Suffix Tree. Ukkonen. Suffix array

Seminar 8.

What is Naive Suffix Tree?

I. Acquire all suffixes

II. Start building from an entire word

III. Pass new suffix

    a. Find the greatest common start

    b. Check the match

    c. Split the common part -> Make a new parent node

    d. Add the rest of the suffix as a new child edge

What is Naive Suffix Tree?

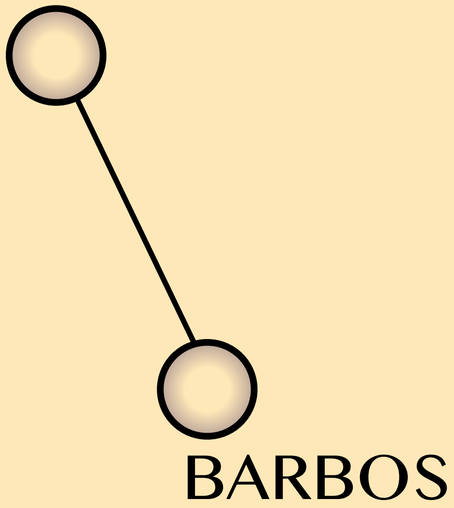
BARBOS
ARBOS
RBOS
BOS
OS
S

I. Acquire all suffixes
II. Start building from an entire word
III. Pass new suffix
    a. Find the greatest common start
    b. Check the match
    c. Split the common part -> Make a new parent node
    d. Add the rest of the suffix as a new child edge

What is Naive Suffix Tree?

ARBOS
RBOS
BOS
OS
S

I. Acquire all suffixes
II. Start building from an entire word
III. Pass new suffix
    a. Find the greatest common start
    b. Check the match
    c. Split the common part -> Make a new parent node
    d. Add the rest of the suffix as a new child edge



What is Naive Suffix Tree?

I. Acquire all suffixes

II. Start building from an entire word

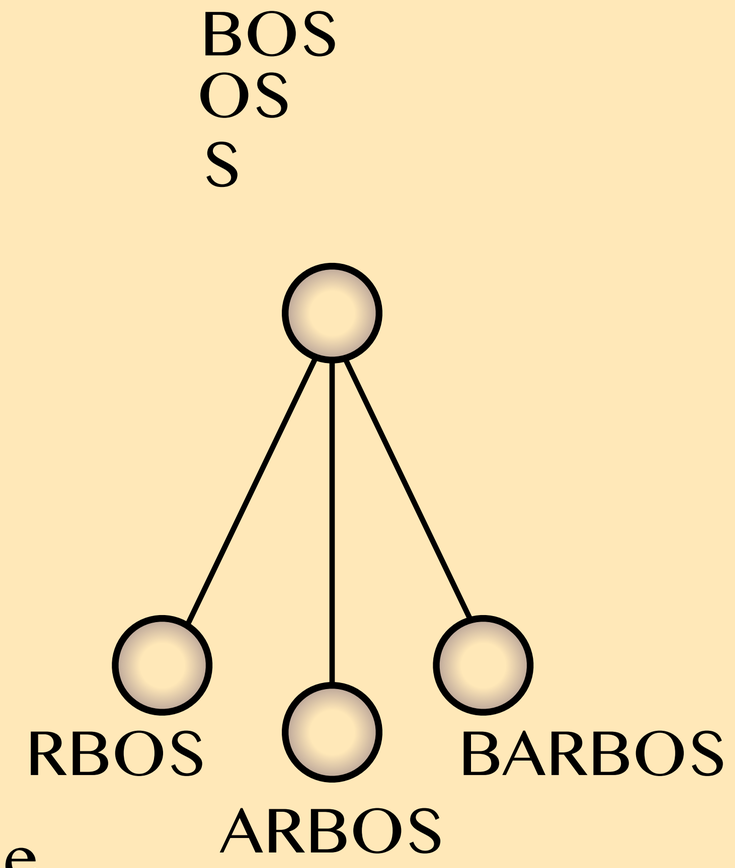
III. Pass new suffix

   a. Find the greatest common start

   b. Check the match

   c. Split the common part -> Make a new parent node

   d. Add the rest of the suffix as a new child edge



What is Naive Suffix Tree?

I. Acquire all suffixes

II. Start building from an entire word

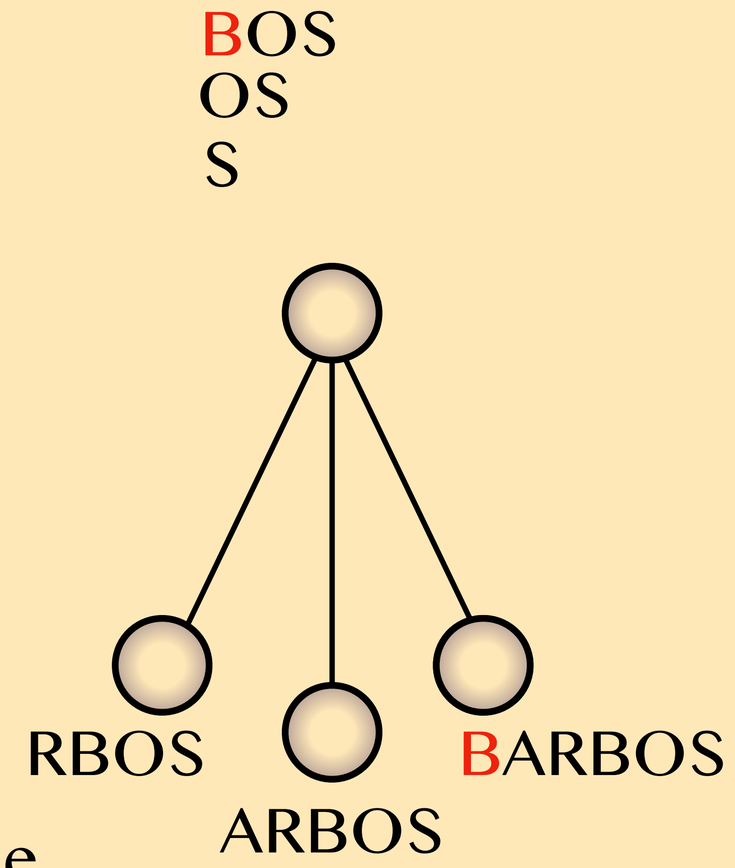
III. Pass new suffix

   a. Find the greatest common start

   b. Check the match

   c. Split the common part -> Make a new parent node

   d. Add the rest of the suffix as a new child edge



What is Naive Suffix Tree?

