# ADS: Algorithmen und Datenstrukturen 2

## Teil VII: Suffix Trees

Steve Hoffmann

Bioinformatics Group, Dept. of Computer Science & Interdisciplinary Center for Bioinformatics, **University Leipzig**

22. Juni 2011

# suffix and prefix: intuition

What is a suffix?

- *"suffix?"* is a suffix of *"What is a suffix?"*
- *" a suffix?"* is a suffix of *"What is a suffix?"*
- *"What is a suffix?"* for sure is a suffix of *"What is a suffix?"*

What is a prefix?

- *"What"* is a prefix of *"What is a suffix? "*
- *"What is"* is a prefix of *"What is a suffix? "*
- *"What is a suffix?"* for sure is a prefix of *"What is a suffix?"*

# formalities: alphabet and characters

Let $\mathcal{A}$ be a finite set, the alphabet.

- the elements of $\mathcal{A}$ are characters.
- $\epsilon$ denotes the empty string.
- strings are written by juxtaposition of characters:

The set $\mathcal{A}^*$ of strings over $\mathcal{A}$ is defined by

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}_i \tag{1}$$

where $\mathcal{A}_0 = \{\epsilon\}$ and $\mathcal{A}_{i+1} = \{aw \mid a \in \mathcal{A}, w \in A^i\}$.

- $\mathcal{A}^+$ denotes a non-empty string

## formalities: strings, suffixes and prefixes

Let $s$ be a string over the alphabet $\mathcal{A}$

- let $|s|$ denote the length of $s$.

We assume a string of the form $s = uvw$, $u, v, w \in \mathcal{A}^*$:

- $u$ is a prefix of $s$
- $v$ is a substring of $s$
- $w$ is a suffix of $s$

Note, that by using $\epsilon$ every string may be partioned to $uvw$!

# Searching strings

Current genome projects and internet data bases accumulate massive amounts of sequences (texts). Searching such texts can be pretty cumbersome!

Considering a large sequence $S$ over $\mathcal{A} = \{\mathtt{A}, \mathtt{C}, \mathtt{T}, \mathtt{G}\}$. We may ask:

- does ACTGCTTACGTACGGTA occur in $S$?
- how often does ACTGCTTACGTACGGTA occur in $S$?
- where does ACTGCTTACGTACGGTA occur in $S$?

For large sequences that are frequently queried we need more sophisticated strategies to answer these questions quickly.

# Indexing

For a large string $s$ that is frequently queried (ie. a genome) we could think about **indexing all substrings**. The following theorem shows why this is **not such a good idea**:

**Theorem**

*A string $s$ of length $|s| = n$ has at most $O(n^2)$ different substrings.*

**Proof.**

Idea: for each $0 \leq i \leq n-1$,

$$\bigcup_{i \leq j \leq n-1} s[i..j] \tag{2}$$

is the set of substrings beginning at position $i$. Alphabet? □

Hence, we need another way to represent this data.

# towards suffix trees: implicit representation of substrings

Consider the string $s = abab$

| | | | | |
|---|---|---|---|---|
| string | a | b | a | b |
| position | 0 | 1 | 2 | 3 |

# towards suffix trees: implicit representation of substrings

Consider the string $s = abab$

| string | a | b | a | b |
|---|---|---|---|---|
| position | 0 | 1 | 2 | 3 |

we observe substrings:

$$a \quad ab \quad aba \quad abab \quad b \quad ba \quad bab$$

- **substrings are prefixes of suffixes**: $abab$, $bab$, $ab$, $b$
- each substring is implicitly represented by at least one suffix
- **at most** $O(n^2)$ **substrings but only** $O(n)$ **suffixes!**

# towards suffix trees: implicit representation of substrings

Consider the string $s = abab$

| string | a | b | a | b |
|---|---|---|---|---|
| position | 0 | 1 | 2 | 3 |

we observe substrings:

$a$ $ab$ $aba$ $abab$ $b$ $ba$ $bab$

- **substrings are prefixes of suffixes**: $abab$, $bab$, $ab$, $b$
- each substring is implicitly represented by at least one suffix
- **at most** $O(n^2)$ **substrings but only** $O(n)$ **suffixes!**

# towards suffix trees: implicit representation of substrings

Consider the string $s = abab$

| string | a | b | a | b |
|---|---|---|---|---|
| position | 0 | 1 | 2 | 3 |

we observe substrings:

$a \quad ab \quad aba \quad abab \quad b \quad ba \quad bab$

- **substrings are prefixes of suffixes**: *abab*, *bab*, *ab*, *b*
- each substring is implicitly represented by at least one suffix
- **at most $O(n^2)$ substrings but only $O(n)$ suffixes!**

# towards suffix trees: implicit representation of substrings

Consider the string $S = abab$

we observe substrings:

$a \quad ab \quad aba \quad abab \quad b \quad ba \quad bab$

substrings are prefixes of suffixes:

$abab, \; bab, \; ab, \; b$

- each substring is implicitly represented by at least one suffix
- **at most $O(n^2)$ substrings but only $O(n)$ suffixes!**

# towards suffix trees: implicit representation of substrings

Consider the string $S = abab$

we observe substrings:

$$a \quad ab \quad aba \quad abab \quad b \quad ba \quad bab$$

substrings are prefixes of suffixes:

$$abab,\ bab,\ ab,\ b$$

- each substring is implicitly represented by at least one suffix
- **at most $O(n^2)$ substrings but only $O(n)$ suffixes!**

# towards suffix trees: implicit representation of substrings

Consider the string $S = abab$

- we observe substrings $a$, $ab$, $aba$, $abab$, $b$, $ba$ and $bab$
- **substrings are prefixes of suffixes**: $abab$, $bab$, $ab$, $b$
- each substring is implicitly represented by at least one suffix
- **at most $O(n^2)$ substrings but only $O(n)$ suffixes!**

We append a unique character (sentinel) \$ to the end of $s$:

$$S\$ = abab\$$$

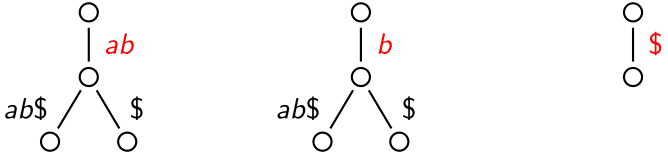
Reason?

# towards suffix trees: common prefixes

Given all suffixes *abab*\$, *bab*\$, *ab*\$, *b*\$, \$ we observe

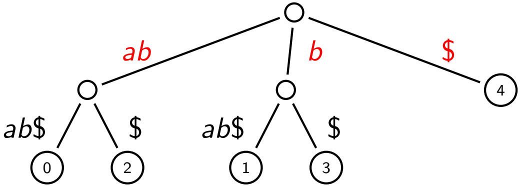
- *ab* is longest common prefix for *abab*\$ and *ab*\$
- *b* is longest common prefix for *bab*\$ and *b*\$
- \$ is longest prefix of \$

This gives us three partial trees:



# a suffix tree

Combining the three partial trees leads to a complete suffix tree:




| | | | | | |
|---|---|---|---|---|---|
| string | a | b | a | b | $ |
| position | 0 | 1 | 2 | 3 | 4 |

More on suffix tree construction later ...

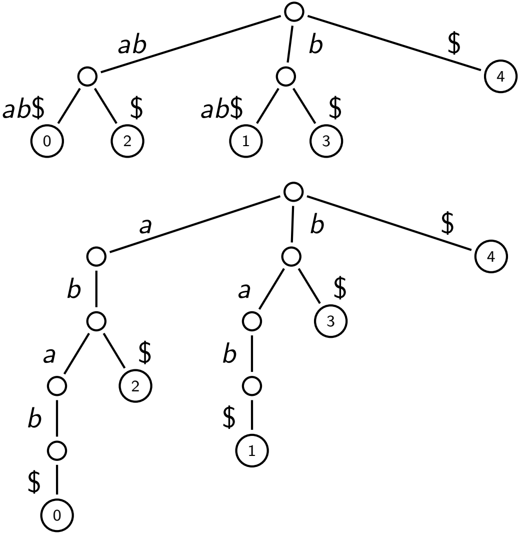
# formalities: suffix tree

## Definition (suffix tree)

A suffix tree $ST(S\$)$ for a sequence $S \in \mathcal{A}^+$

1. edges labeled by non-empty strings over $\mathcal{A}$
2. for every node only one edge begins with same $a \in \mathcal{A}$
3. compact, ie. has no internal nodes with less than two successors (in contrast to suffix trie)
4. represents all substrings of $S$

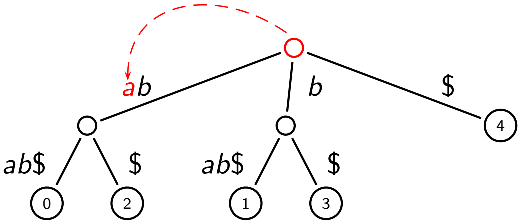
# a suffix tree is compact, a suffix trie is not



# searching in a suffix tree

Searching a query sequence $q$ in a suffix tree is easy. At internal nodes we look for a branch that starts with next character in $q$, on edges we simply continue to match the labels.
Lets search the query $q = ab$.

