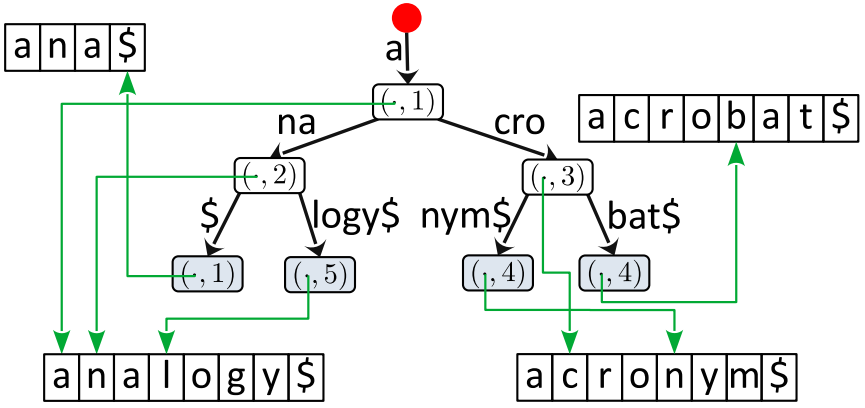




Patricia stablo (11)

- Znakovni niz ”$analysis$$” trebamo ukloniti iz memorije (destruktor u C-u)
  - No, imamo još dva pokazivača na tu instancu
  - Pokazivače prebacimo na znakovni niz jedno od preostale djece

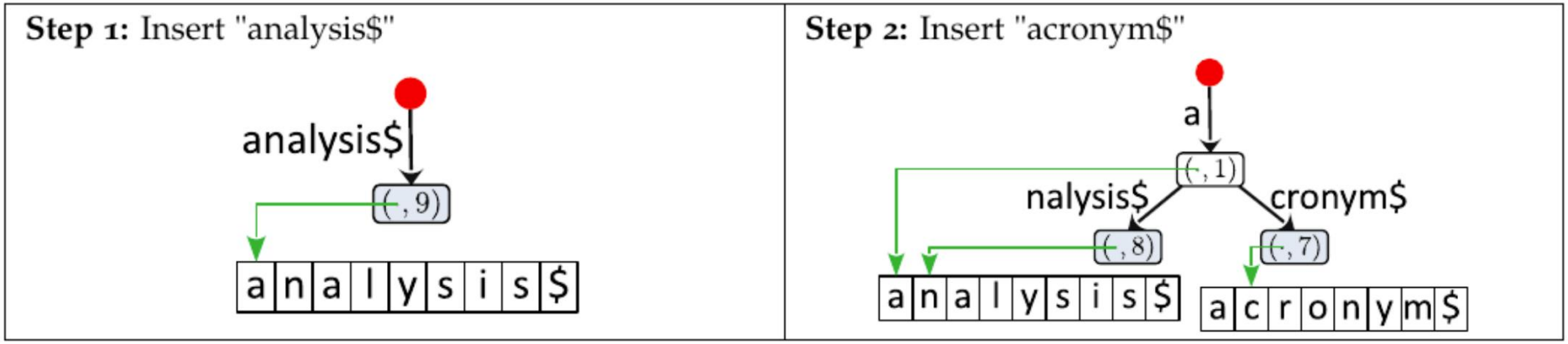




Patricia tree (12)

- An example
  - Draw the evolution of the Patricia tree to write the next sequence of characters „of strings

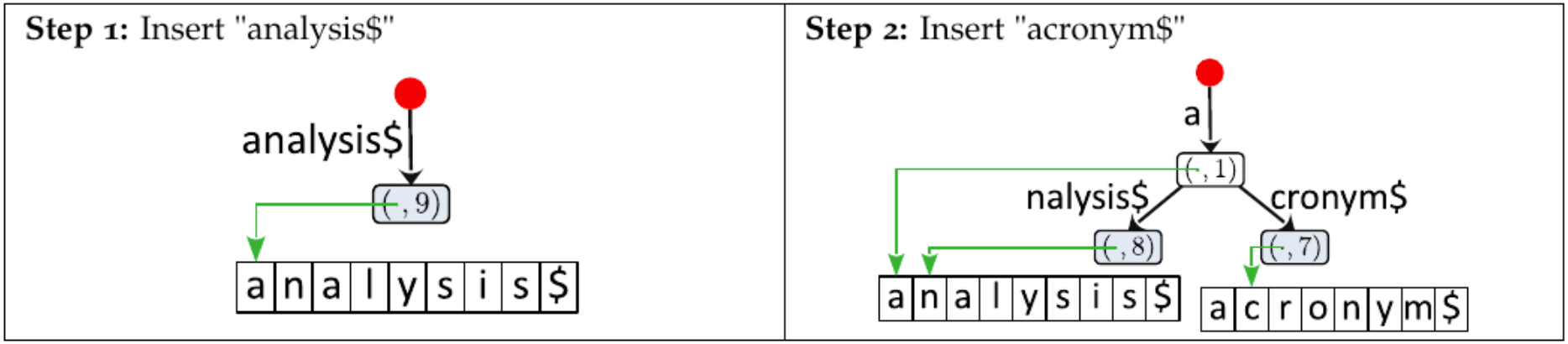
⟨ $"," $"," $"," $"," $"⟩



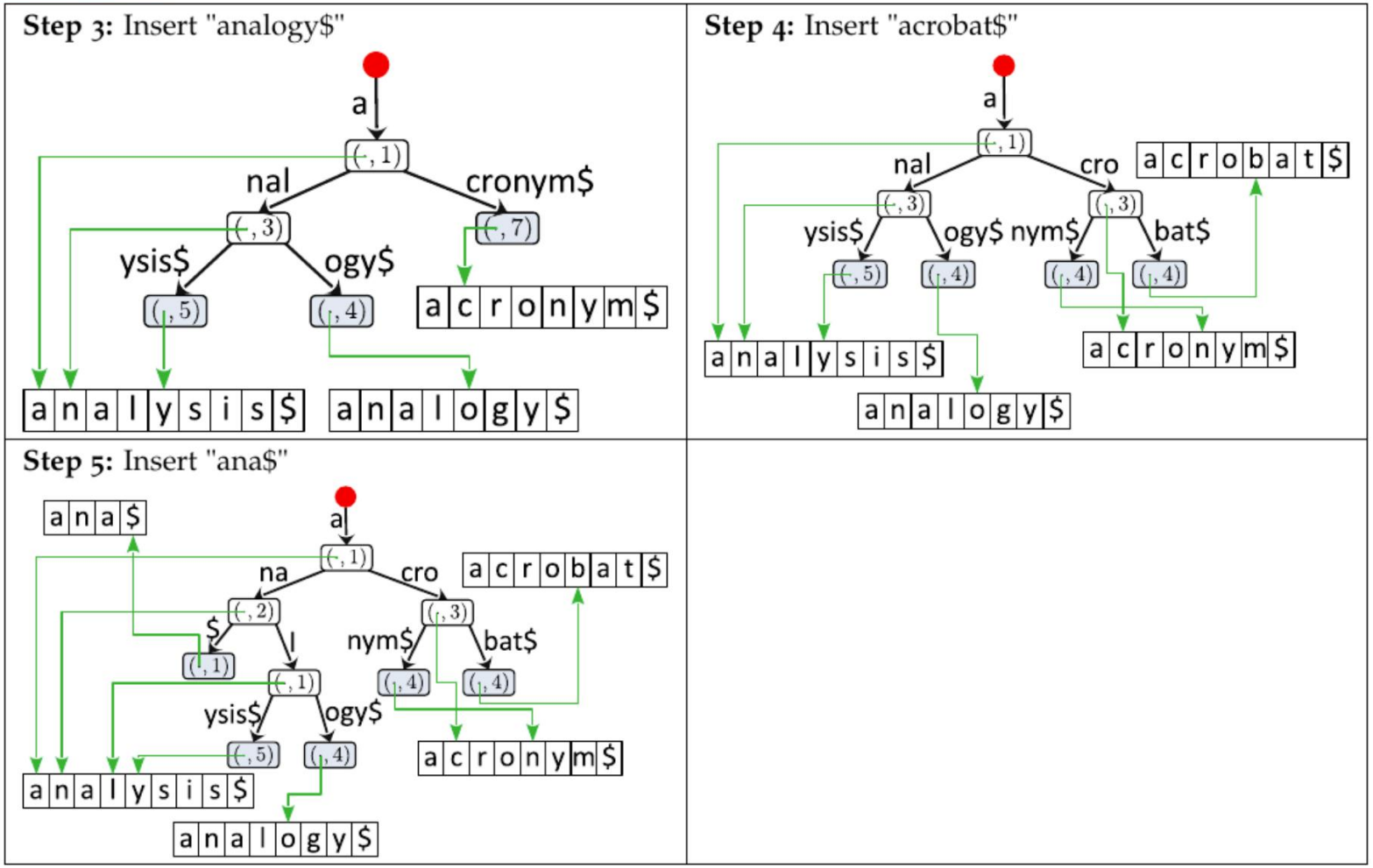
Patricia stablo (12)