I. Acquire all suffixes

II. Start building from an entire word

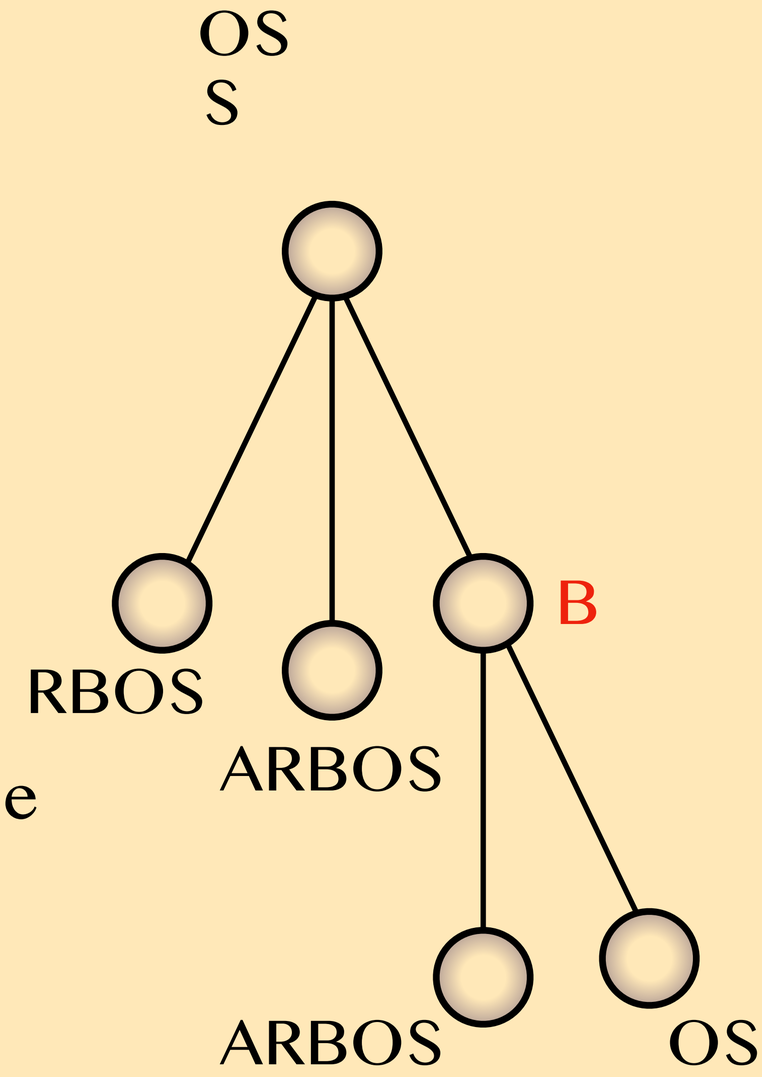
III. Pass new suffix

   a. Find the greatest common start

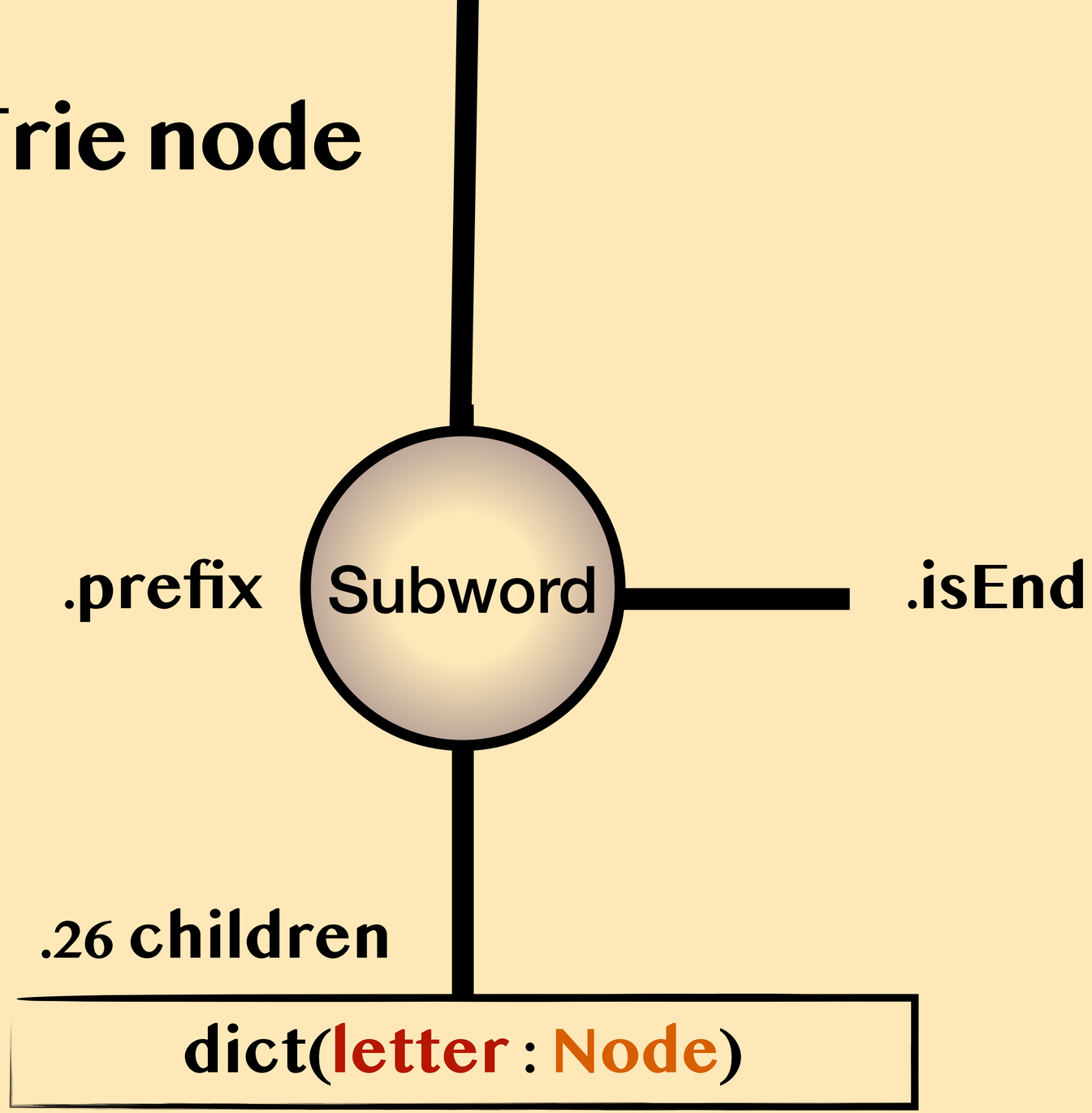
   b. Check the match

   c. Split the common part -> Make a new parent node

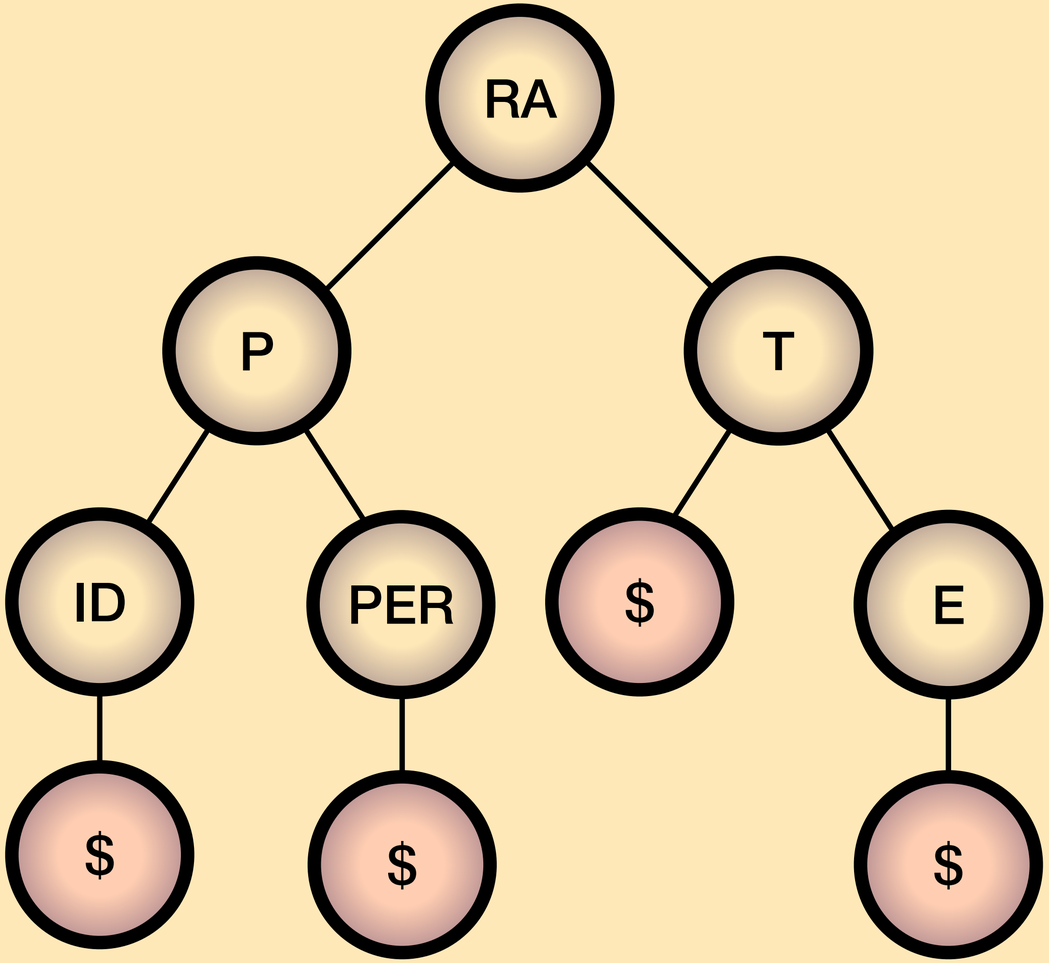
   d. Add the rest of the suffix as a new child edge



