Prepared by: [Arun Konagurthu]

FIT3155: Advanced Algorithms and Data Structures

Week 3: **Linear time suffix tree construction**

Faculty of Information Technology, Monash University

What is covered in this lecture series?

Linear-time suffix tree construction

- Ukkonen algorithm (suffix tree construction)

Source material and recommended reading

- Dan Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press. (Chapter 5)
- Francisco Gomez Martin, Exact String Pattern Recognition
- Ukkonen, E. (1995). "On-line construction of suffix trees". Algorithmica 14 (3): 249260.

Introduction

The substring (matching) problem

Given a reference text **`txt[1...n]`**, preprocess **`txt`** such that any given pattern **`pat[1...m]`** can be searched in time proportional to the length of the pattern, $O(m)$.*

---

*Compare this with what we learnt from Z-algorithm and Boyer-Moore algorithm.

- Suffix trees (and similarly suffix arrays) permit solving the above (and many other related) problems. They are very versatile.
- Suffix trees unravel the composition of any string, and permit efficient access to them.

String definitions

Given $\texttt{str}[1...n]$

- A **prefix** of $\texttt{str}[1..n]$ is a **substring** $\texttt{str}[1..i]$, $\forall 1 \leq i \leq n$.
- A **suffix** of $\texttt{str}[1..n]$ is a **substring** $\texttt{str}[j..n], \forall 1 \leq j \leq n$.
- A **substring** of $\texttt{str}[1..n]$ is any $\texttt{str}[j..i], \forall 1 \leq (j \leq i) \leq n$.
- Therefore,
  - a substring is a **prefix of a suffix**
  - (or equivalently) a substring is a **suffix of a prefix**

Suffixes of a string – Example

Consider the suffixes of $\texttt{str}[1..n]\texttt{\$} = \texttt{abaaba\$}$

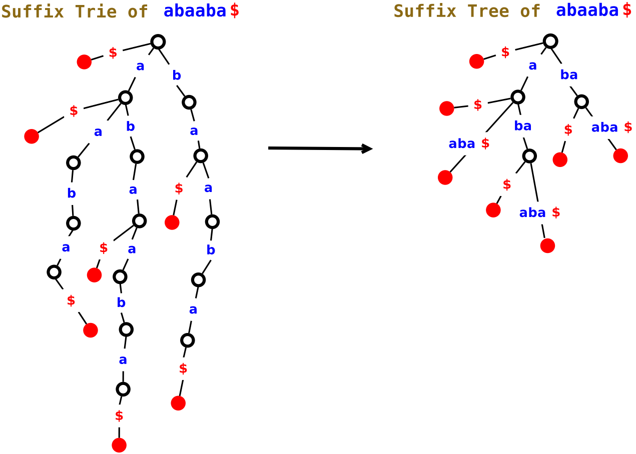
```
SUFFIX 1 = abaaba$
SUFFIX 2 = baaba$
SUFFIX 3 = aaba$
SUFFIX 4 = aba$
SUFFIX 5 = ba$
SUFFIX 6 = a$
SUFFIX 7 = $
```

The '$' symbol

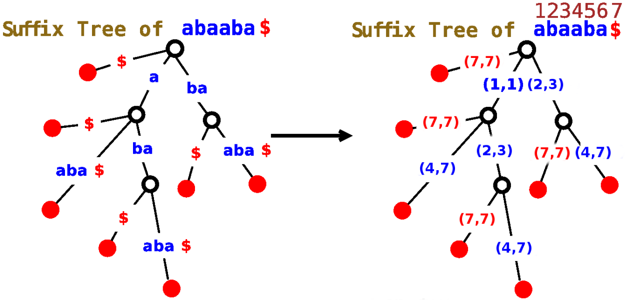
Note that `$` is a **special character** to denote the **`end of the string`**. It is often chosen to be a character that is **`lexicographically smaller`** than **all** other characters in the text.

Efficient suffix tree construction using **`Ukkonen's`** algorithm

Recall from FIT2004: path compressed suffix tries = suffix trees



Recall from FIT2004:
Efficient representation of suffix trees requires $O(n)$ space



Note, instead of storing the edge labels as substrings **explicitly**, we can store them **implicitly** using (j, i) denoting the substring **str**$[j..i]$, where $1 \leq (j \leq i) \leq n$.

Building a suffix tree naïvely

Consider suffixes of **str**$= \overset{1}{\text{a}}\ \overset{2}{\text{b}}\ \overset{3}{\text{c}}\ \overset{4}{\text{a}}\ \overset{5}{\text{b}}\ \overset{6}{\text{\$}}$

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```

Building a suffix tree naïvely

	1	2	3	4	5	6
Consider suffixes of **str**=	a	b	c	a	b	$

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```

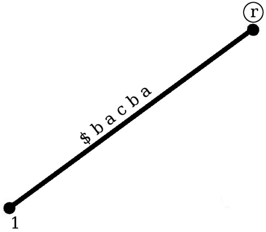
- Start with the (empty) root node of the suffix tree: ⓡ

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

Consider suffixes of **str**= a b c a b $

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```



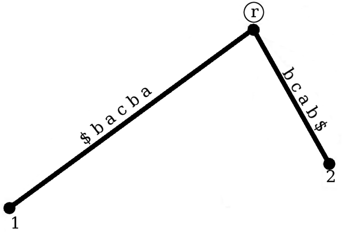
- Insert suffix 1 (**str**$[1 \ldots]$) into an empty tree.
- Call the resultant tree T_1.