- Primjer
  - Nacrtajte evoluciju Patricia stabla za upis sljedećeg slijeda znakovnih nizova

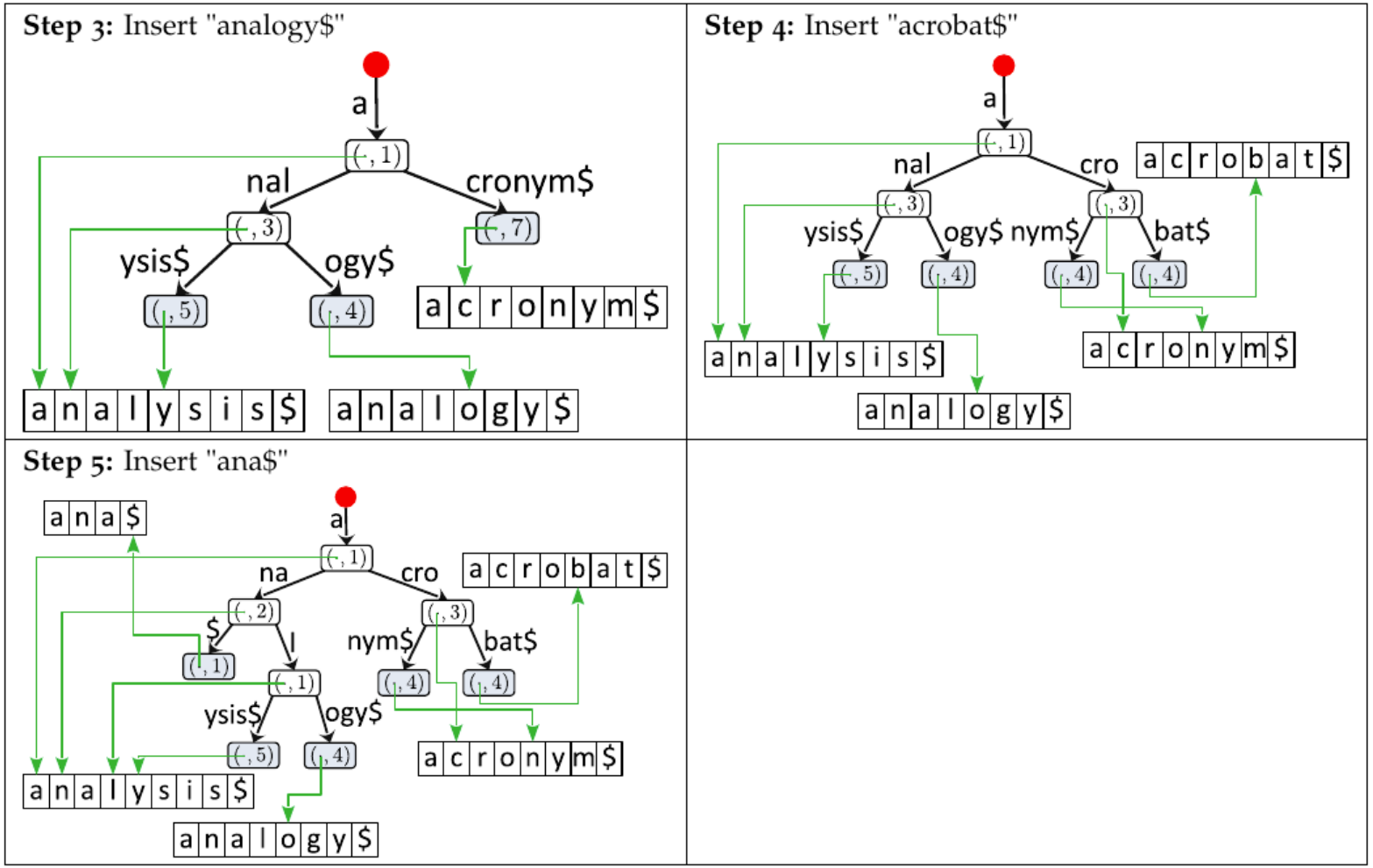
$\langle "analysis\$", "acronym\$", "analogy\$", "acrobat\$", "ana\$" \rangle$



Patricia tree (13)



Patricia stablo (13)



Patricia tree (14)

- Some significant applications: IP routing (in routers), IP filtering, firewall, IP lookup, ...
- If we take ÿ = 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 . * . *

Patricia stablo (14)

- Neke značajne primjene: IP routing (u routerima), IP filtering, firewall, IP lookup, ...
- Ako za alfabet uzmemo $\Sigma = \{0,1\}$
- IPv4 adrese možemo transformirati u znakovni niz kao