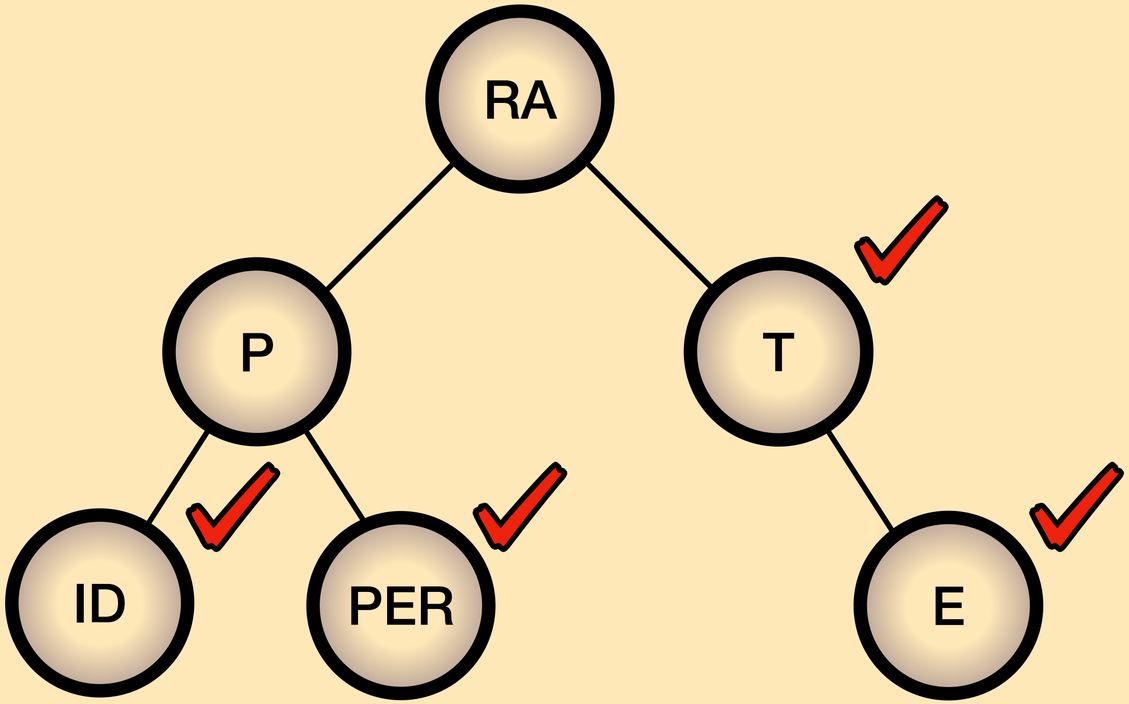
Compact Trie node



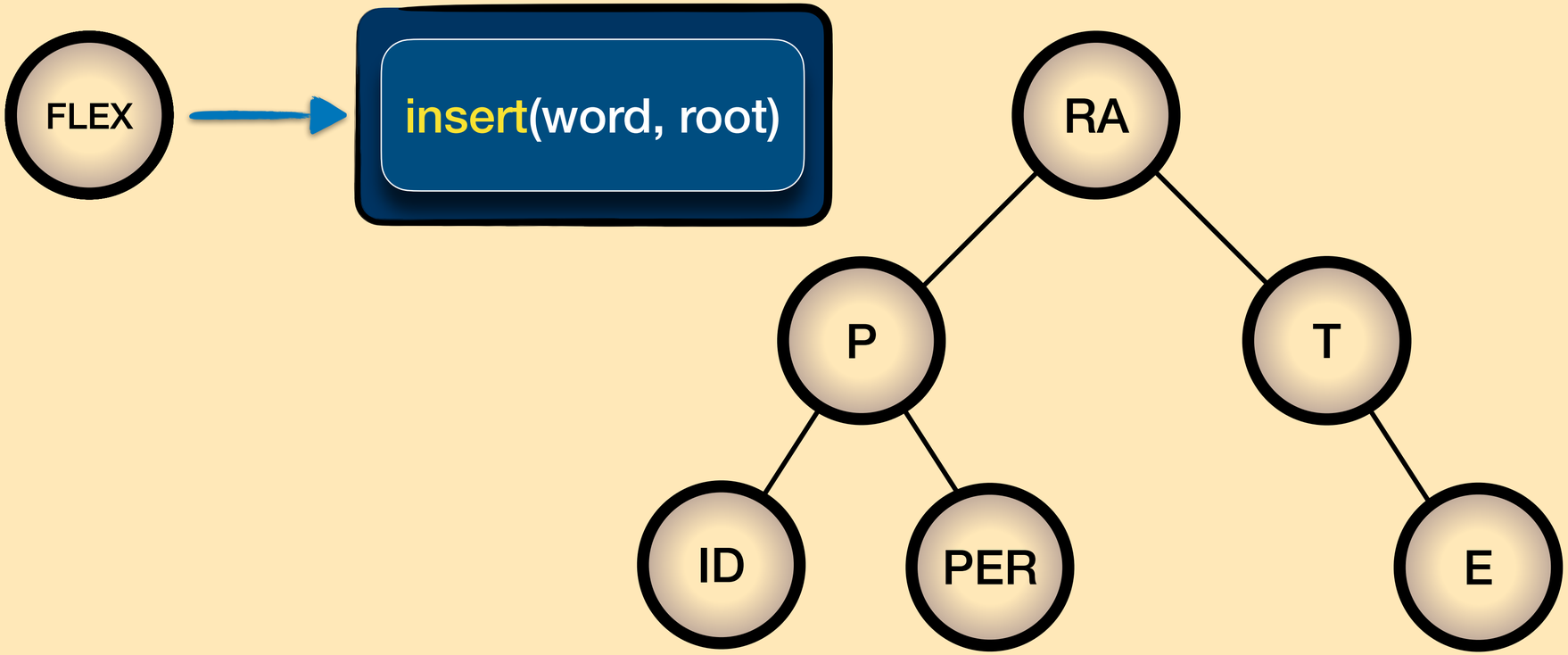
Compact Trie



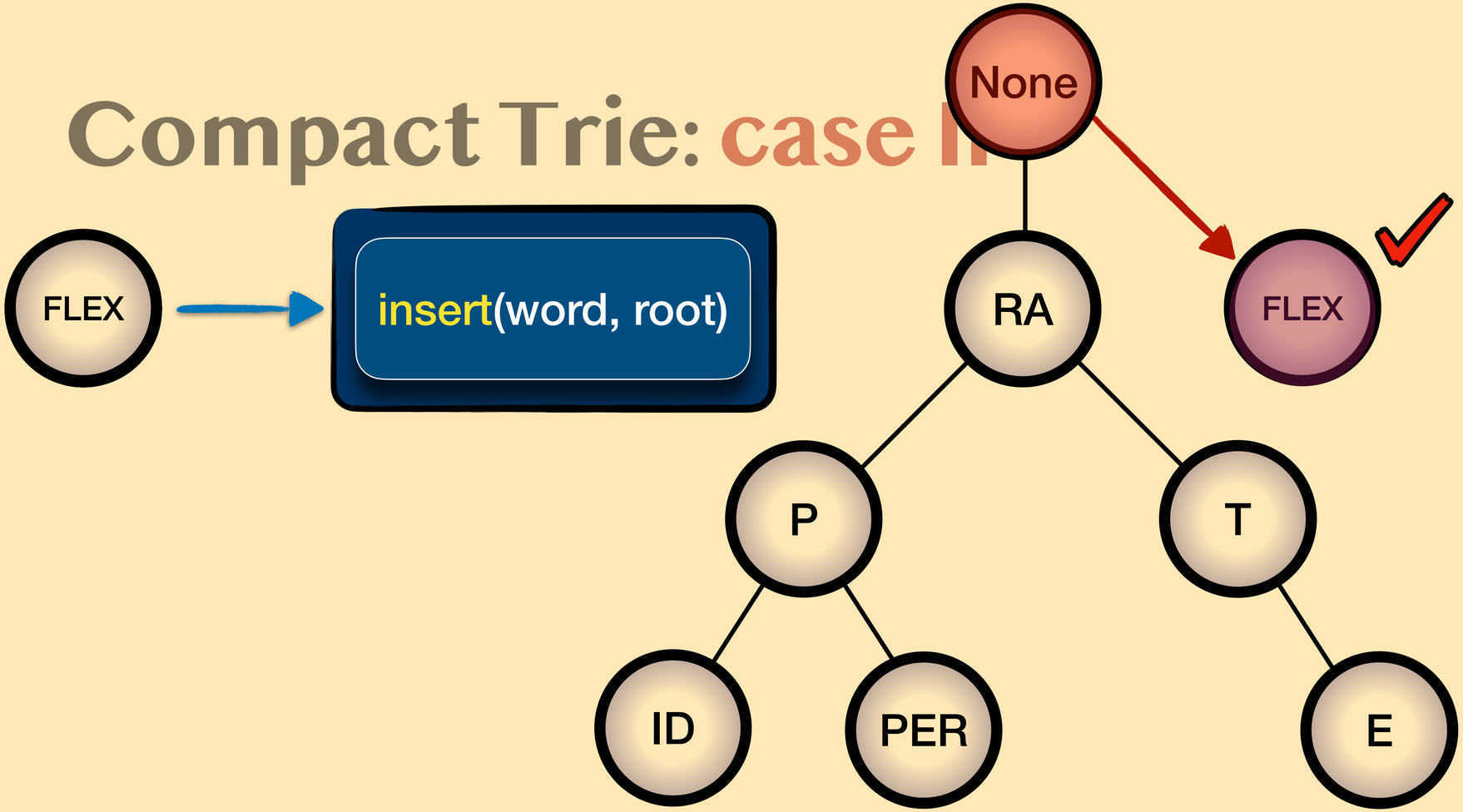
Compact Trie



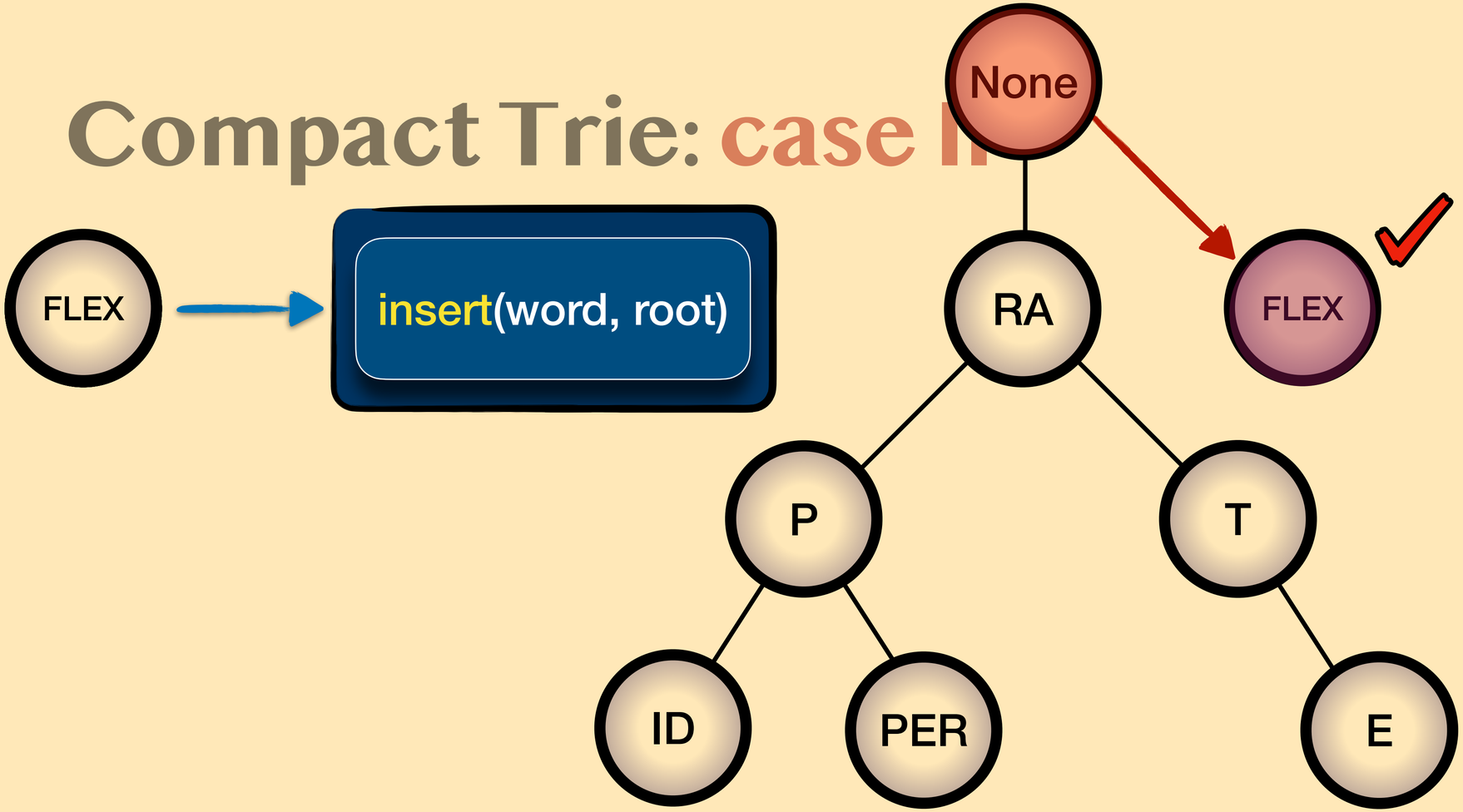