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

Consider suffixes of **str**= $\overset{1}{a}\ \overset{2}{b}\ \overset{3}{c}\ \overset{4}{a}\ \overset{5}{b}\ \overset{6}{\$}$

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```



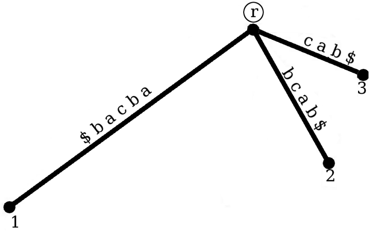
- Insert suffix 2 into T_1.
- Note **str**$[2....]$ is not a prefix of **str**$[1....]$.
- So create a new leaf node for **str**$[2..]$ suffix, branching off at ⓡ.
- Call resultant tree T_2

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

Consider suffixes of **str**= $\overset{1}{a}\ \overset{2}{b}\ \overset{3}{c}\ \overset{4}{a}\ \overset{5}{b}\ \overset{6}{\$}$

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```



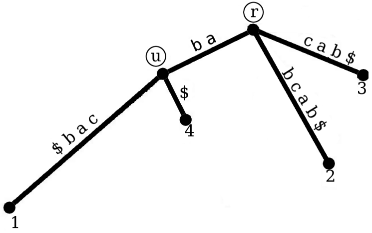
- Insert suffix 3 into T_2.
- Note **str**$[3....]$ is not a prefix of **str**$[1....]$ or **str**$[2....]$.
- So create a new leaf node for **str**$[3..]$ suffix, again branching off at ⓡ.
- Call resultant tree T_3

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

1 2 3 4 5 6
Consider suffixes of **str**= a b c a b $

SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $



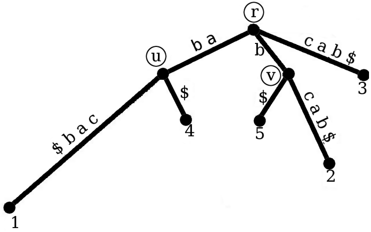
- Inserting suffix 4 into T_3.
- Note **str**$[4..5]$ is the longest common prefix shared with the suffix **str**$[1....]$.
- So, a new node ⓤ is **inserted** ...
- ... along the edge between ⓡ and the leaf node 1...
- ... with another edge branching off ⓤ to the new leaf node 4
- Call resultant tree T_4

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

Consider suffixes of **str**$=\overset{1}{\text{a}}\ \overset{2}{\text{b}}\ \overset{3}{\text{c}}\ \overset{4}{\text{a}}\ \overset{5}{\text{b}}\ \overset{6}{\text{\$}}$

SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $



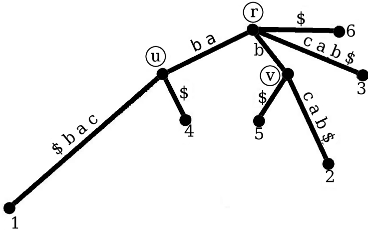
- Insert suffix 5 into T_4.
- Note **str**$[5..5]$ is the longest common prefix shared with suffix **str**$[2..]$.
- So, a new node ⓥ is **inserted**...
- ... along the edge between ⓡ and the leaf node 2...
- ... with another edge branching out from ⓥ to the new leaf node 5
- Call resultant tree T_5

What is the time complexity of this naïve construction?

Building a suffix tree naïvely

Consider suffixes of **str**= $\overset{1}{a}\ \overset{2}{b}\ \overset{3}{c}\ \overset{4}{a}\ \overset{5}{b}\ \overset{6}{\$}$

```
SUFFIX 1: str[1..6] = a b c a b $
SUFFIX 2: str[2..6] = b c a b $
SUFFIX 3: str[3..6] = c a b $
SUFFIX 4: str[4..6] = a b $
SUFFIX 5: str[5..6] = b $
SUFFIX 6: str[6..6] = $
```



- Insert suffix 6 into T_5.
- Note the suffix **str**$[6..6]$ denotes the special terminal character \$.
- This creates a new isolated edge branching off the root ⓡ.

What is the time complexity of this naïve construction?

Linear Time Suffix Tree construction

Esko Ukkonen in 1995 gave a very clever algorithm to construct suffix trees in linear run time.

Ukkonen, E. (1995). "On-line construction of suffix trees". Algorithmica 14 (3): 249260.

Esko Ukkonen

Finnish computer scientist

Esko Juhani Ukkonen is a Finnish theoretical computer scientist known for his contributions to string algorithms, and particularly for Ukkonen's algorithm for suffix tree construction. He is a professor at the University of Helsinki. Wikipedia

Born: 26 January 1950 (age 69 years), Savonlinna, Finland

Alma mater: University of Helsinki

Known for: Ukkonen's algorithm

Doctoral advisor: Martti Tienari

Ukkonen's linear-time algorithm – introduction

Ukkonen's algorithm exploits the properties inherent in any suffix tree, and benefits from several 'tricks' to achieve a linear run time.

Ukkonen's algorithms uses the following main ideas:

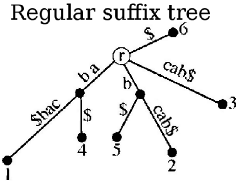
1. construct and use an '**implicit suffix tree**' data structure.
    - the actual suffix tree is computed after iteratively constructing (over many phases) an **implicit suffix tree**.
2. enhance this **implicit suffix tree** using '**suffix links**'.
    - this helps make the traversals on the implicit tree much faster.
3. gain from a set of implementational 'tricks':
    - these tricks avoid unnecessary computations, thus speeding up the algorithm drastically.

Only when all these ideas/elements are used together, Ukkonen's algorithm achieves a linear-time construction of suffix trees.

Implicit suffix tree compared with (regular) suffix tree

The relationship between an **`implicit suffix tree`** and its **`regular suffix tree`** can be understood by the following operations on the regular suffix tree:

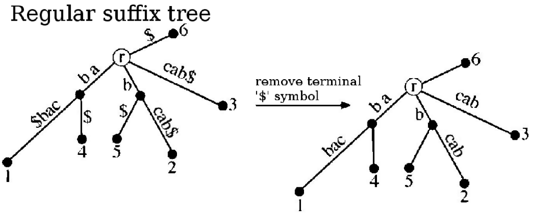
- Start with a regular suffix tree (of **`str`**=$\overset{1}{\texttt{a}}\ \overset{2}{\texttt{b}}\ \overset{3}{\texttt{c}}\ \overset{4}{\texttt{a}}\ \overset{5}{\texttt{b}}\ \overset{6}{\texttt{\$}}$).