192.168.11.45/32

1100 0000 . 1010 1000 . 0000 1011 . 0010 1101

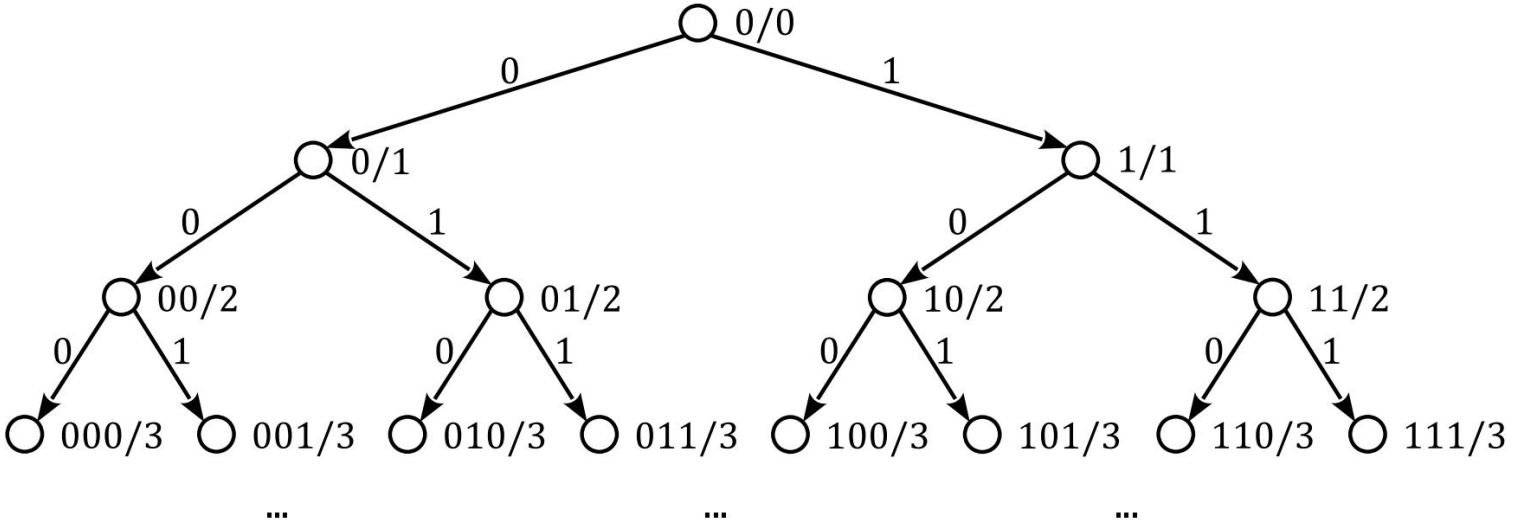
- Ili masku kao

192.168.0.0/16

1100 0000 . 1010 1000 . * . *

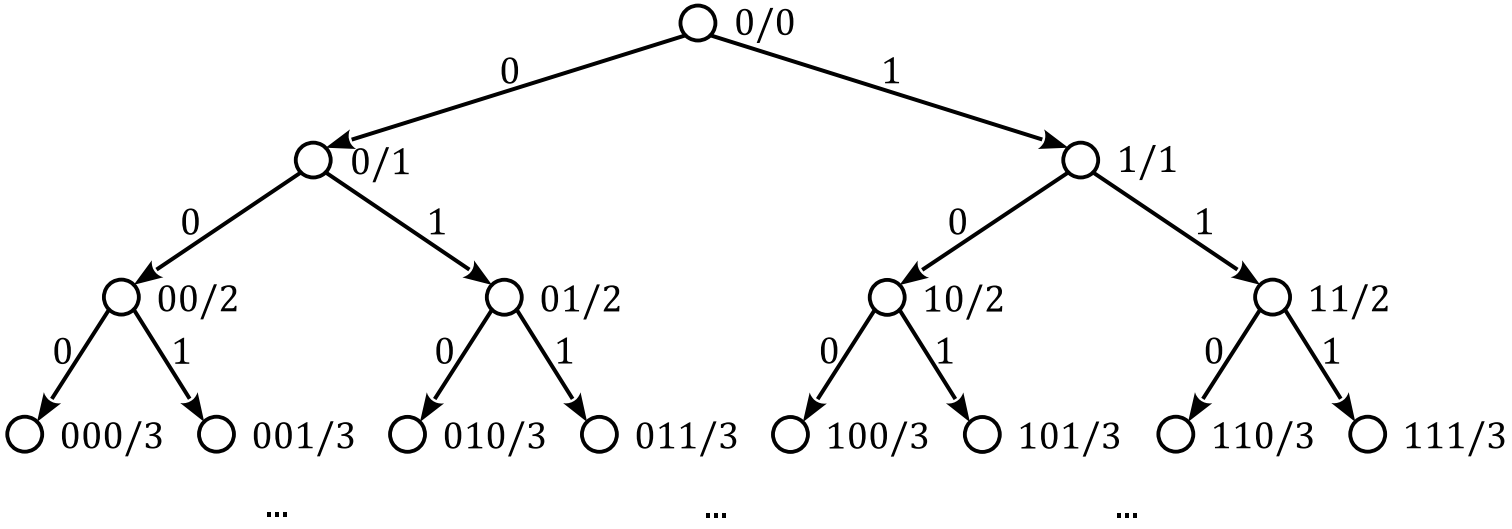
Radix trees (1)

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



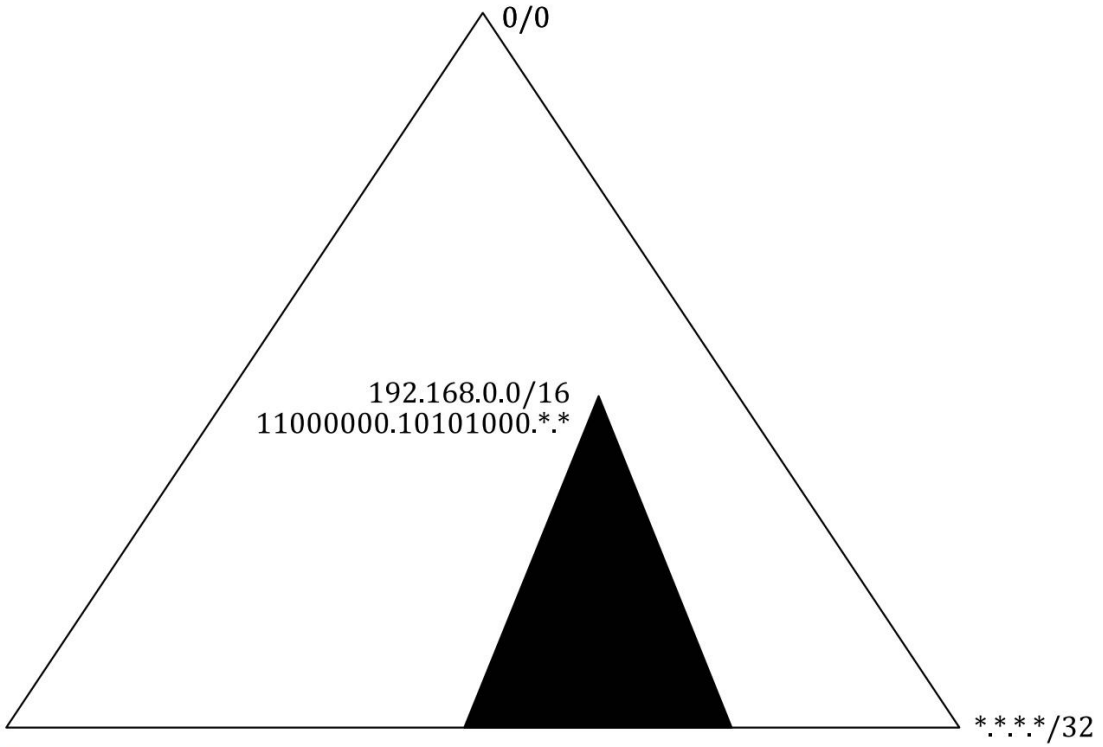
Radix stabla (1)

- Radix stablo za IP routing gradimo na sljedeći način – binarno stablo



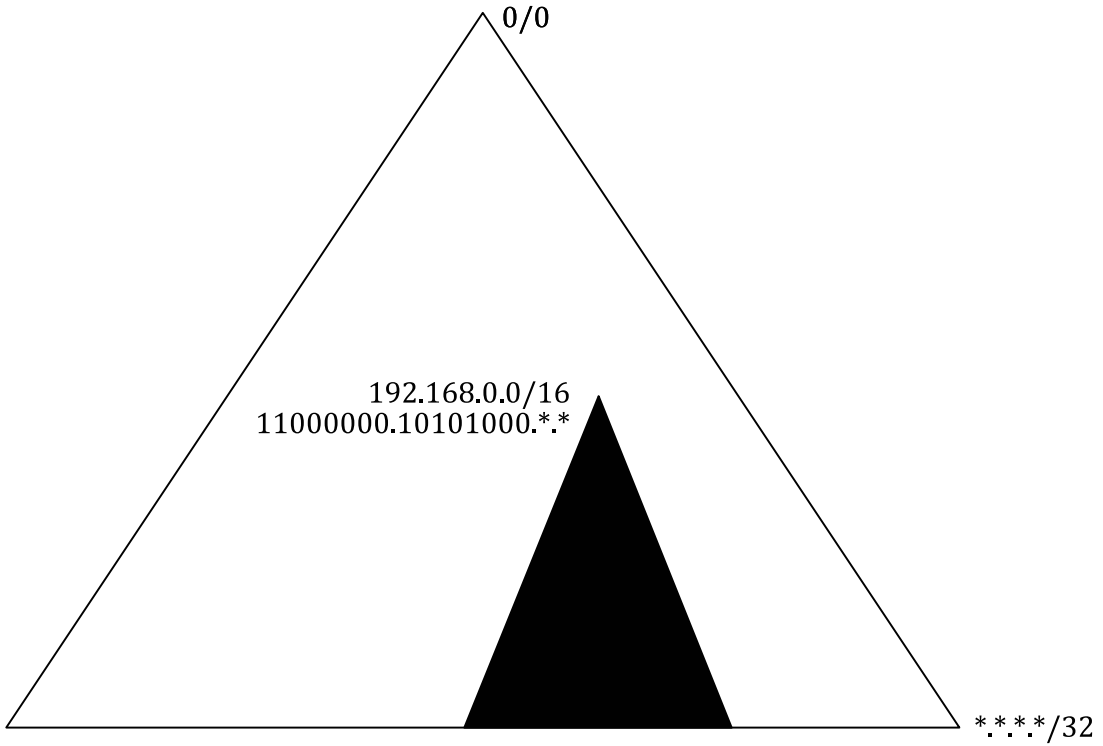
Radix trees (2)

- 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



Radix stabla (2)

- U takvom Radix stablu, podmreža se detektira kroz svoju masku
- Svi čvorovi podstabla predstavljaju članove podmreže 192.168.0.0/16



IP routing

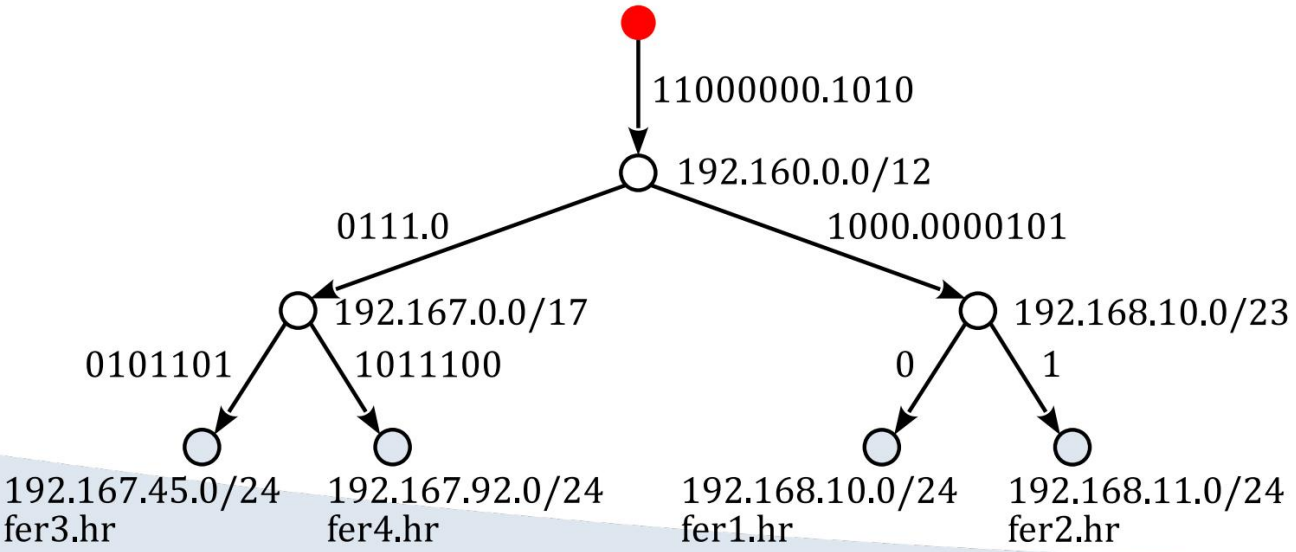
• We compress the Radix tree – we get a binary Patricia tree •