Compact Trie: case II



Compact Trie: case II

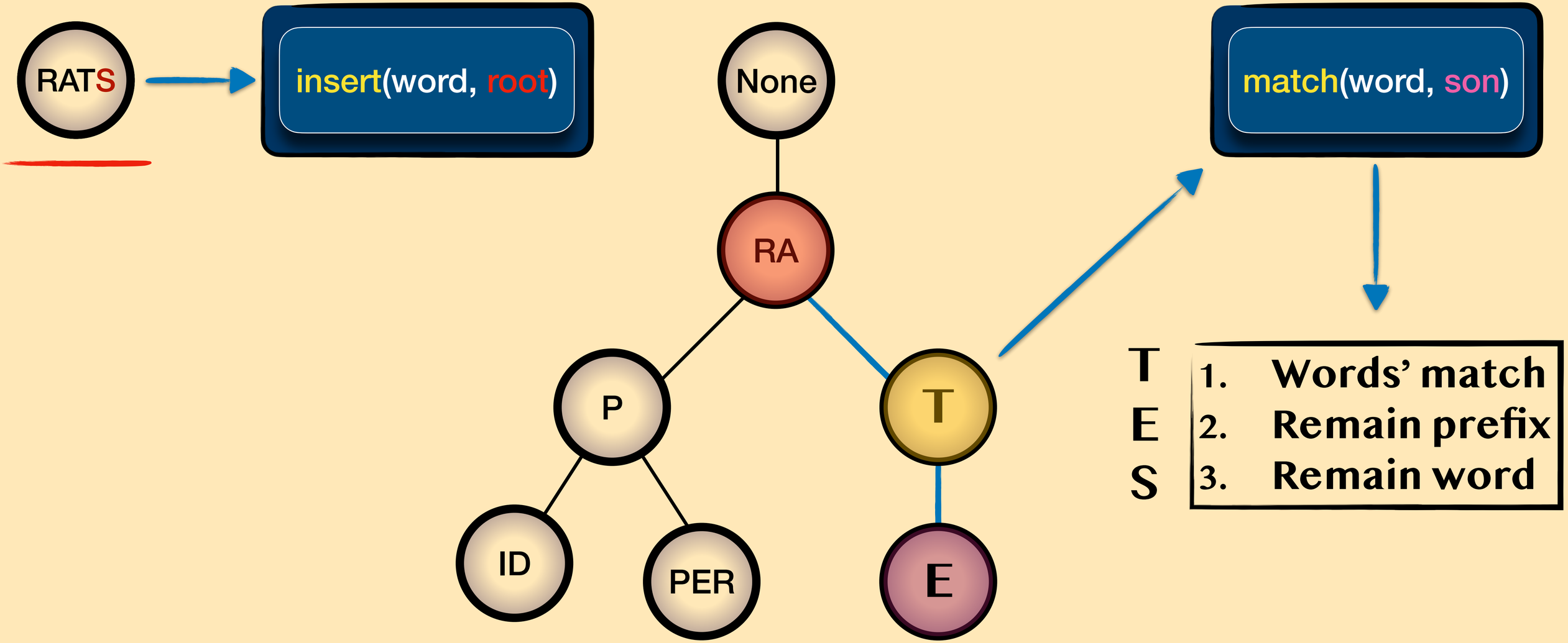


Compact Trie: case I

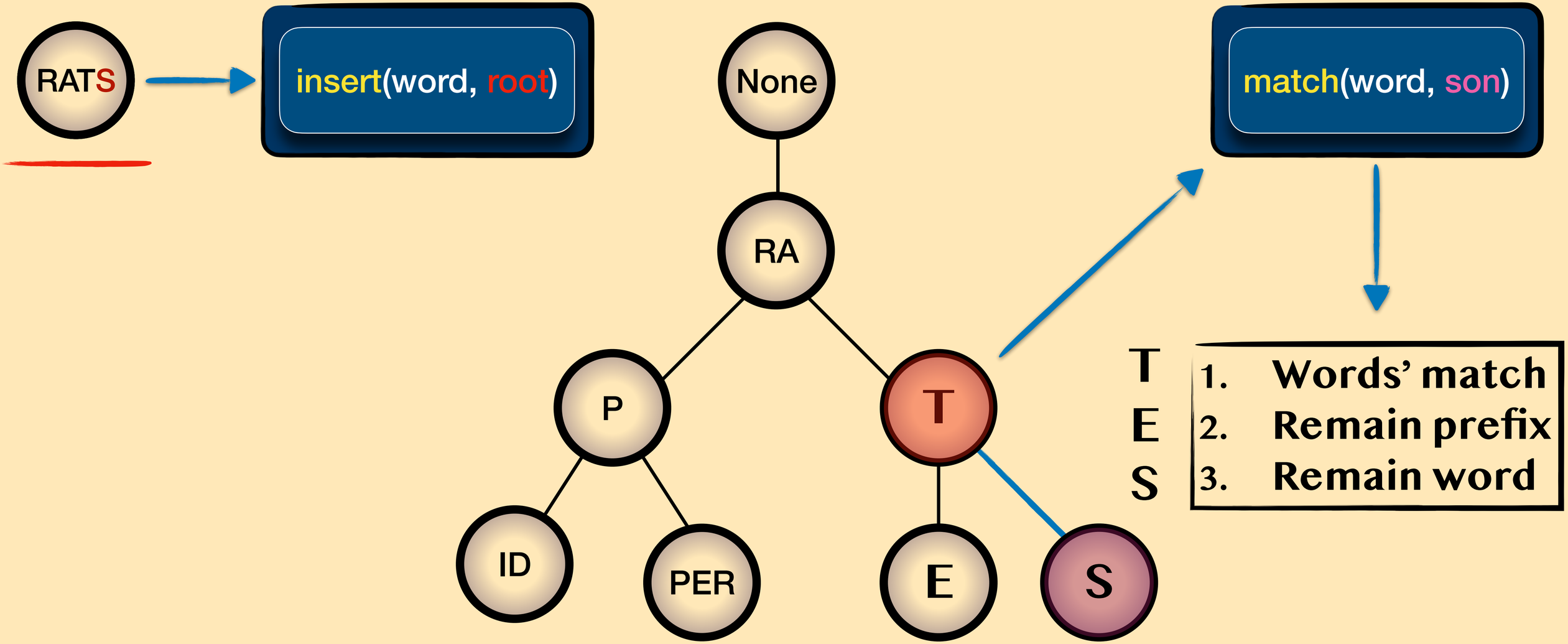


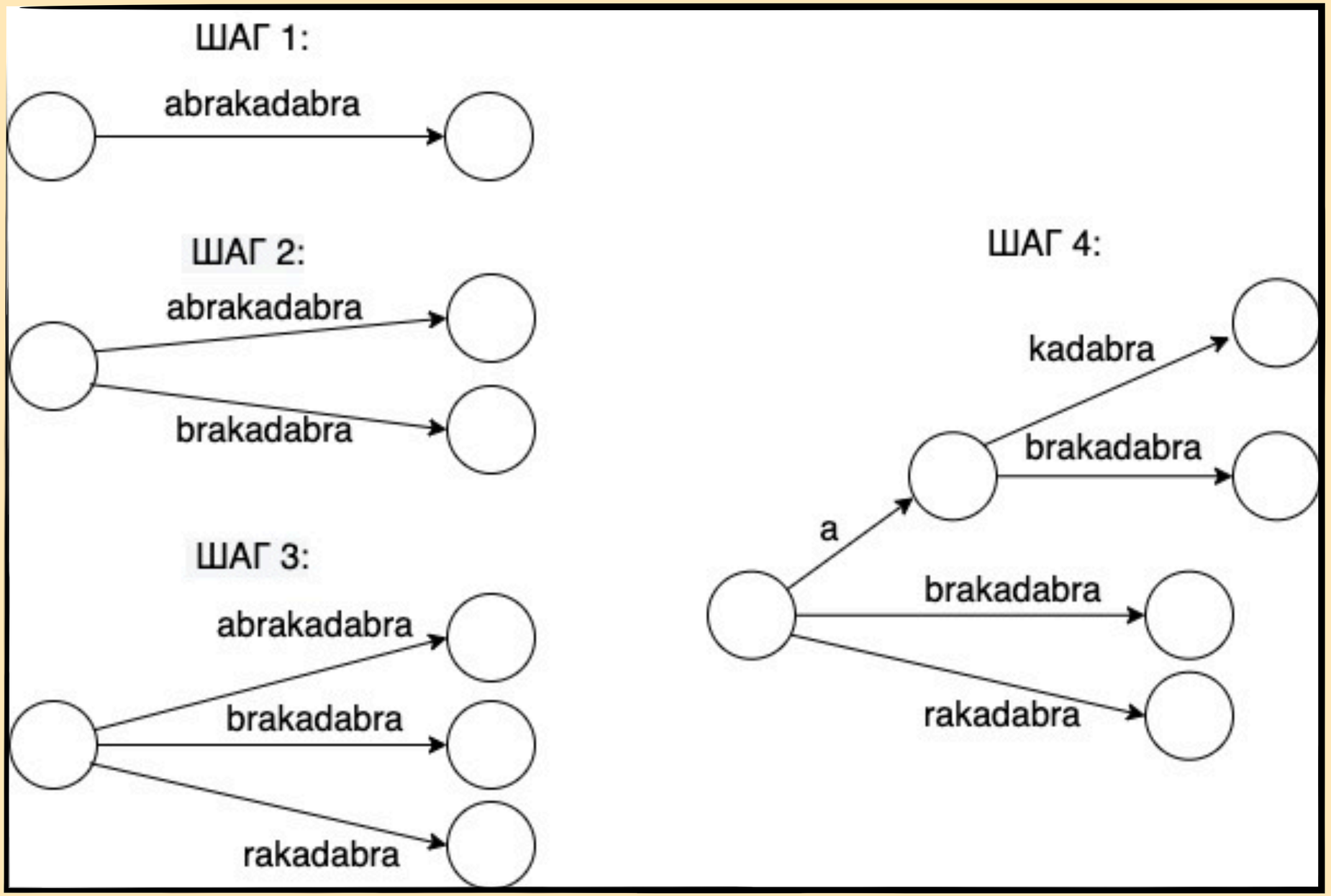
Check existence of first letter

Compact Trie: case IV.II



Compact Trie: case IV.II




ШАГ 1:
abrakadabra
ШАГ 2:
abrakadabra
brakadabra
ШАГ 3:
abrakadabra
brakadabra
rakadabra
ШАГ 4:
kadabra
brakadabra
a
brakadabra
rakadabra