Implicit suffix tree compared with (regular) suffix tree

The relationship between an **`implicit suffix tree`** and its **`regular suffix tree`** can be understood by the following operations on the regular suffix tree:

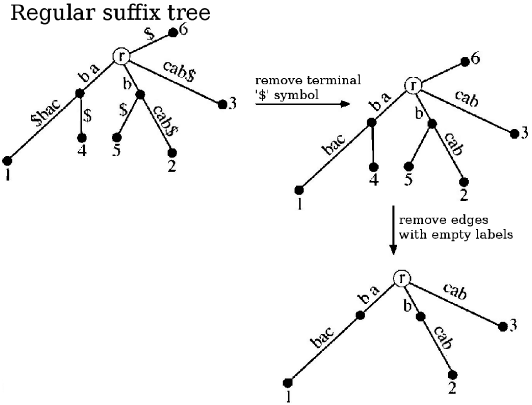
- Start with a regular suffix tree (of **`str`**=$\overset{1}{\texttt{a}}\ \overset{2}{\texttt{b}}\ \overset{3}{\texttt{c}}\ \overset{4}{\texttt{a}}\ \overset{5}{\texttt{b}}\ \overset{6}{\texttt{\$}}$).
- Remove all terminal (**`$`**) characters in the regular suffix tree.



Implicit suffix tree compared with (regular) suffix tree

The relationship between an **`implicit suffix tree`** and its **`regular suffix tree`** can be understood by the following operations on the regular suffix tree:

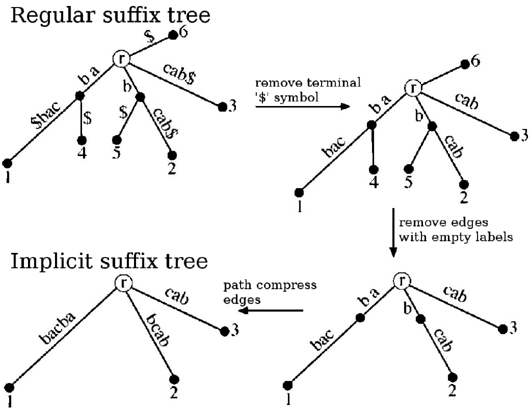
- Start with a regular suffix tree (of **`str`**=$\overset{1}{\texttt{a}}\ \overset{2}{\texttt{b}}\ \overset{3}{\texttt{c}}\ \overset{4}{\texttt{a}}\ \overset{5}{\texttt{b}}\ \overset{6}{\texttt{\$}}$).
- Remove all terminal (**`$`**) characters in the regular suffix tree.
- Then, remove all edges without edge labels (i.e. substrings).



Implicit suffix tree compared with (regular) suffix tree

The relationship between an **`implicit suffix tree`** and its **`regular suffix tree`** can be understood by the following operations on the regular suffix tree:

- Start with a regular suffix tree (of **`str`**=$\overset{1}{\texttt{a}}\ \overset{2}{\texttt{b}}\ \overset{3}{\texttt{c}}\ \overset{4}{\texttt{a}}\ \overset{5}{\texttt{b}}\ \overset{6}{\texttt{\$}}$).
- Remove all terminal (**`$`**) characters in the regular suffix tree.
- Then, remove all edges without edge labels (i.e. substrings).
- Then, path compress the tree by removing all nodes that do not have at least two children.



Ukkonen's algorithm builds implicit suffix trees incrementally in **phases**

Given a string **str**$[1..n]$, Ukkonen's algorithm proceeds over n "**phases**"

- Each **phase** $i+1$ (where $1 \leq i+1 \leq n$) builds the **implicit** suffix tree (denoted by $\mathbf{implicitST}_{i+1}$) for the **prefix** **str**$[1..i+1]$.

- Importantly, each $\mathbf{implicitST}_{i+1}$ is incrementally computed using the $\mathbf{implicitST}_{i}$ from the previous phase.
  - The construction of $\mathbf{implicitST}_{i+1}$ from $\mathbf{implicitST}_{i}$, in turn, involves several **suffix extension** steps, one for each suffix of the form **str**$[j..i+1]$, where $j = 1...i+1$ in that order .

Each phase involves suffix extensions

Suffix extension steps in each phase

- In any phase $i+1$, the suffixes in $\texttt{implicitST}_i$ (from previous phase i) undergo **suffix extensions** to accommodate the **new character**, **str**$[i+1]$.
- Thus, extending any suffix starting at position j, where $1 \leq j \leq i+1$, in the current phase $i+1$ involves:
    - finding the end of the path from root node ⓡ corresponding to the suffix **str**$[j \ldots i]$ in the current state of the implicit suffix tree, and
    - extending the end of **str**$[j \ldots i]$ path by appending **str**$[i+1]$ to that (growing) suffix.

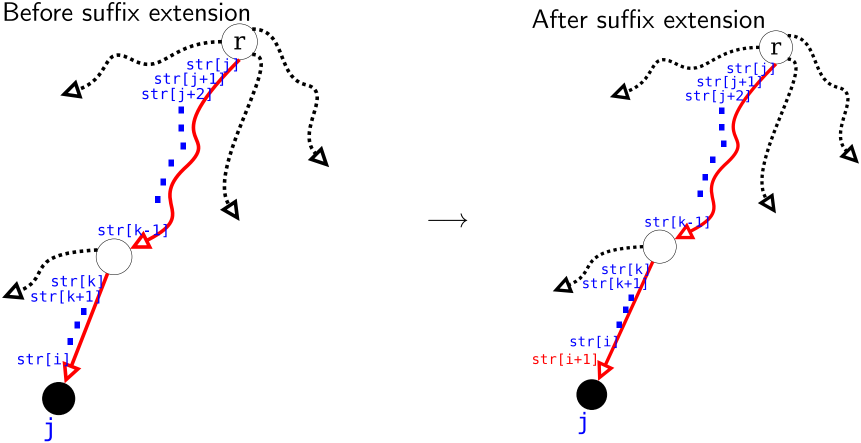
Algorithm at an very high level

```
Construct **implicitST**$_1$
For $i$ from 1 to $n-1$
    Begin **PHASE** $i+1$
    ▸ For $j$ from 1 to $i+1$
            Begin **SUFFIX EXTENSION** $j$
        ★ Find end of path from root denoting **str**$[j..i]$ in the current state of the suffix tree.
        ★ Apply one of the three suffix extension rules (see slides 18-20 discussed below).
      End of extension step $j$ (i.e., **str**$[j..i+1]$ extension computed).
End of phase $i+1$ (**implicitST**$_{i+1}$ computed)
```