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)

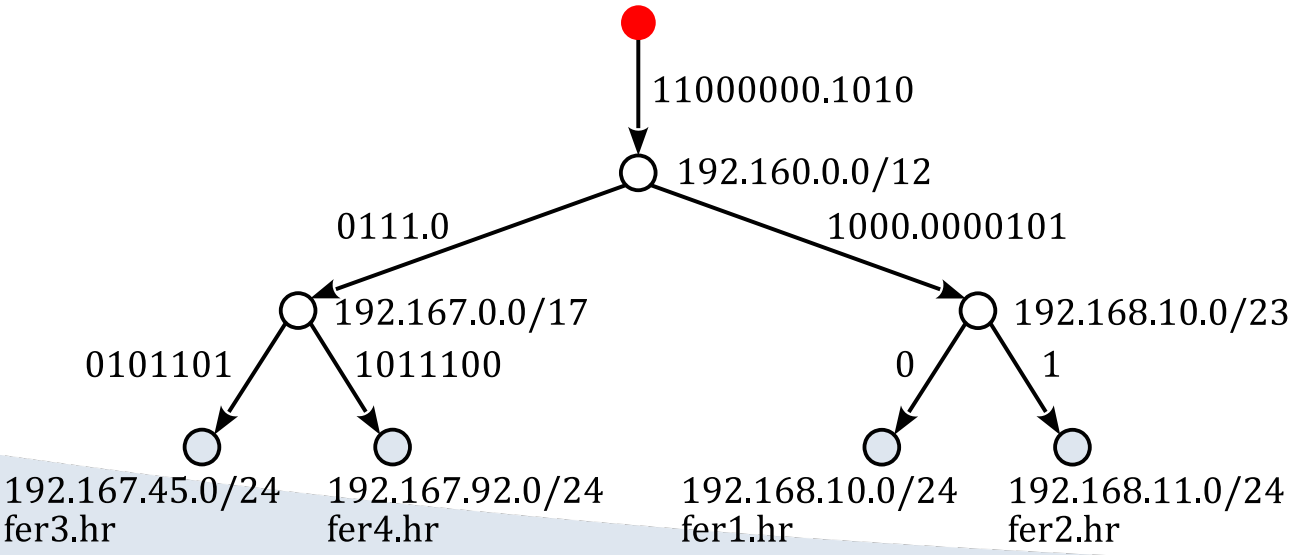


IP routing

- Kompresiramo Radix stablo – dobivamo binarno Patricia stablo
- Sljedeća IP routing tablica

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

  - pretvara se u Patricia stablo
  - listovi mogu imati pokazivače na definiciju sučelja (ili na nešto drugo)



Suffix fields
Suffix trees

Sufiksna polja
Sufiksna stabla

String suffix

- Let's imagine the string = • " $”

  where = 1 [ ], [2], ... , [ ], ... , [ ], and is the jth 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 , ... , [ ]

  ! = [ ], [+ 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	$							

Sufiks znakovnog niza

- Zamislimo znakovni niz $t =$ „$program\$$”
  - gdje je $t = t[1], t[2], \ldots, t[j], \ldots, t[n]$, a $t[j]$ je j-ti znak znakovnog niza
- Iz znakovnog niza možemo izvesti skup svih njegovih sufiksa, gdje je i-ti sufiks znakovnog niza t, definiran kao

$$t_i = t[i], t[i+1], \ldots, t[n]$$

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 ti suffix [ ], we should copy the characters from i to nu into that new instance.
  - For all character string suffixes, this means the next copy number

(ÿ 1 +)ÿ 2 (+ ÿ + 1 =)

$$\frac{(\quad +1\ )}{\qquad}\ -$$

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

Svi sufiksi znakovnog niza

- Reprezentacija znakovnog niza s uređenim parom daje nam prednost kod stvaranja liste svih sufiksa znakovnog niza
- Naivno: kada bismo htjeli stvoriti novu instancu znakovnog niza za i-ti sufiks $t[i]$, trebali bismo kopirati znakove od i do n u tu novu instancu.
  - Za sve sufikse znakovnog niza to znači sljedeći broj kopiranja

$$(n-1)+(n-2)+\cdots+1=\frac{n(n+1)}{2}-n$$

- Kada koristimo uređeni par (p, l) kao reprezentaciju znakovnog niza, u n iteracija radimo
  - Pomičemo pokazivač p za jedno mjesto unaprijed
  - Smanjimo l za jedan

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

  = $⟨ $, $, $, $, $, $, $⟩

Sufiksno polje (1)

- Prethodnu listu sufiksa sortiramo alfabetski (abecedno) i dobivamo sljedeću sortiranu listu sufiksa (sjetimo se $=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	$	

  - gdje je $A = \langle A_1, A_2, \ldots, A_i, \ldots, A_n \rangle$ sortirani niz sufiksa koji definira sufiksno polje

$SA = \langle \$, am\$, gram\$, m\$, ogram\$, program\$, ram\$, rogram\$ \rangle$

Suffix field (2)

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

| |

  - Suffix field search with binary search algorithm is (log)
  - 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

Sufiksno polje (2)

- Kompleksnost stvaranja sufiksnog polja je dosta visoka
- Koristimo algoritam za sortiranje što je $O(n \log n)$, pa kad tome dodamo usporedbu znakovnih nizova, to se može aproksimirati kao $O(n^2 \log n)$
- Ideja sufiksnog polja je provjeriti da li se znakovni niz q nalazi u znakovnom nizu t
  - Naivno pretraživanje bi bilo $O(|q| * n)$
  - Pretraživanje sufiksnog polja sa binarnim algoritmom za pretraživanje je $O(|q| \log n)$
  - Binarni algoritam za pretraživanje je sličan pretraživanju binarnog stabla
    - Pronađemo sredinu, pa provjerimo da li je q manji ili veći od te sredine
    - S obzirom na rezultat, pretraživanje svodimo na pola sufiksnog polja
    - Nastavljamo rekurzivno do konačnog rezultata: $false$ ili $true$

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 ÿ-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.

Sufiksno polje (3) (info)

- Stvaranje sufiksnog polja može biti linearno korištenjem posebnih algoritama, kao *skew* algoritam
- Korištenjem koncepata LCP-a i ℓ-matrice, pretraživanje se može ubrzati na $O(|q| + \log n)$
- Ti su algoritmi i načini stvaranja kompleksni. Studenti koji žele znati više o ovome mogu pronaći te teme u skripti NASP-a.

Suffix Trie (1)

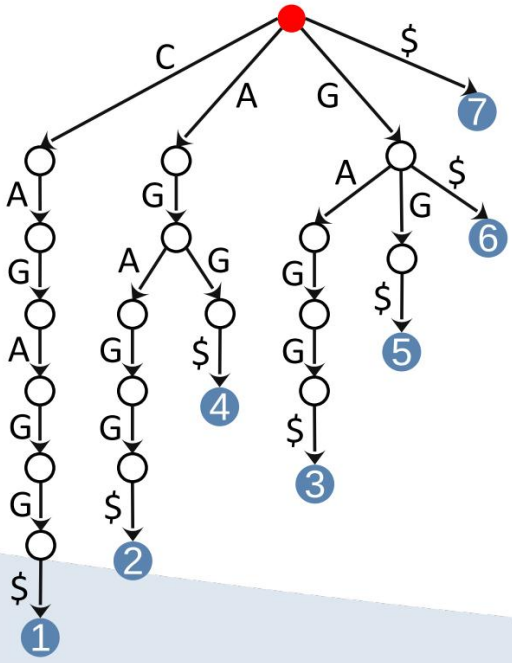
- We can transform the list of all suffixes of a certain character string in Trie