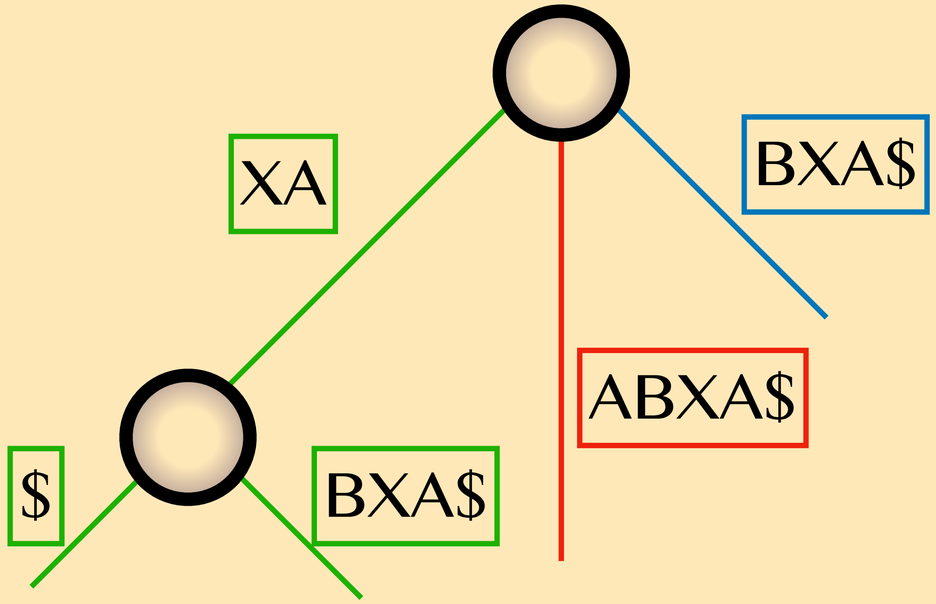
Ukkonen…

get ready 🤕

Ukkonen…

XABXA$
ABXA$
BXA$
XA$
A$
$

explicit

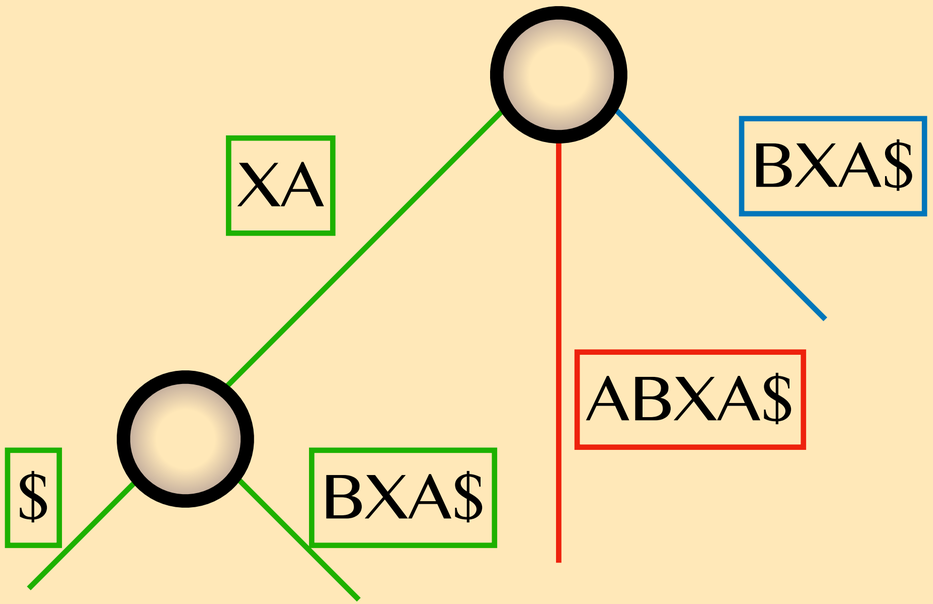


implicit

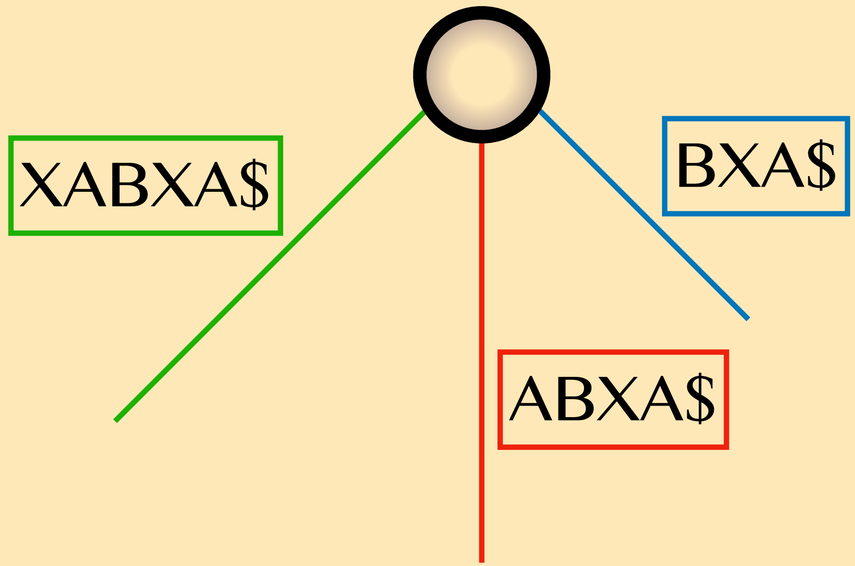
Ukkonen…

XABXA$
ABXA$
BXA$
XA$
A$
$

explicit



implicit



Ukkonen...

Intro

I. Iteratively construct trees T_i for **prefixes** $S[0:i]$

II. Each new **phase** constructs an **implicit** tree based on previous T_{i-1}

III. For each phase we have i **extensions** - the suffixes $S[j:i]$ we add to T_i

Ukkonen...

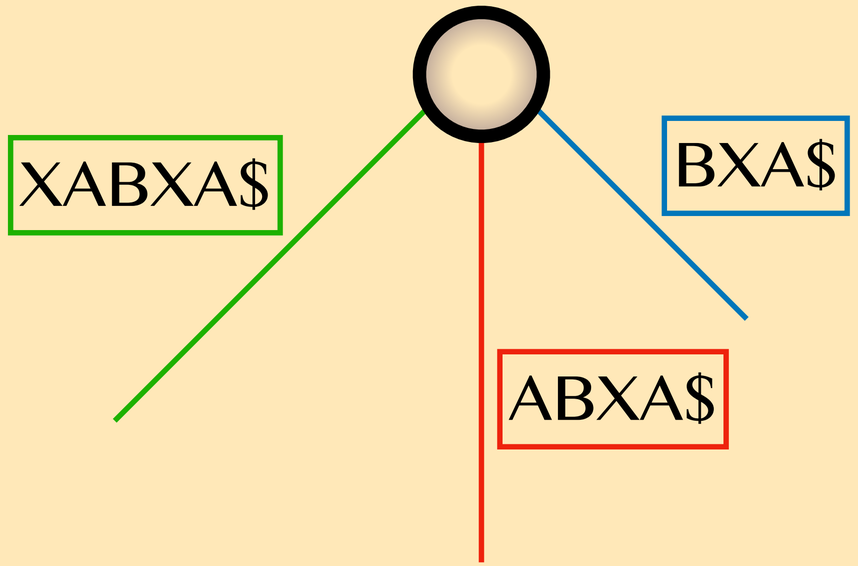
Intro

I. Iteratively construct trees T_i for **prefixes** $S[0:i]$

II. Each new **phase** constructs an **implicit** tree based on previous T_{i-1}

III. For each phase we have i **extensions** - the suffixes $S[j:i]$ we add to T_i

XABXA$
ABXA$
BXA$
XA$
A$
$



Ukkonen…

3 RULES OF EXTENSION!

I. ADDITIVE: if comparison with path $S[j:i]$ ends at **leaf edge** -> add new letter to $S[i+1]$ edge

II. CREATIVE: if it ends not at **leaf edge** - more letters on the path & **mismatch** occurred:

   a. Create a new leaf edge at the place of mismatch

   b. Create a new internal node in case of partial match with path

III. STOPPING: if it ends not at **leaf edge** & **no mismatch** - suffix is in edge - do nothing

Ukkonen…

3 OBSERVATIONS

I. Why Rule III Stops?

- because if path $S[j:i]$ continues with character $S[i+1]$ ==> paths $S[j+1:i], S[j+2:i], \ldots, S[i:i]$ will also continue with $S[i+1]$

Ukkonen...

3 OBSERVATIONS

II. Once a leaf - Always a leaf

- Imagine creating a leaf edge on j**th extension** (suffix from jth letter)
- Each next phase will be **adding** new letter $S[i+1]$ during j**th extension**
- This is regulated by ADDITIVE Rule

Ukkonen…

3 OBSERVATIONS

II. Global Ending

- Imagine several leaf edges
- Each phase they will be all incremented by letter $S[i+1]$
- Hence, we can just keep the global index & add to it each phase

Ukkonen…

So, how does it go?

I. Phase 1 starts with Rule 2, all other phases start with Rule 1

II. Any phase ends with either Rule 2 or Rule 3

III. Each phase has j extensions:
   a. first p extensions will follow Rule 1
   b. next q extensions will follow Rule 2
   c. next r extensions will follow Rule 3

IV. At the end of any phase i, there will be p+q leaf edges and next phase i+1 will go through Rule 1 for the first p+q extensions

Ukkonen…

Avoid launching from the root

I. Active point - letter, where we finished j**th extension** of i**th** phase and where we should start j**th extension** of $i+1$**th**

II. It consists of:
   a. **activeNode** - closest node to end-letter
   b. **activeEdge** - first edge's letter, coming from **activeNode** towards letter
   c. **activeLength** - how many steps to make along edge towards letter

Ukkonen…

Avoid launching from the root

I. Active point - letter, where we finished ***j*th extension** of ***i*th** phase and where we should start ***j*th extension** of $i+1$**th**

II. How to change between extensions?
   a. **After Rule#3** - Increment activeLength
   b. **During movement** - in case of meeting some internal nodes, which are closer to the end-letter
   c. **Zero length** - if current activeLength equals zero, change activeEdge to the new extension's letter
   d. **After Rule#2 - …**
      i) activeNode is not root - follow the suffix link!
      ii) activeNode is root & activeLength > 0 ==> decrement activeLength by 1 & change activeEdge to [**i - remainingSuffixCount + 1**]