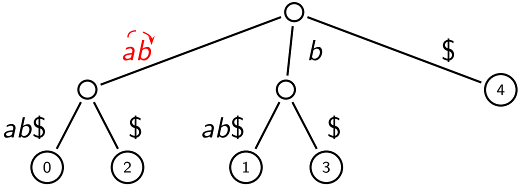
- the search starts with the first character of $q$, $q[0] = a$



# searching in a suffix tree

Searching a query sequence $q$ in a suffix tree is easy. At internal nodes we look for a branch that starts with next character in $q$, on edges we simply continue to match the labels.
Lets search the query $q = ab$.

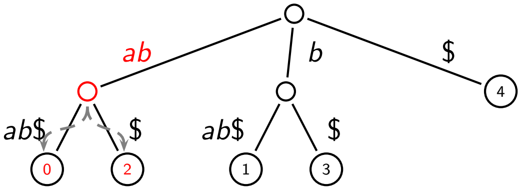
- the search starts with the first character of $q$, $q[0] = a$
- and continues with the next character $q[1] = b$



# searching in a suffix tree

Searching a query sequence $q$ in a suffix tree is easy. At internal nodes we look for a branch that starts with next character in $q$, on edges we simply continue to match the labels.
Lets search the query $q = ab$.

- the search starts with the first character of $q$, $q[0] = a$
- and continues with the next character $q[1] = b$
- we immediately see: $q$ occurs twice in $S\$$ at positions 0 and 2.



# complexity considerations

The suffix tree $ST(S)$ for the sequence $S\$$ with $|S| = n$

- hast at most $n - 1$ internal nodes and $n$ leaves $\Rightarrow O_{space}(n)$

Due to their compactness suffix trees require much less space in practice compared to suffix tries.

- for a query of length $m$, searches in the suffix tree take at most $m$ character comparisons $\Rightarrow O_{search}(m)$

# naive construction: top-down

In order to devise an algorithm to construct suffix trees, we will make use of the following observations:

- a node $\alpha$ represents all suffixes of $S\$$ that start with the same prefix $u$
- only one outgoing edge from $\alpha$ begins with same $a \in \mathcal{A}$
- we can restrict our evaluation to at most $|\mathcal{A}|$ subsets of suffixes with $uav$, where $a \in \mathcal{A}$ and $v \in \mathcal{A}^*$

## naive construction: simplifications

We will now devise the algorithm **topdown**. For reasons of simplicity we assume

- To construct the suffix tree we call **topdown** $(U, 0)$, where $U$ is a set of pairs that initially(!) holds all suffixes and their positions in S$, ie.
  $U = \{(s, i) \mid S\$[i..n-1] = s, 0 \leq i \leq n-1\}$
- let **getlcp(U)** be a function evaluates the longest common prefix for all strings $s$ in $(s, i) \in U$
- let **insertEdge**($\alpha$, $u$, $\beta$) be a function that inserts an edge with label $u$ between the nodes $\alpha$ and $\beta$.
- let **insertLeaf**($\alpha$, $u$, $i$) be a function that attaches a leaf $i$ with edge label $u$ to $\alpha$.

# naive construction: pseudo-code topdown

**Require:** set of pairs $U$, length of lcp $\ell$

$\forall a \in \mathcal{A} : P_a = \emptyset$, node $\alpha$

**for all** $(s, i) \in U$ **do**

  $a := s[\ell]$

  $P_a := P_a \cup (s[\ell + 1..n - 1], i)$

**end for**

**for all** $a \in \mathcal{A}$ **do**

  **if** $|P_a| > 1$ **then**

    $v :=$ getlcp($P_a$)

    $\beta :=$ topdown($P_a$, $|v|$)

    insertEdge($\alpha$, $av$, $\beta$)

  **else if** $|P_a| = 1$ **then**

    $(s, i) := P_a$

    attachLeaf($\alpha$, $as$, $i$)