- For the alphabet ÿ = { , , , , $} (DNA bases) and = $

  we have the following list of all suffixes

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

- Converted to the suffix Trie

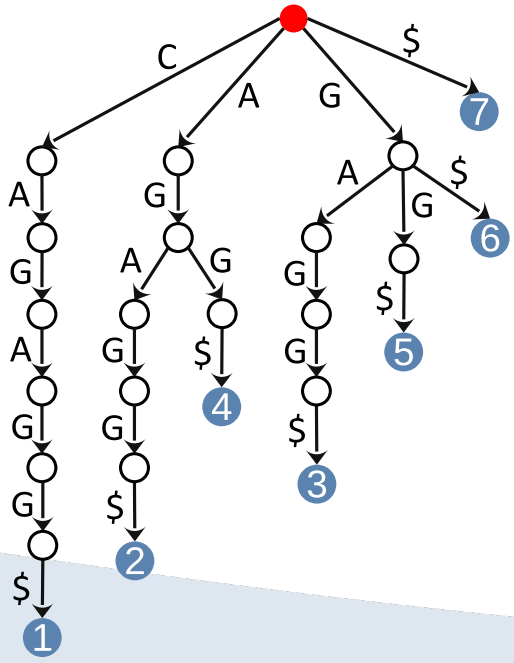


- Each leaf of a suffix Trie contains the sequence number of that suffix or

Sufiksni Trie (1)

- Listu svih sufiksa određenog znakovnog niza možemo transformirati u Trie
- Za alfabet $\Sigma = \{C, A, G, T, \$\}$ (DNA baze) i $t = CAGAGG\$$ imamo sljedeću listu svih sufiksa
- Pretvoreno u sufiksni Trie

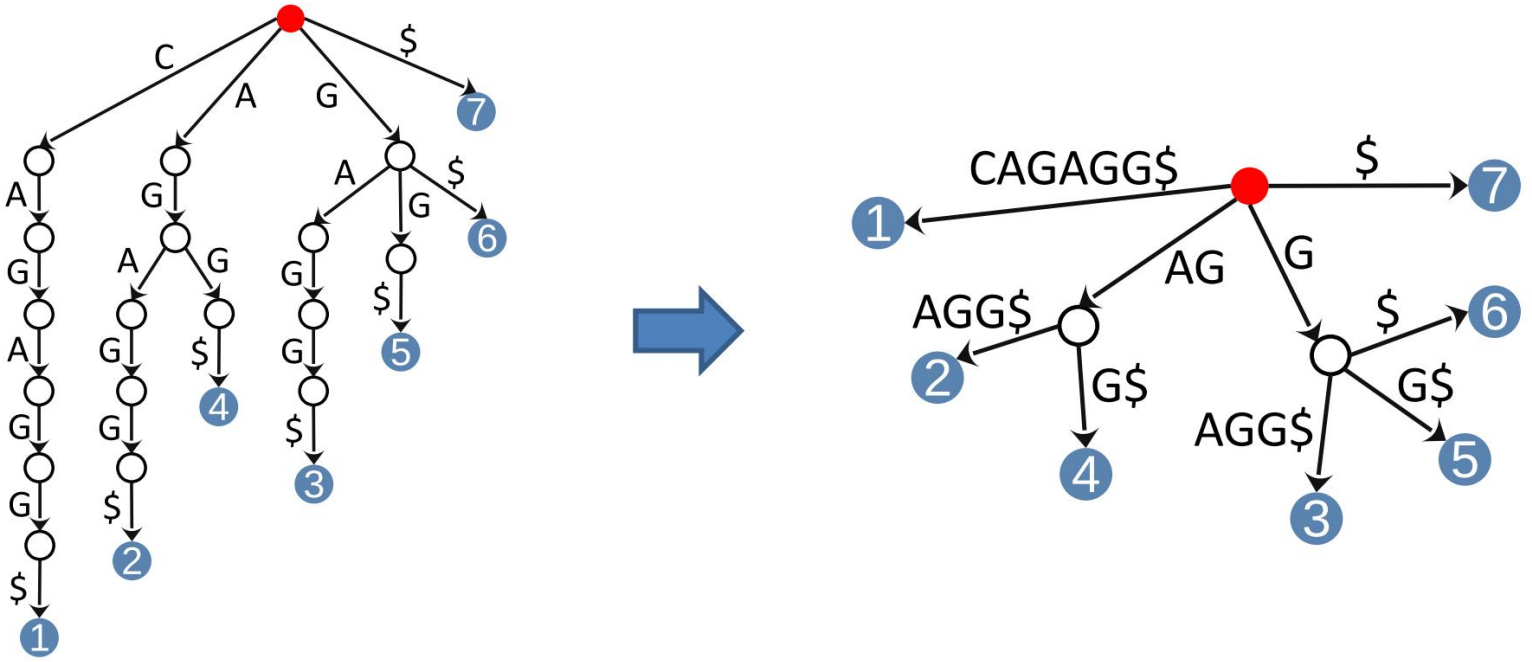
C	A	G	A	G	G	$
A	G	A	G	G	$	
G	A	G	G	$		
A	G	G	$			
G	G	$				
G	$					
$						



- Svaki list sufiksnog Trie-a sadrži redni broj tog sufiksa ili A_i

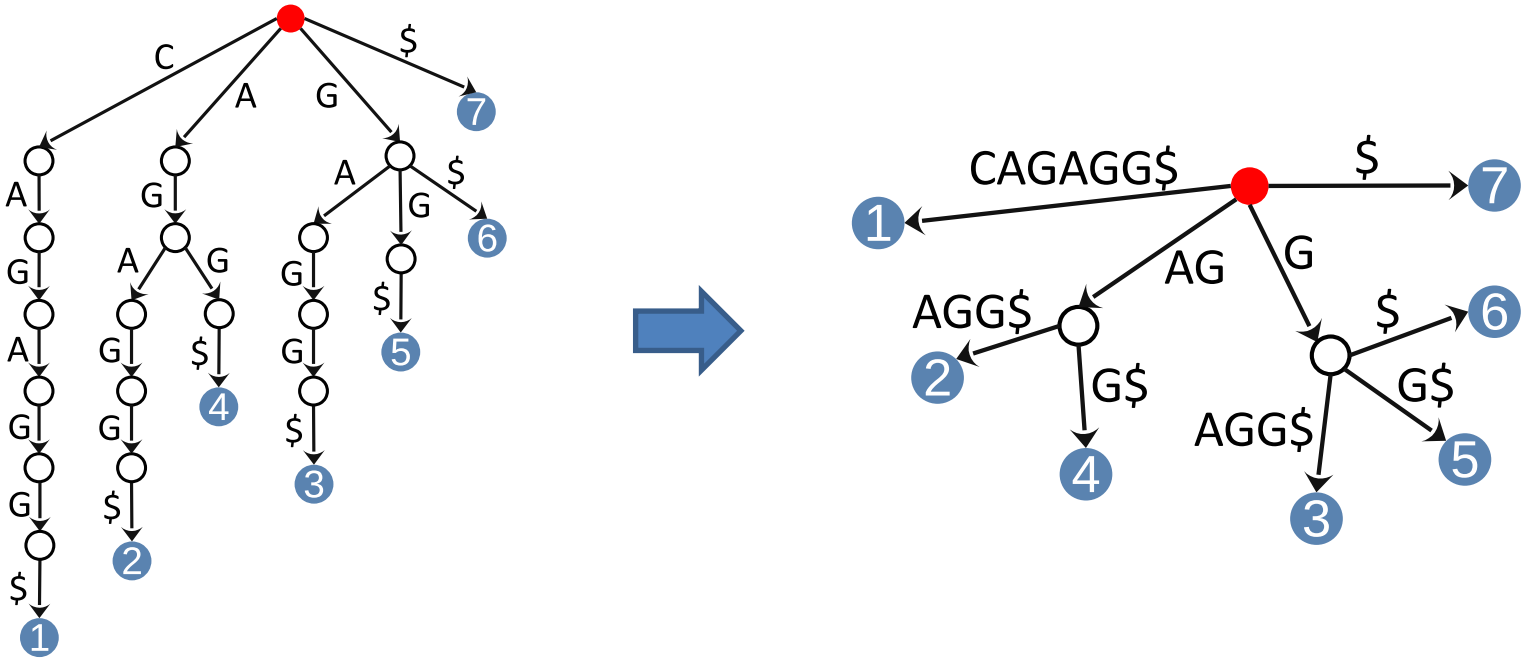
Suffix tree (1)

- 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**



Sufiksno stablo (1)