Ukkonen…

Last words

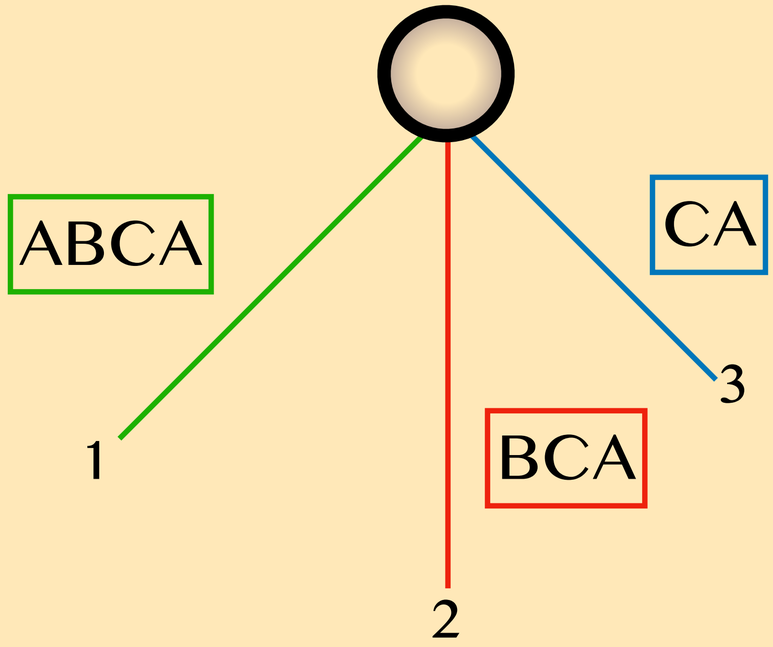
I. **remainingSuffixCount** - check how many suffixes to be added
   - If greater than zero ==> several suffixes are implicit

II. Suffix Links
   - pointer from edge = ‘_A’ to edge ‘A’, where A - sequence of letters
   - Internal node, created during j**th extension**, will be a suffix link from $j-1$**th extension**

III. Skip nodes

Ukkonen…

ABCABXABCD $



Phase: #4

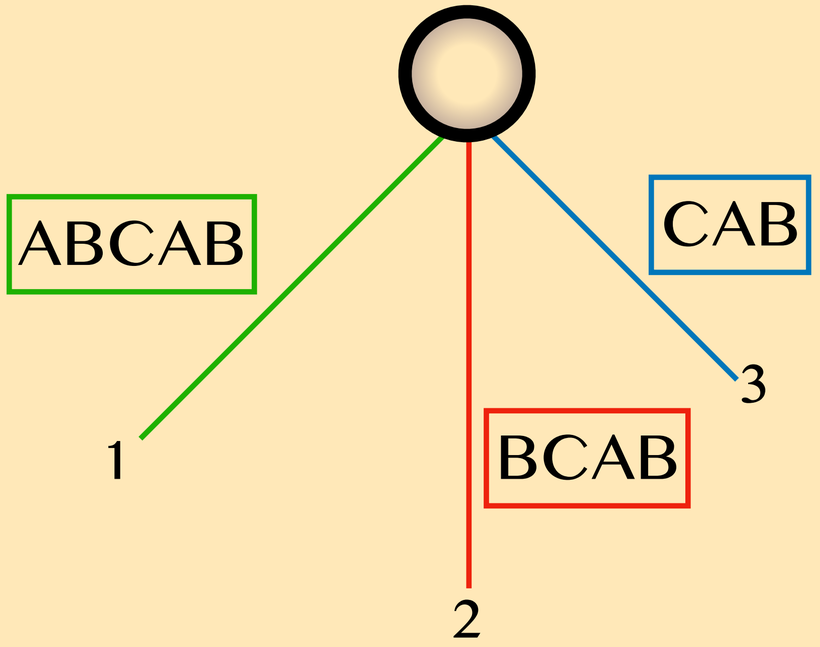
remSuffixCount = 1

activePoint = (root, ‘a’, 1)

end = 4

Ukkonen…

ABCABXABCD $



Phase: #5

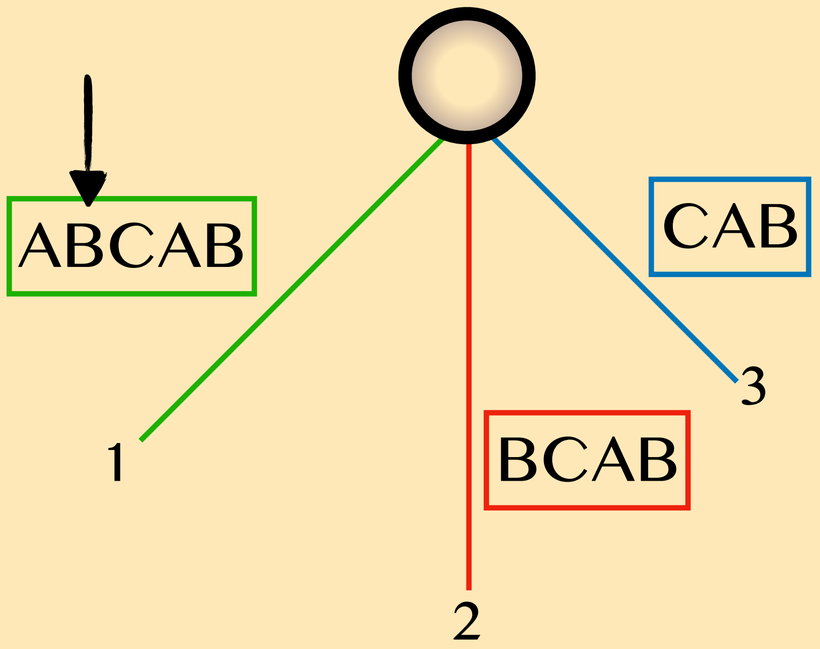
remSuffixCount = 2

activePoint = (root, ‘a’, 2)

end = 5

Ukkonen…

ABCABXABCD $



Phase: #6

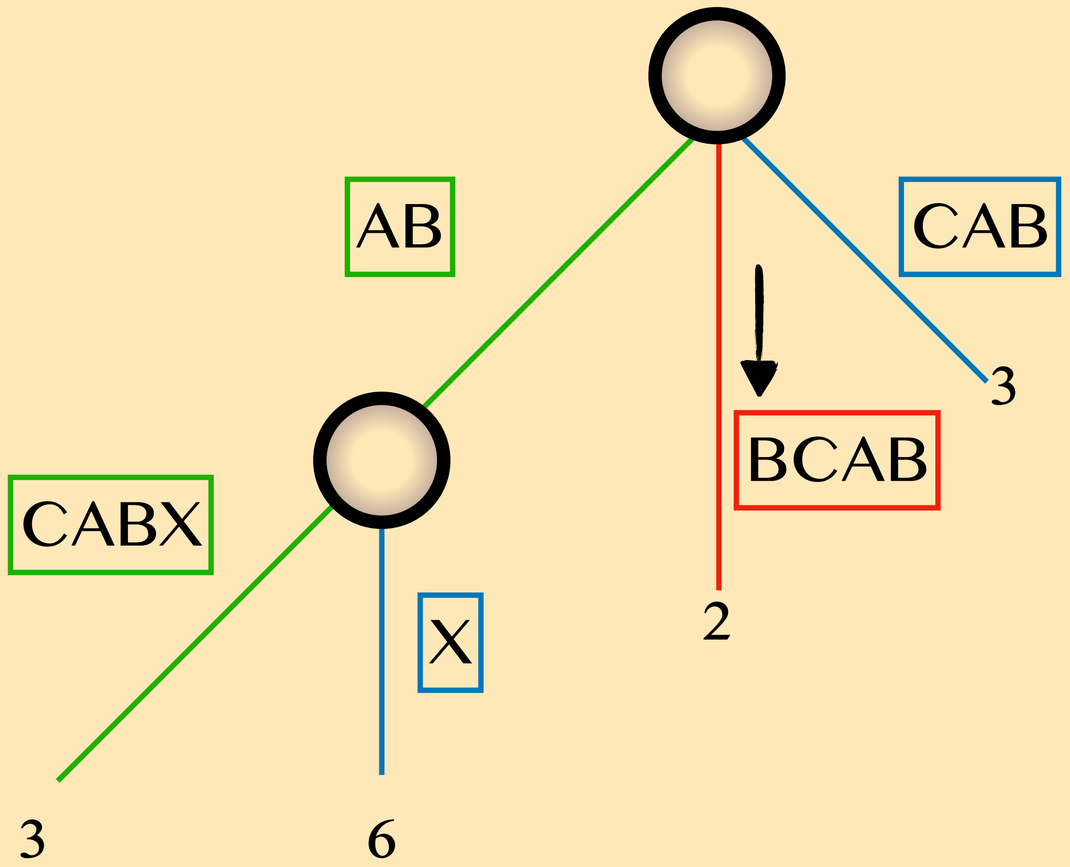
remSuffixCount = 2

activePoint = (root, ‘a’, 2)

end = 5

Ukkonen...