  **end if**

**end for**

**return** $\alpha$

# naive construction: pseudo-code topdown

**Require:** set of pairs $U$, length of lcp $\ell$
  $\forall a \in \mathcal{A} : P_a = \emptyset$, node $\alpha$
  **for all** $(s, i) \in U$ **do**
    $a := s[\ell]$
    $P_a := P_a \cup (s[\ell + 1..n - 1], i)$
  **end for**
  **for all** $a \in \mathcal{A}$ **do**
    **if** $|P_a| > 1$ **then**
      $v :=$ getlcp($P_a$)
      $\beta :=$ topdown($P_a$, $|v|$)
      insertEdge($\alpha$, $av$, $\beta$)
    **else if** $|P_a| = 1$ **then**
      $(s, i) := P_a$
      attachLeaf($\alpha$, $as$, $i$)
    **end if**
  **end for**
  **return** $\alpha$

partition wrt. to first character not belonging to the lcp

# naive construction: pseudo-code topdown

**Require:** set of pairs $U$, length of lcp $\ell$
  $\forall a \in \mathcal{A} : P_a = \emptyset$, node $\alpha$
  **for all** $(s, i) \in U$ **do**
    $a := s[\ell]$
    $P_a := P_a \cup (s[\ell + 1..n - 1], i)$
  **end for**
  **for all** $a \in \mathcal{A}$ **do**
    **if** $|P_a| > 1$ **then**
      $v :=$ getlcp$(P_a)$
      $\beta :=$ topdown$(P_a, |v|)$
      insertEdge$(\alpha, av, \beta)$
    **else if** $|P_a| = 1$ **then**
      $(s, i) := P_a$
      attachLeaf$(\alpha, as, i)$
    **end if**
  **end for**
  **return** $\alpha$

partition wrt. to first character not belonging to the lcp

if there are >= 2 suffixes
get lcp for the set
call topdown
insert edge

# naive construction: pseudo-code topdown

**Require:** set of pairs $U$, length of lcp $\ell$
    $\forall a \in \mathcal{A} : P_a = \emptyset$, node $\alpha$
    **for all** $(s, i) \in U$ **do**
        $a := s[\ell]$
        $P_a := P_a \cup (s[\ell + 1..n - 1], i)$
    **end for**
    **for all** $a \in \mathcal{A}$ **do**
        **if** $|P_a| > 1$ **then**
            $v :=$ getlcp$(P_a)$
            $\beta :=$ topdown$(P_a, |v|)$
            insertEdge$(\alpha, av, \beta)$
        **else if** $|P_a| = 1$ **then**
            $(s, i) := P_a$
            attachLeaf$(\alpha, as, i)$
        **end if**
    **end for**
    **return** $\alpha$

partition wrt. to first character not belonging to the lcp

if there are >= 2 suffixes get lcp for the set call topdown insert edge

this suffix is unique we attach a leaf

# naive construction: example

For $s = abab\$$: $U = \{(abab\$, 0), (bab\$, 1), (ab\$, 2), (b\$, 3), (\$, 4)\}$

$\rightarrow$ `first call`

- $P_a = \{(bab\$, 0), (b\$, 2)\}$, getlcp$(P_a)$ = b, $|$ getlcp$(P_a)$ $| = 1$

# naive construction: example

For $s = abab\$$: $U = \{(abab\$, 0), (bab\$, 1), (ab\$, 2), (b\$, 3), (\$, 4)\}$

$\rightarrow$ `first call`

- $P_a = \{(bab\$, 0), (b\$, 2)\}$, $\text{getlcp}(P_a) = \text{b}$, $|\ \text{getlcp}(P_a)\ | = 1$
- $P_b = \{(ab\$, 1), (\$, 3)\}$, $\text{getlcp}(P_b) = \epsilon$, $|\ \text{getlcp}(P_b)\ | = 0$

# naive construction: example

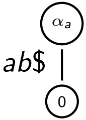
For $s = abab\$$: $U = \{(abab\$, 0), (bab\$, 1), (ab\$, 2), (b\$, 3), (\$, 4)\}$

$\rightarrow$ `first call`

- $P_a = \{(bab\$, 0), (b\$, 2)\}$, $\text{getlcp}(P_a) = \text{b}$, $|\ \text{getlcp}(P_a)\ | = 1$
- $P_b = \{(ab\$, 1), (\$, 3)\}$, $\text{getlcp}(P_b) = \epsilon$, $|\ \text{getlcp}(P_b)\ | = 0$