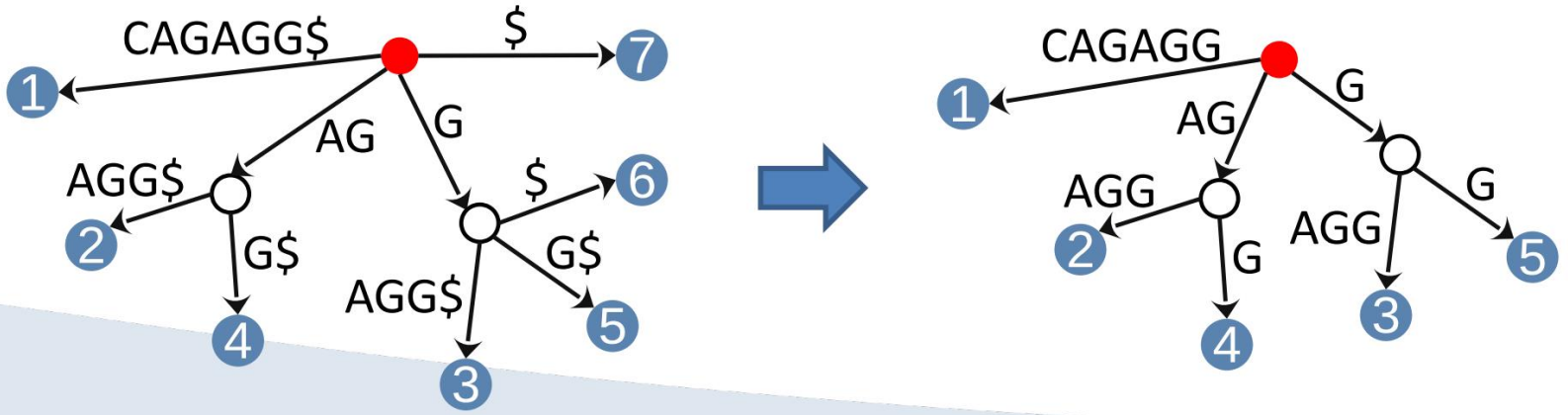
- Sufiksno stablo je kompresirani sufiksni Trie ili Patricia stablo svih sufiksa određenog znakovnog niza
- Ovo se još i naziva **eksplicitno sufiksno stablo**



Suffix tree (2)

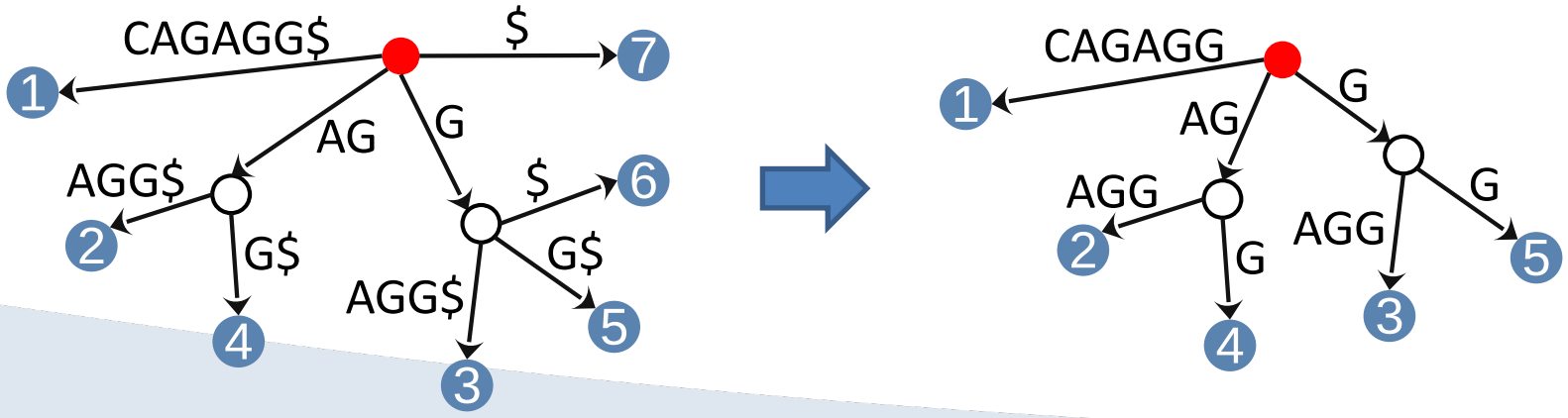
- **The implicit suffix tree** is obtained from the explicit one by the following transformation:

  1. Let's remove all NULL terminators ($) for the edges of the explicit suffix trees
  2. Let's remove all edges that remain with empty character strings, , including their subtrees
  3. 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



Sufiksno stablo (2)

- **Implicitno sufiksno stablo** dobiva se iz eksplicitnog sljedećom transformacijom:
  1. Uklonimo sve NULL terminatore ($) za bridova eksplicitnog sufiksnog stabla
  2. Uklonimo sve bridove koji su ostali s praznim znakovnim nizovima ϵ, uključujući i njihova podstabla
  3. Ponovno kompresiramo sufiksno stablo: svi čvorovi osim korijenskog, koji imaju manje od dvoje djece se uklanjaju, a njihove tranzicije konkateniraju



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%

Ukkonenov algoritam (1)

- Naivni pokušaj stvaranja sufiksnog stabla je $O(n^2)$
  - Ovo je nekoliko puta bilo poboljšano raznim algoritmima: McCreigh i Weiner
  - Ukkonen gradi svoj algoritam na Weinerovom algoritmu
  - Ukkonenov algoritam je *online* algoritam koji gradi sufiksno stablo znak po znak iz t
- Imamo implicitno sufiksno stablo nakon i koraka koje označimo S_i
  - Uzimamo znak $t[i+1]$ i izvršavamo $i+1$ fazu nadogradnje stabla
  - $i+1$ faza se izvodi u $i+1$ koraka
    - U svakom koraku dodajemo $t[i+1]$ na sve sufikse prefiksa $t[1, i]$
    - U $i+1$ koraku dodajemo $t[i+1]$ na korijenski čvor stabla
  - Rezultat je implicitno sufiksno stablo S_{i+1}

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	C
Step 2, $j=2$	B	A	B	A	B	C	
...							
Step 6, $j=6$	B	C					
Step 7	C						

Ukkonenov algoritam (2)

- Za $t = ABABABC\$$ imamo sljedeće faze i korake
  - Varijablom j definiramo sufiks prefiksa t[1,i], ili t[j,i]
- Prolazimo stablo za svaki $t[j,i]$ i dodajemo $t[i+1]$, a imamo tri slučaja:
  1. Kod prolaska $t[j,i]$ dolazimo do lista. Cijela vertikalna putanja predstavlja $t[j,i]$. U zadnjoj tranziciji prije lista samo dodajemo $t[i+1]$ na kraj tranzicije.

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	C
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.

Ukkonenov algoritam (3)

2. Prošli smo sve znakove $t[j,i]$ i stali na sredini tranzicije. Sljedeći znak u tranziciji **nije** $t[i+1]$. Dodajemo unutarnji čvor na stablo i razdvajamo tranziciju. Dodajemo list i tranziciju $t[i+1]$ od novo dodanog unutarnjeg čvora do novo dodanog lista. Ista operacija kao i kod Patricia stabla.
3. Prošli smo sve znakove $t[j,i]$ i stali na sredini tranzicije. Sljedeći znak u tranziciji **je** $t[i+1]$. Ne radimo ništa jer $t[j,i]t[i+1]$ je već sadržan u stablu.

Nakon što smo završili gradnju implicitnog sufiksnog stabla, pretvaramo ga u eksplicitno dodavanjem $ u sve tranzicije koje vode u listove stabla.

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 ÿ we need to start from both nodes, which effectively 7, 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 =

Ukkonenov algoritam (4)

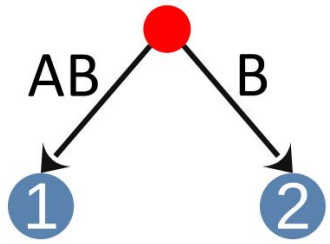
- **Sufiksne veze** dodatak su implicitnom sufiksnom stablu koji omogućavaju brže prolaženje stabla za $t[j, i]$
  - Imamo čvor n_1 u koji smo stigli prolaskom $t[j, i]$
  - Ako imamo neki drugi čvor n_2 do kojeg se stigne prolaskom $at[j, i]$, tada je n_2 spojen na n_1 sufiksnom vezom
  - Ovo samo znači da za sve faze $\geq i$ trebamo početi od od oba čvora n_1 i n_2, čime zapravo preskačemo sve korake koji uključuju podniz a
    - To znači da ne moramo startati s j=1, već odmah nakon podniza a
  - n_2 može biti i korijenski čvor kada je $a = \epsilon$ i $j = i$

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.

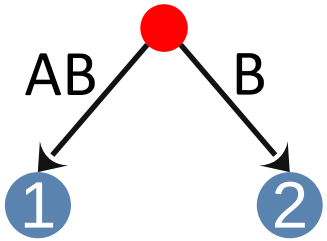
- **Stage 2 :** + 1 = 2, [+ 1] = •
  Step 1 : = 1, [,] = leads to The passing of us leaf 1, which is case 1. We add B to the last transition.
  - Step 2: [,] = to root . We are adding a new transition node and associated leaf.