Suffix extension rules – Rule 1 of 3

RULE 1 of 3

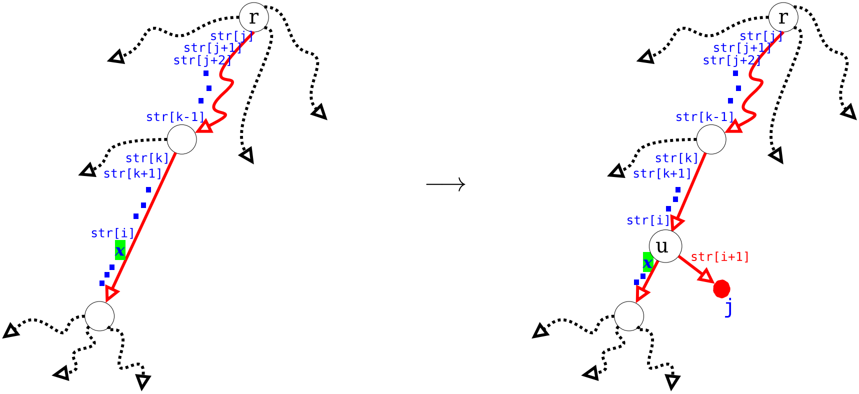
If the path **str**$[j..i]$ in **implicitST**$_i$ **ends at a leaf**, adjust the **label of the edge** to that leaf to account for the added character **str**$[i+1]$.



Suffix extension rules – Rule 2 of 3

RULE 2 of 3

If the path **str**$[j..i]$ in **implicitST**$_i$ **does NOT** end at a leaf, and the next character in the existing path is some $x \neq$ **str**$[i+1]$, then split the edge after **str**$[..i]$ and create a new node ⓤ, followed by a new leaf numbered j; assign character **str**$[i+1]$ as the edge label between the new node ⓤ and leaf j.

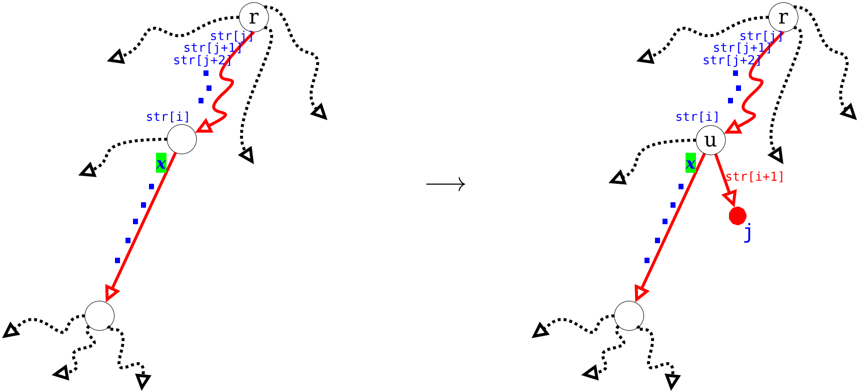

Before suffix extension

After suffix extension

Suffix extension rules – Rule 2 of 3 (an alternative scenario)

RULE 2 of 3 – an alternative scenario that can arise

If the path **str**$[j..i]$ in **implicitST**$_i$ **does NOT** end at a leaf, and the next character in the existing path is some $x \neq$ **str**$[i+1]$, and **str**$[i]$ and x are separated by an existing node ⓤ, then create a new leaf numbered j; assign character **str**$[i+1]$ as the edge label between the ⓤ and the leaf j.



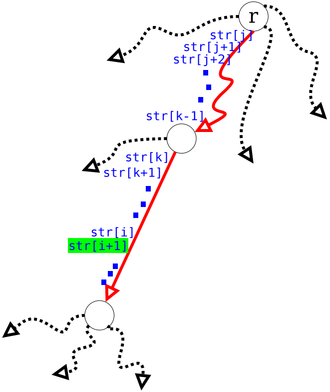
Before suffix extension

After suffix extension

Suffix extension rules – Rule 3 of 3

RULE 3 of 3

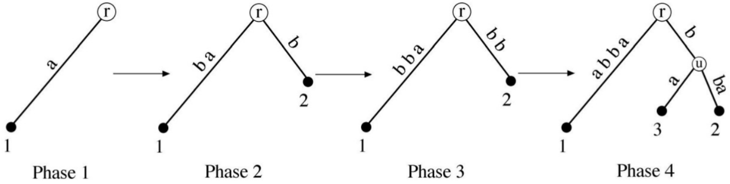
If the path **str**$[j..i]$ in **implicitST**$_i$ **does NOT** end at a leaf, but is within some edge label, and the next character in that path **is** **str**$[i+1]$, then **str**$[i+1]$ is already in the tree. **`No further action needed.`**



⟶ **`No further action needed.`**

Suffix extension rules – Example

For the string **str**=$\overset{1}{a}\ \overset{2}{b}\ \overset{3}{b}\ \overset{4}{a}$, the implicit suffix trees for each phase, along with their corresponding suffix extensions are shown below:



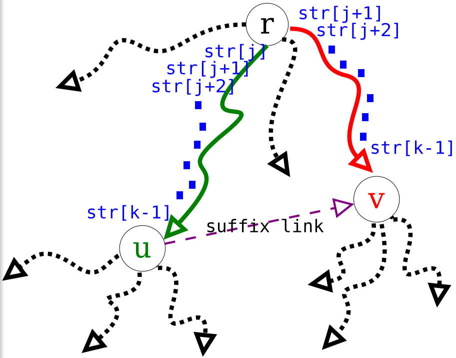
path	phase	extn	rule	comment
str[1..1]	1	1	rule-2	branch out of ⓡ into a new leaf 1 with label **str**[1]
str[1..2]	2	1	rule-1	extend edge label between nodes ⓡ and leaf 1 with **str**[2]
str[2..2]	2	2	rule-2	branch of ⓡ into a new leaf 2 with label **str**[2]
str[1..3]	3	1	rule-1	extend edge label between nodes ⓡ and leaf 1 with **str**[3]
str[2..3]	3	2	rule-1	extend edge label between nodes ⓡ and leaf 2 with **str**[3]
str[3..3]	3	3	rule-3	no further action necessary
str[1..4]	4	1	rule-1	extend edge label between nodes ⓡ and leaf 1 with **str**[4]
str[2..4]	4	2	rule-1	extend edge label between nodes ⓡ and leaf 2 with **str**[4]
str[3..4]	4	3	rule-2	insert node ⓤ along (ⓡ,2); attach leaf 3; edge-label **str**[4]
str[4..4]	4	4	rule-3	no further action necessary

Speeding up tree traversal using suffix links

Suffix links are simply **pointers** between internal nodes of an (implicit) suffix tree, that speed up traversal time in each phase.

Definition of a suffix link

- Let ⓤ and ⓥ be two internal nodes of an implicit suffix tree.
- Let the traversal from root node ⓡ to ⓤ yield some substring **str**$[j..k-1]$.[*]
- Let the traversal from root node ⓡ to ⓥ yield a substring **str**$[j+1..k-1]$.[†]
- Then the pointer from ⓤ to ⓥ defines a suffix link between those nodes.



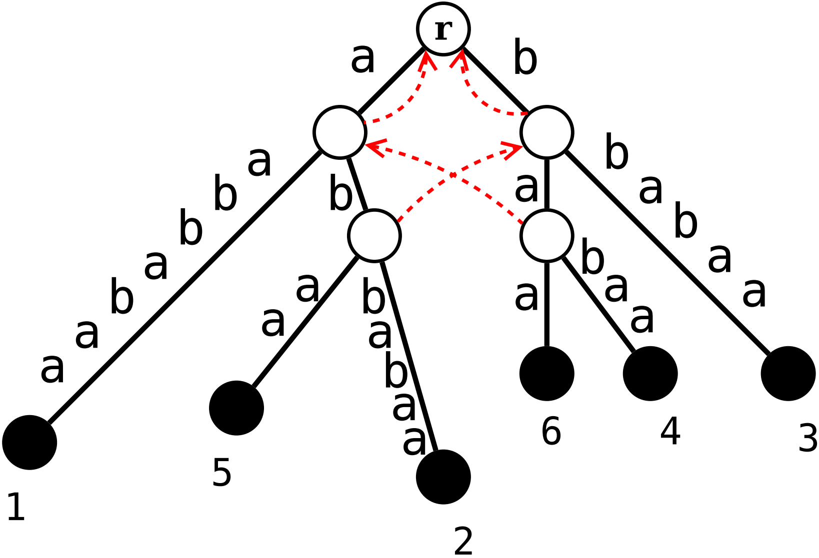
---

[*] Note1: $1 \leq j \leq k-1$

[†] Note2: When $j = k-1$, the path from ⓡ to ⓤ yields a **single character** substring. This implies, the path from ⓡ to ⓥ will yield an empty substring. In other words, ⓥ and ⓡ are one and the same in this case.

Example – Implicit suffix tree with suffix links

$\mathbf{str} = \overset{1}{a}\ \overset{2}{a}\ \overset{3}{b}\ \overset{4}{b}\ \overset{5}{a}\ \overset{6}{b}\ \overset{7}{a}\ \overset{8}{a}$



KEY OBSERVATION:

Every internal node of an implicit suffix tree has a suffix link from it

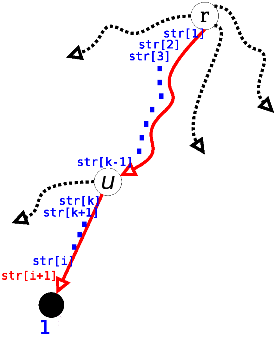
We will reason this from the point of view of Ukkonen's incremental algorithm

- If, in some suffix extension j of phase $i+1$, a new internal node ⓤ is added to the current state of the implicit suffix tree,
  - i.e., rule 2 of the suffix extension rules (slide 19) has been applied,
  - this implies, before ⓤ was newly created, the path $\textbf{str}[j..i]$ is continued by a character (say) x, where $x \neq \textbf{str}[i+1]$.
- Then, in the very next suffix extension $j+1$ of the same phase $i+1$:
  - **either** the path $\textbf{str}[j+1..i]$ continues **ONLY VIA** character x.
    - ★ which implies, a new internal node ⓥ must also be created, after $\textbf{str}[j+1..i]$, that branches to the new leaf $j+1$ via character $\textbf{str}[i+1]$.
  - **or** the path $\textbf{str}[j+1..i]$ already ends in an existing internal node ⓥ,
    - ★ ... with one branch below extending via character x
    - ★ ... and one (or more) branch(es), via other character(s).
  - Thus, a new suffix link ⓤ to ⓥ WILL be created in $j+1$ extension.

Following the trail of suffix links to build **implicitST**$_{i+1}$ from **implicitST**$_i$

Recall that in the extension j of phase $i+1$ the algorithm locates suffix **str**$[j..i]$, and extends it by **str**$[i+1]$, for each j increasing from 1 to $i+1$. Suffix links are used to speed these extensions. Let's see how.