$\rightarrow$ `second call`

  - $P_a = \{(b\$, 0)\} \Longrightarrow \text{addLeaf}(\alpha_a,\ ab\$,\ 0)$



# naive construction: example

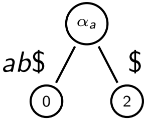
For $s = abab\$$: $U = \{(abab\$, 0), (bab\$, 1), (ab\$, 2), (b\$, 3), (\$, 4)\}$

→ `first call`

- $P_a = \{(bab\$, 0), (b\$, 2)\}$, $\text{getlcp}(P_a) = \text{b}$, $|\ \text{getlcp}(P_a)\ | = 1$
- $P_b = \{(ab\$, 1), (\$, 3)\}$, $\text{getlcp}(P_b) = \epsilon$, $|\ \text{getlcp}(P_b)\ | = 0$

→ `second call`

  - $P_a = \{(b\$, 0)\} \Longrightarrow \text{addLeaf}(\alpha_a,\ ab\$,\ 0)$
  - $P_\$ = \{(\epsilon, 2)\} \Longrightarrow \text{addLeaf}(\alpha_a,\ \$,\ 2)$



# naive construction: example

For $s = abab\$$: $U = \{(abab\$, 0), (bab\$, 1), (ab\$, 2), (b\$, 3), (\$, 4)\}$

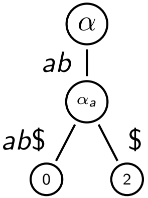
→ `first call`

- $P_a = \{(bab\$, 0), (b\$, 2)\}$, $\text{getlcp}(P_a) = \text{b}$, $|\text{getlcp}(P_a)| = 1$
- $P_b = \{(ab\$, 1), (\$, 3)\}$, $\text{getlcp}(P_b) = \epsilon$, $|\text{getlcp}(P_b)| = 0$

→ `second call`

  - $P_a = \{(b\$, 0)\} \Longrightarrow \text{addLeaf}(\alpha_a,\ ab\$,\ 0)$
  - $P_\$ = \{(\epsilon, 2)\} \Longrightarrow \text{addLeaf}(\alpha_a,\ \$,\ 2)$

- $\Longrightarrow \text{insertEdge}(\alpha,\ ab,\ \alpha_a)$




Can you continue this example?

## naive construction: complexity

The naive construction does not perform very well

- complexity of naive suffix tree construction is $O(n^2)$. Proof?

We can do better:

- Ukkonen has devised an online construction algorithm that runs in O(n).
- McCreight's O(n)-algorithm also runs slightly faster in practice.

# Growing trees

**Question**

Assume you have built a suffix tree for the string

`aba$`

Which modifications are necessary to obtain the suffix tree for

`ababa$`

Can the old tree be of any use?

# McCreight's algorithm

- The idea: successively merge $ST(S\$[i..n])$ with $ST(S\$[i+1..n])$, $i \in [0, n]$
- exploits the properties of longest common prefixes

## Definition (head)

The *head(i)* of some suffix $S\$[i..n]$, $0 \leq i \leq n-1$ is the longest common prefix of with $S\$[j..n]$, where $j < i$.

## Definition (tail)

The *tail(i)* is simply the rest: $S\$[i..n] = head(i)tail(i)$

# heads up: McCreight's trick

To appreciate the trick we need to understand the following observation:

1. Once the suffix tree for $S\$[0..n]..S\$[i..n]$ is build, we need to **update only those parts of ST that are "affected"** by introducing the suffix $S\$[i+1..n]$
2. Only parts beyond the head, ie. within the tail, may be "affected".

# heads up: McCreight's trick

**Beweis.**

Let ST(S) be as suffix tree of a string S. Assume the string $av$ has a prefix implicitly represented in $ST(S)$. Hence, there is a search-path from the root to some edge or node (leaf?)
$\Rightarrow$ the search-path is of length $max_{j \leq |S|} lcp(av, S[j..n]) = \ell$
$\Rightarrow$ $lcp(v, S[j..n]) = k \geq \ell - 1$
$\Rightarrow$ substring $v[0...\ell-1]$ is also implicitly represented by ST(S)
$\Rightarrow$ only suffix tree modifications below internal nodes or edge labels representing $v[\ell-1..n]$ are necessary □

Hence, finding an inexpensive way to locate the head yields an inexpensive algorithm. Let's pretend we've found it:

**magic(head(i))** delivers a position within $ST$ with $p$ representing the substring $S\$[i+1..i+\ell-1]$. $p$ may be on an edge or a vertex.