Primjer Ukkonen (1)

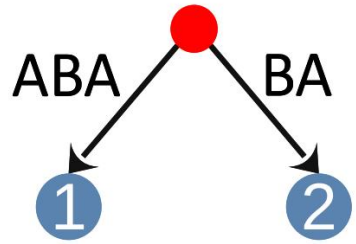
- Stvaramo implicitno sufiksno stablo korištenjem Ukkonenovog algoritma za $t = ABABABC\$$
- **Faza 1** : $i + 1 = 1, t[i + 1] = A$
  - Korak 1 : Imamo samo korijenski čvor. Dodajemo novu tranziciju i list.

- **Faza 2** : $i + 1 = 2, t[i + 1] = B$
  - Korak 1 : $j = 1, t[j, i] = A$. Prolazak nas dovodi do lista 1, što je slučaj 1. Dodajemo B na posljednju tranziciju.
  - Korak 2 : $t[j, i] = \epsilon$. Dodajemo novu tranziciju na korijenski čvor i pripadni list.



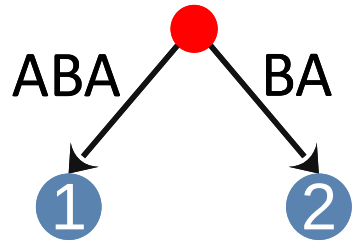
Example Ukkonen (2)

- **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.



Primjer Ukkonen (2)

- **Faza 3** : $i + 1 = 3, t[i + 1] = A$
  - Korak 1 : $j = 1, t[j, i] = AB$. Prolazak nas dovodi do lista 1, što je slučaj 1. Dodajemo A na posljednju tranziciju.
  - Korak 2 : $j = 2, t[j, i] = B$. Prolazak nas dovodi do lista 2, što je slučaj 1. Dodajemo A na posljednju tranziciju.
  - Korak 3: $t[j, i] = \epsilon$. S obzirom na ϵ, ostajemo u korijenskom čvoru, no prolazak $t[i + 1]$ nas vodi u tranziciju, što je slučaj 3. Ne radimo ništa.

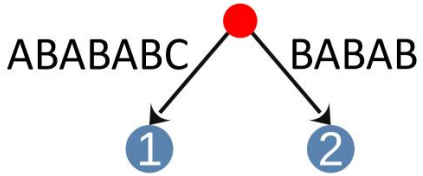


Example Ukkonen (3)

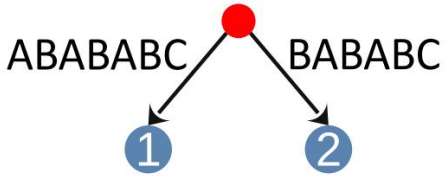
- We skip a series of the same stages

- **Stage 7 :** + 1 = 7, [+ 1] = • Step 1 : = 1, [,]

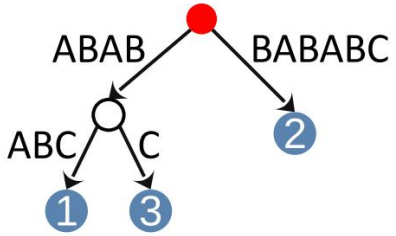
  = . The traversal brings us to sheet 1, which is case 1. We add C to the last transition.



  - Step 2 : = 2, [,] = . Traversal brings us to sheet 2, which is case 1. We add C to the last transition.



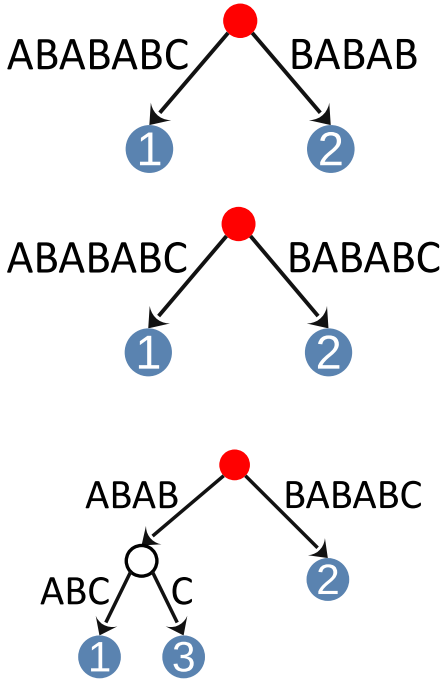
  - Step 3 : = 3, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for



.

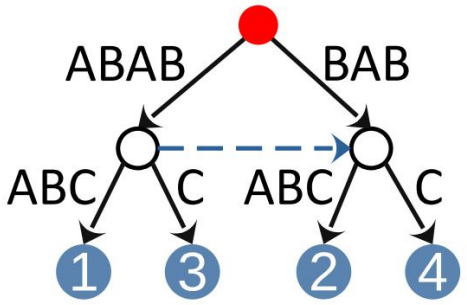
Primjer Ukkonen (3)

- Preskačemo niz istih faza
- **Faza 7** : $i + 1 = 7, t[i + 1] = C$
  - Korak 1 : $j = 1, t[j, i] = ABABAB$. Prolazak nas dovodi do lista 1, što je slučaj 1. Dodajemo C na posljednju tranziciju.
  - Korak 2 : $j = 2, t[j, i] = BABAB$. Prolazak nas dovodi do lista 2, što je slučaj 1. Dodajemo C na posljednju tranziciju.
  - Korak 3 : $j = 3, t[j, i] = ABAB$. Prelazak nas vodi na sredinu tranzicije, a sljedeći znak nije C, što je slučaj 2. Razdvajamo puteve za $ABABABC$ i $ABABC$.

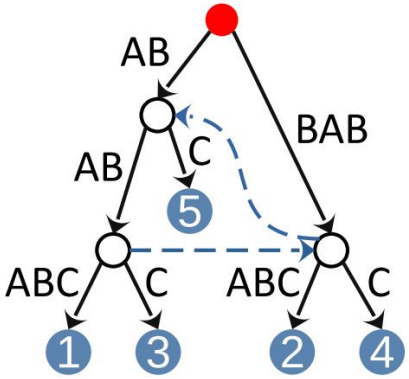


Example Ukkonen (4)

- 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 two internal nodes. We connect with a suffix link.

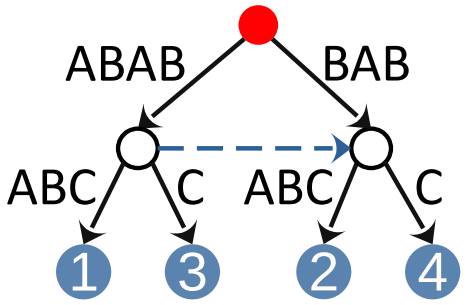


- Step 5 : = 5, [,] = . The transition takes us to the middle of the transition, and the next character is not C, which is case 2. We separate the paths for { , } and . In this step we have two internal nodes = i . We connect with a suffix connection.

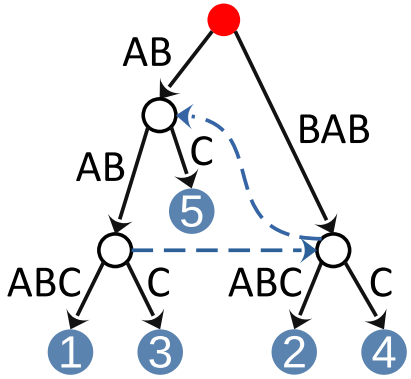


Primjer Ukkonen (4)

- Korak 4 : $j = 4, t[j,i] = BAB$. Prelazak nas vodi na sredinu tranzicije, a sljedeći znak nije C, što je slučaj 2. Razdvajamo puteve za $BABABC$ i $BABC$. U ovom koraku imamo dva unutarnja čvora $n_1 = BAB$ i $n_2 = ABAB$. Spajamo sufiksnom vezom.

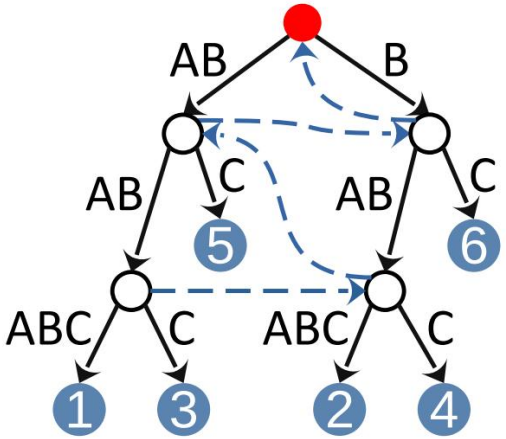


- Korak 5 : $j = 5, t[j,i] = AB$. Prelazak nas vodi na sredinu tranzicije, a sljedeći znak nije C, što je slučaj 2. Razdvajamo puteve za $\{ABABABC, ABABC\}$ i ABC. U ovom koraku imamo dva unutarnja čvora $n_1 = AB$ i $n_2 = BAB$. Spajamo sufiksnom vezom.