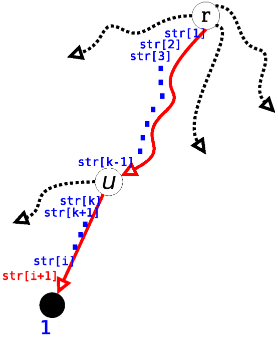
Extension 1, phase $i+1$



Extension $j = 1$, phase $i+1$

- The first suffix **str**$[1..i]$ in any **implicitST**$_i$ always ends in a leaf node.
- Therefore, extending this suffix by the character **str**$[i+1]$ in the next phase $i+1$ is always via Rule 1.
- To achieve this, start from the root node ⓡ, traverse to the node ⓤ that is the parent of this leaf node.
- From ⓤ, apply Rule 1 extension to its appropriate edge.

Extension 1, phase $i + 1$



Extension $j = 1$, phase $i + 1$

- The first suffix **str**$[1..i]$ in any **implicitST**$_i$ always ends in a leaf node.
- Therefore, extending this suffix by the character **str**$[i + 1]$ in the next phase $i + 1$ is always via Rule 1.
- To achieve this, start from the root node ⓡ, traverse to the node ⓤ that is the parent of this leaf node.
- From ⓤ, apply Rule 1 extension to its appropriate edge.

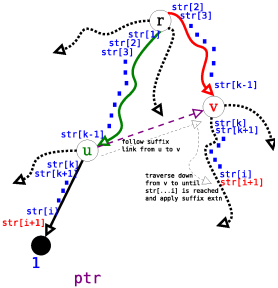
Terminology alert: "Active node" and "remainder" (substring)

In general, in any extension, the node ⓤ under whose direct edge the suffix extension rules are applied, is termed the "**active node**".

The substring **str**$[k \ldots i]$ that is remaining below the active node ⓤ and before the extended character **str**$[i + 1]$ is termed the "**remainder**" (substring)"

Extension 2, phase $i+1$

...continued



Extension $j+1=2$, phase $i+1$

- Note: Coming into this extension from the previous, ⓤ is the **active node**, and the **remainder** substring is **str**$[k..i]$. (Note: sometimes ⓤ $=$ ⓡ)
- In the current extension, to find the end of **str**$[2...i]$ (and extend it by **str**$[i+1]$):
  - From the **active node** ⓤ, follow its suffix link (shortcut) to ⓥ. (Note: If ⓤ $=$ ⓡ, then ⓥ $=$ ⓡ)
  - From ⓥ, traverse down along the **remainder** substring **str**$[k..i]$.
    (Note: If ⓥ $=$ ⓡ, then **remainder** is the full suffix, here **str**$[2..i]$, that must be extended naïvely by **str**[i+1])
  - Apply the pertinent suffix extension rule (1, 2 or 3), to extend by **str**$[i+1]$.

General extension procedure for phase $i + 1$

In fact, any extension $j + 1 \geq 2$ of phase $i + 1$ repeats the same procedure shown in the earlier slide:

1. Coming into any general extension from its previous, you know the **`active`** node ⓤ and the **`remainder`** (possibly empty) substring **`str`**$[k..i]$.
2. If ⓤ≠ⓡ, follow the suffix link (shortcut) from ⓤ to ⓥ. Traverse from ⓥ to the end of the **`remainder`** substring **`str`**$[k..i]$, which is same as the end of the path **`str`**$[j + 1..i]$ (our desired position to extend the suffix from phase i by **`str`**$[i + 1]$).
    - On the other hand, if ⓤ=ⓡ, then no choice but to naïvely traverse to the end of **`str`**$[j + 1..i]$ starting from the root ⓡ.
3. Once at the end of **`str`**$[j + 1..i]$, apply pertinent suffix extension rules (1, 2 or 3), to extend by **`str`**$[i + 1]$.

General extension procedure for phase $i + 1$

In fact, any extension $j + 1 \geq 2$ of phase $i + 1$ repeats the same procedure shown in the earlier slide:

1. Coming into any general extension from its previous, you know the **active** node ⓤ and the **remainder** (possibly empty) substring **str**$[k..i]$.
2. If ⓤ≠ⓡ, follow the suffix link (shortcut) from ⓤ to ⓥ. Traverse from ⓥ to the end of the **remainder** substring **str**$[k..i]$, which is same as the end of the path **str**$[j + 1..i]$ (our desired position to extend the suffix from phase i by **str**$[i + 1]$).
    - On the other hand, if ⓤ=ⓡ, then no choice but to naïvely traverse to the end of **str**$[j + 1..i]$ starting from the root ⓡ.
3. Once at the end of **str**$[j + 1..i]$, apply pertinent suffix extension rules (1, 2 or 3), to extend by **str**$[i + 1]$.

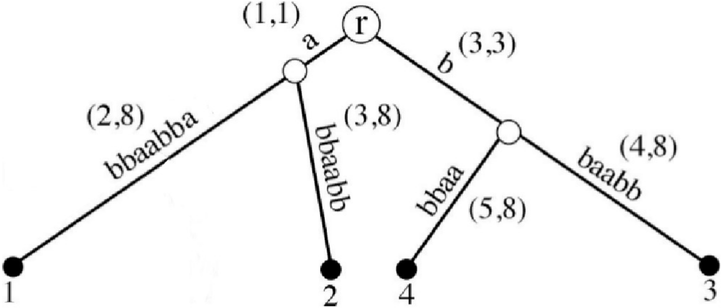
What about suffix links for newly created internal nodes during suffix extensions?

Note, when Rule 2 is applied under some active node ⓤ of any extension, a new iternal node is created below it, and this new node does not (yet) have a known suffix link. However, its suffix link gets resolved in the very next extension as observed on slide 24.