ABCABXABCD $



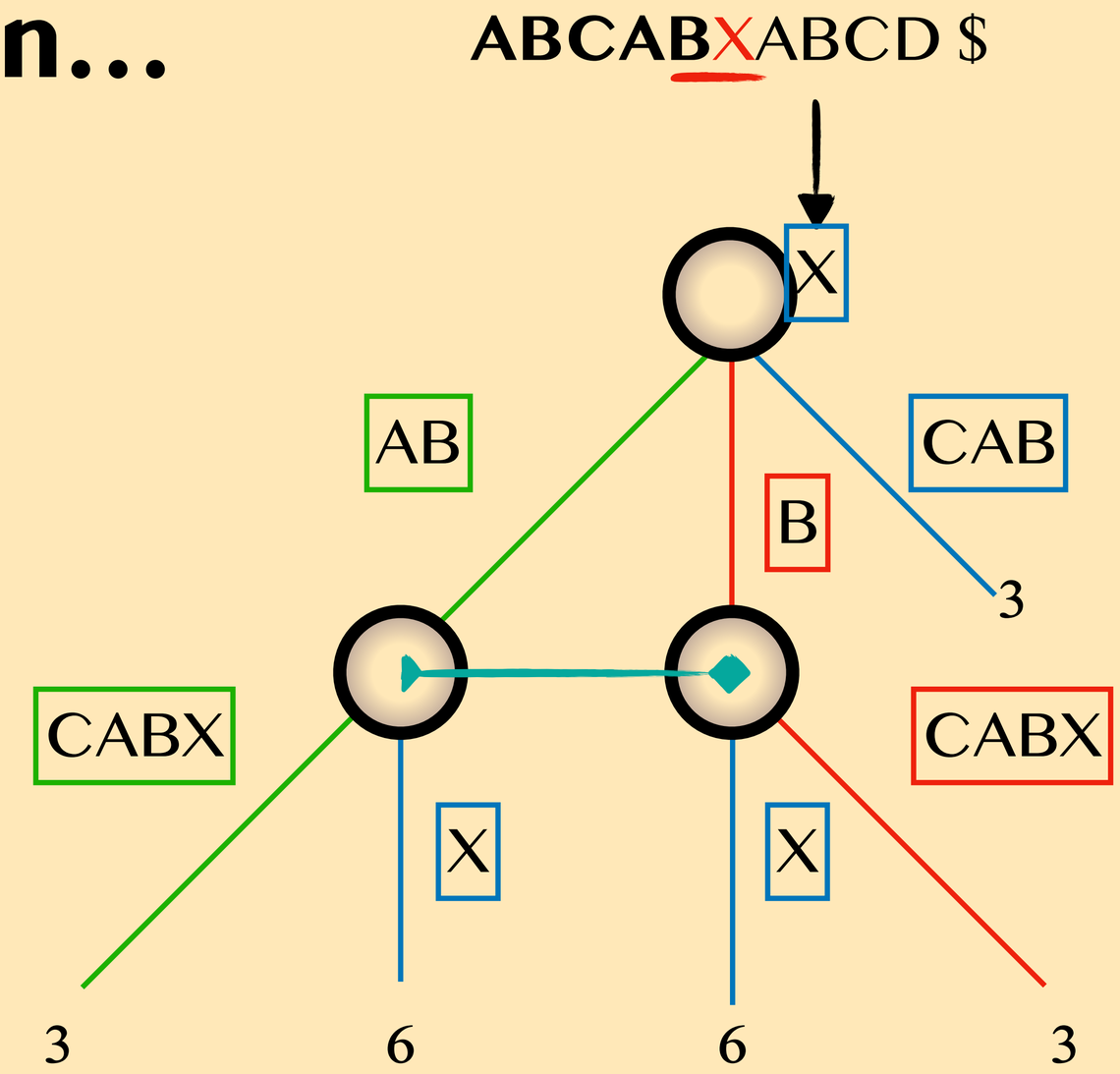
Phase: #6.1

remSuffixCount = 2

activePoint = (root, ‘b’, 1)

end = 6

Ukkonen…



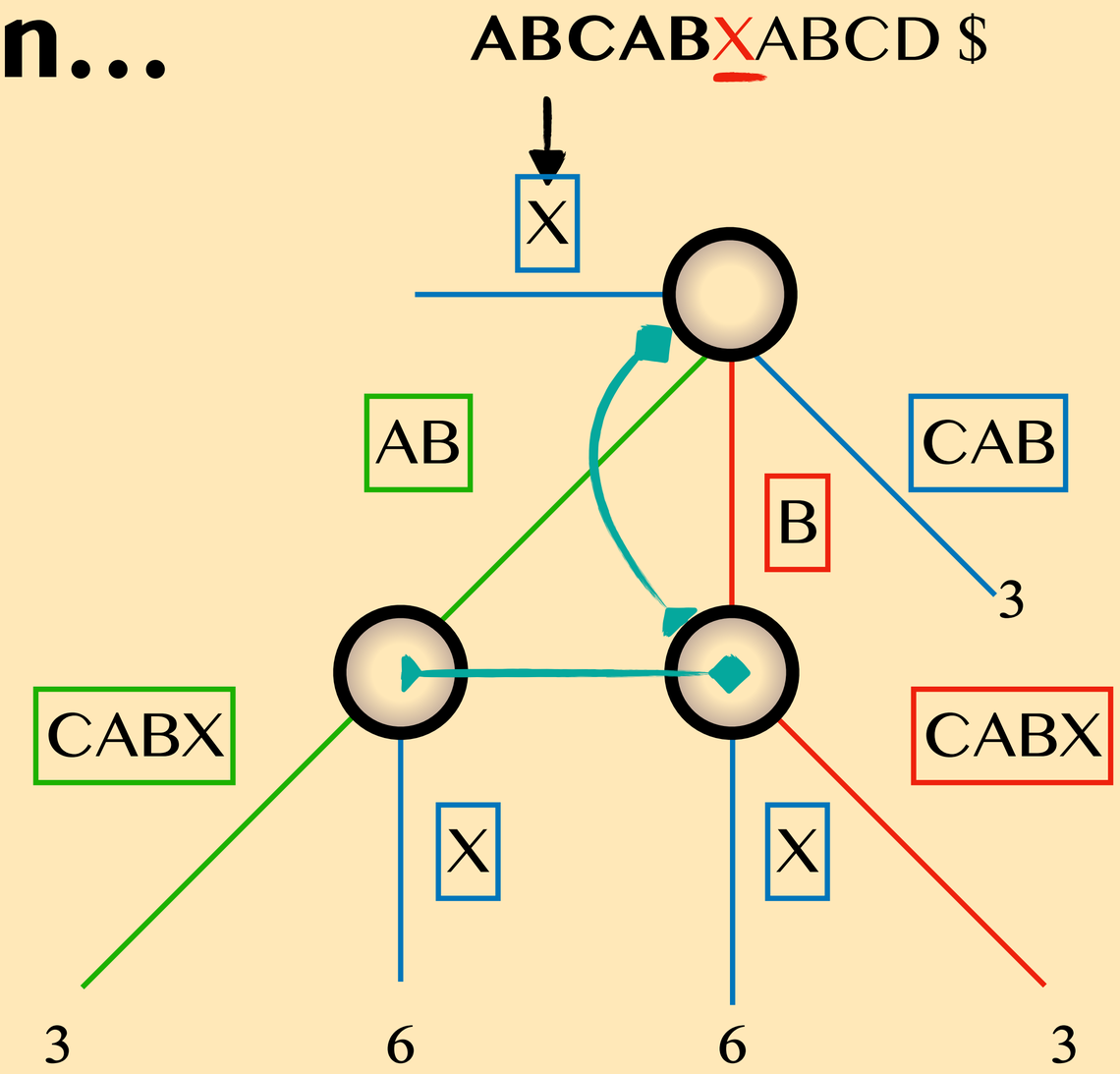
Phase: #6.2

remSuffixCount = 1

activePoint = (root, ‘x’, 0)

end = 6

Ukkonen...



Phase: #6

remSuffixCount = 0

activePoint = (root, ‘x’, 0)

end = 6