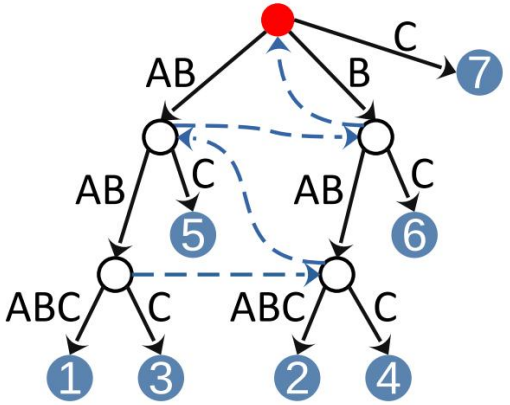


Example Ukkonen (5)

- 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 with a suffix $\bar{x}$ to the last node. We also add a suffix link between the = node and the root node.

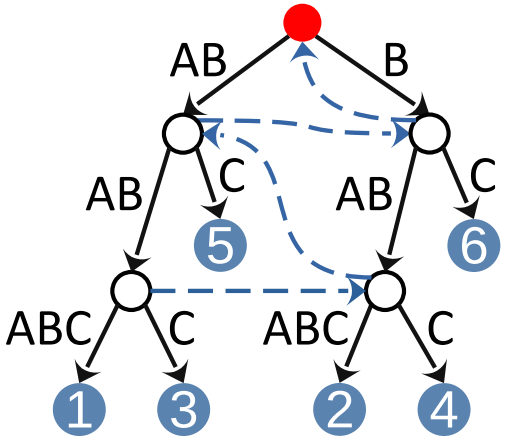


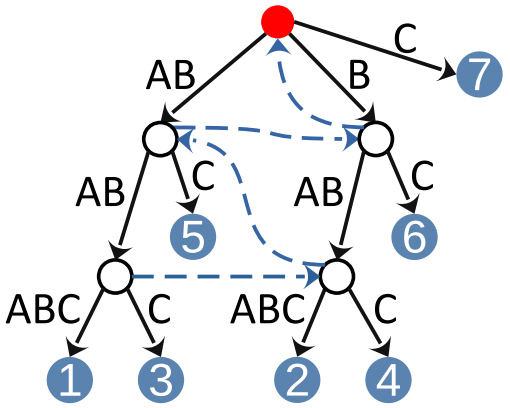
- Step 7: [,] = . Since no transition starts with C, we add a new sheet and a new transition.



Primjer Ukkonen (5)

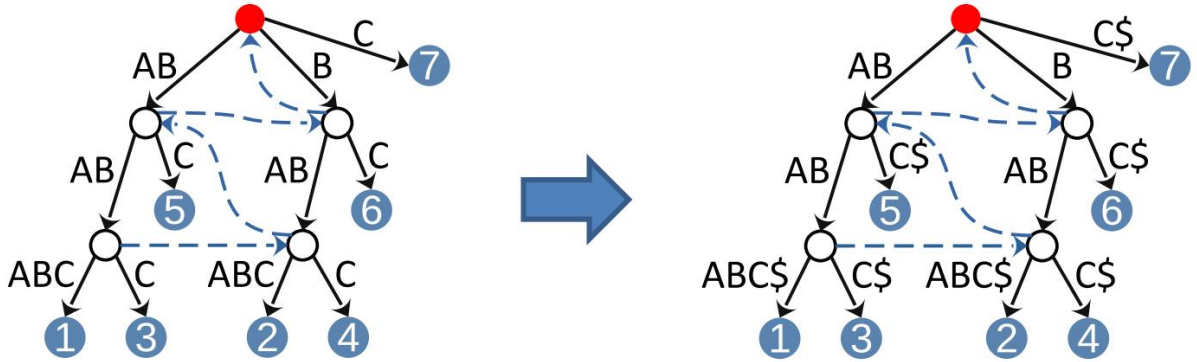
- Korak 6 : $j = 6, t[j,i] = B$. Prelazak nas vodi na sredinu tranzicije, a sljedeći znak nije C, što je slučaj 2. Razdvajamo puteve za $\{BABABC, BABC\}$ i BC. U ovom koraku imamo dva unutarnja čvora $n_1 = B$ i $n_2 = AB$. Spajamo sufiksnom vezom. Dodajemo i sufiksnu vezu između čvora $n_2 = B$ i korijenskog čvora.
- Korak 7 : $t[j,i] = \epsilon$. S obzirom da niti jedna tranzicija ne počinje C, dodajemo novi list i novu tranziciju.





Ukkonen's algorithm (5)

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



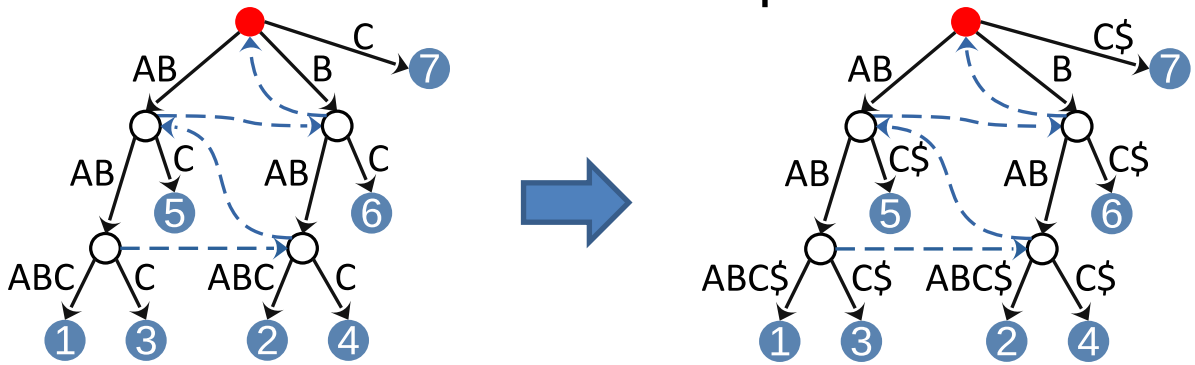
- Let's say that we now add the eighth character D (+
  1 = 8)

- 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

Ukkonenov algoritam (5)

- Sada možemo transformirati implicitno sufiksno stablo u eksplicitno

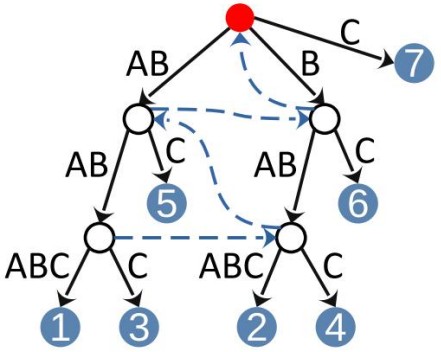


- Recimo da na ovo sufiksno stablo sada dodajemo osmi znak D ($i + 1 = 8$)
- U trenutku kada prolazimo prefiks $t[1,7] = ABABABC$, nailazimo na sufiksne veze
  - Kod $ABAB$ imamo sufiksne veze na 4 dodatna čvora
  - Kod prelaska imamo 5 čvorova od kojih krećemo sa tranzicijama
  - Znamo da možemo preskočiti na $t[6,7]$ jer smo sufikse $BAB, AB, B\ i\ \epsilon$ imali u prolasku $ABAB$

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

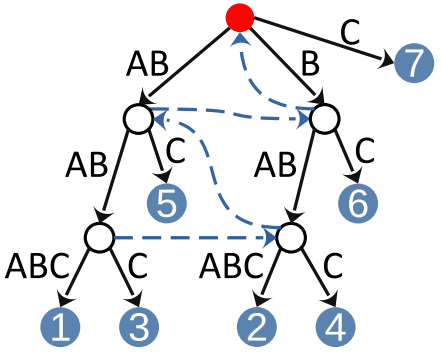


- This reduces both the number of steps and the number of characters passed in a step • Reducing the number of steps - we actually parallelize the execution • The real gain is in reducing the number of characters passed in a certain step • In the previous example, we omitted crossing 6 characters

Ukkonenov algoritam (6)

- Na t[6,7] krećemo ispočetka jer u našem skupu čvorova nemamo korijenski čvor

$t[1,7]$	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



- Time smanjujemo i broj koraka, a i broj prijeđenih znakova u koraku
  - Smanjivanje broja koraka - zapravo paraleliziramo izvođenje
  - Pravi dobitak je u smanjenju broja prijeđenih znakova u određenom koraku
  - U prethodnom primjeru smo izostavili prelazak 6 znakova

Pitanja ?

Pitanja ?