Implementation trick 1 – space-efficient representation of edge-labels/substrings

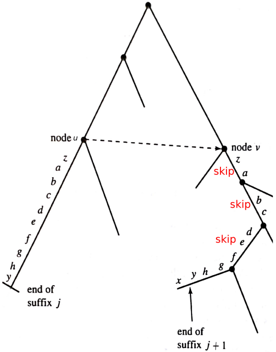
Recall, given any string **str**$[1..n]$, any substring can be represented by just two numbers: (**start-index**,**end-index**). Thus, the entire Ukkonen algorithm is processed using this space-efficient ($O(n)$-space) representation.

Below is an implicit suffix tree of the string **str**= $\overset{1}{\text{a}}\ \overset{2}{\text{a}}\ \overset{3}{\text{b}}\ \overset{4}{\text{b}}\ \overset{5}{\text{a}}\ \overset{6}{\text{a}}\ \overset{7}{\text{b}}\ \overset{8}{\text{b}}$, using (**start-index**,**end-index**) edge label representation.



Implementation trick 2 – skip/count trick

Using the space-efficient edge representation seen in the previous slide, all traversals, instead of being character-by-character comparisons, can be rapidly done by skipping over successive nodes along any desired path in a implicit suffix tree, while keep track of the total substring length skipped along that path. For example:



- Let the **remainder** substr be **str**$[k..i]$= $\begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \mathtt{z} & \mathtt{a} & \mathtt{b} & \mathtt{c} & \mathtt{d} & \mathtt{e} & \mathtt{f} & \mathtt{g} & \mathtt{h} & \mathtt{y} \end{array}$
- From node ⓥ, ask how many characters representing the edge starting with `z...` Here. it is 2.
- Since $10 > 2$, end not reached, so skip to the node receiving that edge.
- In **str**$[k..i]$, the 3rd $(3 = 2 + 1)$ character is `b`.
- Again, ask how many characters representing the edge starting with `b....` Again it is 2.
- Since $10 > 2 + 2$, skip to the node receiving that edge.
- From **str**$[k..i]$, the 5th $(5 = 2 + 2 + 1)$ character is `d`.
- Again, ask how many characters representing the edge starting with `d....` It is 3.
- Since $10 > 2 + 2 + 3$, skip to the node receiving that edge.
- ...and so on until the node beyond which further skips are not possible/necessary.

Implementation trick 3 – premature extension stopping criterion: **'Showstopper' rule!**

In any phase $i+1$, if `rule 3` extension (refer slide 20) applies in some suffix extension j, then extensions $j+1, j+2, \ldots i+1$ will all use `rule 3`. Because:

- when `rule 3` is used in extension j, implies the path corresponding to $\mathtt{str}[j..i]$ continues with the character $\mathtt{str}[i+1]$.
- This implies, the path corresponding to the substring $\mathtt{str}[j+1..i]$ also continues with the character $\mathtt{str}[i+1]$
- Similarly, this remains true for all subsequent extensions.
- Punchline: Since `rule 3` requires **no further action**, extensions in this phase can **STOP** prematurely on encountering `rule` 3, and the algorithm can directly start the extensions for the next phase.

Implementation trick 4 – rapid leaf extension trick (slide 1/4)

Observation– In Ukkonen's algorithm, `once a leaf, always a leaf`

If at some phase i in Ukkonen's algorithm, a leaf is created and labeled j (denoting a suffix $\texttt{str}[j..i]$ of the prefix $\texttt{str}[1..i]$), then that leaf will remain a leaf in all subsequent phases ($> i$).

Why?

Implementation trick 4 – rapid leaf extension trick (slide 1/4)

Observation– In Ukkonen's algorithm, **`once a leaf, always a leaf`**

If at some phase i in Ukkonen's algorithm, a leaf is created and labeled j (denoting a suffix **`str`**$[j..i]$ of the prefix **`str`**$[1..i]$), then that leaf will remain a leaf in all subsequent phases ($> i$).

Why?

- Leaf node, when created (via **`rule 2`**) always stores as its label, the starting index j denoting where the corresponding suffix starts.
- In subsequent phases, whenever this suffix is extended at the leaf (via **`rule 1`**), only the edge-label connecting the leaf gets updated, and not the leaf node label.

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).
- In each phase i, suffixes get extended (using rule 1) and (potentially) new suffixes get added (using rule 2)...

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).
- In each phase i, suffixes get extended (using rule 1) and (potentially) new suffixes get added (using rule 2)...
- ...before the phase ends (either prematurely using rule 3, or naturally when j reaches i).

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).
- In each phase i, suffixes get extended (using rule 1) and (potentially) new suffixes get added (using rule 2)...
- ...before the phase ends (either prematurely using rule 3, or naturally when j reaches i).
- Let $\texttt{last}_{j_i}$ denote the **last** extension j (via rule 1 or 2) for phase i.

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).
- In each phase i, suffixes get extended (using rule 1) and (potentially) new suffixes get added (using rule 2)...
- ...before the phase ends (either prematurely using rule 3, or naturally when j reaches i).
- Let $\mathbf{last}_{j_i}$ denote the **last** extension j (via rule 1 or 2) for phase i.
- Since the number of leaves between two consecutive phases is non-decreasing...

Implementational trick 4 – rapid leaf extension trick (slide 2/4)

- Phase 1 consists of a single edge tree, root node ⓡ to leaf node numbered 1 (created using rule 2).
- In each phase i, suffixes get extended (using rule 1) and (potentially) new suffixes get added (using rule 2)...
- ...before the phase ends (either prematurely using rule 3, or naturally when j reaches i).
- Let $\texttt{last}_{j_i}$ denote the **last** extension j (via rule 1 or 2) for phase i.
- Since the number of leaves between two consecutive phases is non-decreasing...
- ... and a new leaf is created only upon application of rule 2, it follows that $\texttt{last}_{j_i} \leq \texttt{last}_{j_{i+1}}$.

Implementational trick 4 – rapid leaf extension trick (slide 3/4)

- Note: if for any suffix extension j in phase i we applied rule 1 or rule 2, it automatically implies that the suffix extension j in phase $i+1$ will require (only) rule 1.
- Therefore, after each phase i, the observation (from the previous slide) that $\texttt{last}_{j_i} \leq \texttt{last}_{j_{i+1}}$ can be exploited and we can omit/avoid **explicit** suffix extensions 1 to $\texttt{last}_{j_i}$ for the next phase $i+1$, and do so rapidly using the (implicit) extension trick shown on the next slide...

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).
- For each edge connecting the **`leaf node`** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring **`str`**$[k..i+1]$.

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).
- For each edge connecting the **leaf node** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring **str**$[k..i+1]$.
- In each suffix extension, instead of EXPLICITLY updating the **`end-index`** (of the edge to the leaves) to i+1, index it IMPLICITLY to a $global\_end$ variable, i.e., $(k, global\_end)$

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).
- For each edge connecting the **leaf node** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring **str**$[k..i+1]$.
- In each suffix extension, instead of EXPLICITLY updating the **end-index** (of the edge to the leaves) to i+1, index it IMPLICITLY to a $global\_end$ variable, i.e., $(k, global\_end)$
  - Note: when phase $i+1$ starts, $global\_end$ is implicitly $i+1$.

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).
- For each edge connecting the **`leaf node`** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring **`str`**$[k..i+1]$.
- In each suffix extension, instead of EXPLICITLY updating the **`end-index`** (of the edge to the leaves) to i+1, index it IMPLICITLY to a $global\_end$ variable, i.e., $(k, global\_end)$
  - Note: when phase $i+1$ starts, $global\_end$ is implicitly $i+1$.
- In phase i+1, since rule 1 applies to all extensions of leaf nodes from 1 to **`last`**$_{j_i}$ (see previous slide)...

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers: (`start-index`, `end-index`).
- For each edge connecting the **leaf node** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring $\texttt{str}[k..i+1]$.
- In each suffix extension, instead of EXPLICITLY updating the `end-index` (of the edge to the leaves) to i+1, index it IMPLICITLY to a $global\_end$ variable, i.e., $(k, global\_end)$
  - Note: when phase $i+1$ starts, $global\_end$ is implicitly $i+1$.
- In phase i+1, since rule 1 applies to all extensions of leaf nodes from 1 to $\texttt{last}_{j_i}$ (see previous slide)...
- ...**no additional explicit work** is required to implement extensions $j = 1$, $j = 2$, $\ldots j = \texttt{last}_{j_i}$. These can be strightaway ignored.

Implementational trick 4 – rapid leaf extension trick (slide 4/4)

- From the space-efficient representation (see slide 29) we know that any edge is represented using two numbers:
  (`start-index`, `end-index`).
- For each edge connecting the **leaf node** (whose index is j, in phase $i+1$, the edge label would be of the form $(k, i+1)$, denoting the substring **str**$[k..i+1]$.
- In each suffix extension, instead of EXPLICITLY updating the `end-index` (of the edge to the leaves) to i+1, index it IMPLICITLY to a $global\_end$ variable, i.e., $(k, global\_end)$
  - Note: when phase $i+1$ starts, $global\_end$ is implicitly $i+1$.
- In phase i+1, since rule 1 applies to all extensions of leaf nodes from 1 to **last**$_{j_i}$ (see previous slide)...
- ...**no additional explicit work** is required to implement extensions $j = 1$, $j = 2$, $\ldots j =$ **last**$_{j_i}$. These can be strightaway ignored.
- EXPLICIT extensions only start from $j =$ **last**$_{j_i} + 1$ until the first extension using rule 3 or until the phase ends.

Putting trick 4 pieces together – procedure to handle extensions in any single phase

Summarizing trick 4 , for any phase $i + 1$, the extension procedure is as follows:

1. Increment $global\_end$ index to i+1.
    - With just this operation, suffix extensions 1 to $\texttt{last}_{j_i}$ are implicitly complete without any additional work.
2. Explicitly compute successive extensions (refer slide 28) starting from $j = \texttt{last}_{j_i} + 1$, until some position $q \leq i + 1$ where the phase either prematurely terminates (after first encountering rule 3 extension), or all extensions are completed for this phase – treat q as $i + 2$ in this latter case.
3. To prepare for next phase $i + 2$, set $\texttt{last}_{j_{i+1}}$ to $q - 1$ and repeat the procedure above, until all phases are complete.

Key Observation

Two consecutive phases share **at most** one index q where an EXPLICIT extension is carried out.

Creating the final suffix tree from **implicitST**$_n$ for **str**$[1...n]$

The final suffix tree from its implicit version can be computed in $O(n)$-time as follows:

- First add a string terminal symbol $\$$ to the end of **str**, i.e. **str**$[1...n]\$$.
- Continue one more phase on **implicitST**$_n$ to account for this new character.
- The effect is that no suffix is now a prefix in **implicitST**$_n$.
- So each suffix of **str**$[1..n]$ gets appended by $\$$, yielding the **regular/explicit** suffix tree.

Ukkonen's algorithm runs in $O(n)$ time for a string **str**[1..n]

- There are only n phases in the algorithm.
- Each phase shares at most 1 explicit suffix extension (see previous slide)
- Hence, total number of explicit suffix extensions is at most $2n$.
- To quantify the effort in each extension:
  - (refer slide 28)
  - If the node in the tree after which the suffix extension **str**$[j-1..i]$ ends is at depth d from ⓡ
  - Then ⓤ is at depth at least $d-1$.
  - This implies the node receiving its suffix link, ⓥ is at least $d-2$
- Total number of skips over **all phases** is $O(n)$
- From this, it follows, Ukkonen takes $O(n)$-time

Alternate reasoning

An equivalent reasoning can be explored using the indexes of characters (instead of depth) leading into active nodes between successive iterations. Specifically, these indexes (unlike depths) between iterations have to be monotonically increasing. This bounds the total number of hops during skip-counts over all explicit suffix extensions by n.

Next...

Efficient Disjoint-set/Union-find data structures

-=o0o=-

END

-=o0o=-