# linear construction: algorithm M

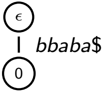
**Require:** tree for the suffix S$[0..n-1]

```
for i in 0 to n − 2 do
    if head(i) = ε then
        head(i+1) := scan(ST(ε), S$[i+1..n])
        if head(i+1) ends in edge then
            addNode(head(i+1))
        end if
        addLeaf(head(i+1), tail(i+1), i+1)
    else
        p := magic(head(i))
        if p ends in edge then
            head(i+1) = p
        else if p ends in vertex then
            head(i+1) = scan(ST(p), tail(i))
        end if
        if head(i + 1) ends in edge then
            addNode(head(i+1))
        end if
        addLeaf(head(i+1), tail(i+1), i+1)
    end if
end for
```

# linear construction: algorithm M

```
Require: tree for the suffix S$[0..n-1]
  for i in 0 to n − 2 do
    if head(i) = ϵ then
      head(i+1) := scan(ST(ϵ), S$[i+1..n])
      if head(i+1) ends in edge then
        addNode(head(i+1))
      end if
      addLeaf(head(i+1), tail(i+1), i+1)
    else
      p := magic(head(i))
      if p ends in edge then
        head(i+1) = p
      else if p ends in vertex then
        head(i+1) = scan(ST(p), tail(i))
      end if
      if head(i + 1) ends in edge then
        addNode(head(i+1))
      end if
      addLeaf(head(i+1), tail(i+1), i+1)
    end if
  end for
```

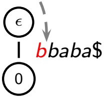
| | |
|---|---|
| i | 0 |
| S[i+1..n] | baba$ |
| head(i) | $\epsilon$ |
| tail(i) | bbaba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) $= p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

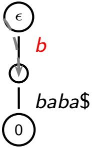
| | |
|---|---|
| i | 0 |
| S[i+1..n] | baba$ |
| head(i) | $\epsilon$ |
| tail(i) | bbaba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

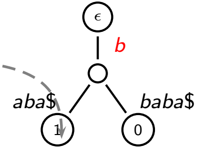
| | |
|---|---|
| i | 0 |
| S[i+1..n] | baba$ |
| head(i) | $\epsilon$ |
| tail(i) | bbaba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) $= p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

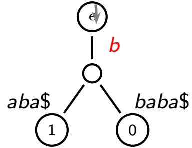
| | |
|---|---|
| i | 0 |
| S[i+1..n] | baba$ |
| head(i) | $\epsilon$ |
| tail(i) | bbaba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) $= p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

| | |
|---|---|
| i | 1 |
| S[i+1..n] | aba$ |
| head(i) | b |
| tail(i) | aba$ |

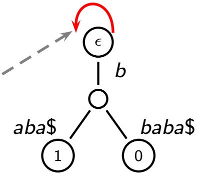



$S\$[i+1..\ell-1]$ not yet in tree
$\Rightarrow$ magic(head(i)) $= \epsilon$

# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

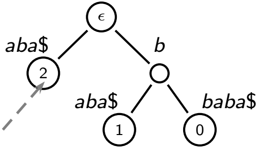
| | |
|---|---|
| i | 1 |
| S[i+1..n] | aba$ |
| head(i) | b |
| tail(i) | aba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S\$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

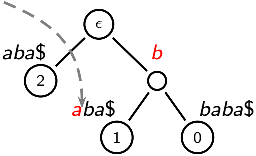
| | |
|---|---|
| i | 1 |
| S[i+1..n] | aba\$ |
| head(i) | b |
| tail(i) | aba\$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
**for** $i$ in 0 to $n-2$ **do**
  **if** head($i$) $= \epsilon$ **then**
    head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
    **if** head($i$+1) ends in edge **then**
      addNode(head($i$+1))
    **end if**
    addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **else**
    $p$ := magic(head($i$))
    **if** $p$ ends in edge **then**
      head($i$+1) = $p$
    **else if** $p$ ends in vertex **then**
      head($i$+1) = scan(ST(p), tail($i$))
    **end if**
    **if** $head(i+1)$ ends in edge **then**
      addNode(head($i$+1))
    **end if**
    addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **end if**
**end for**

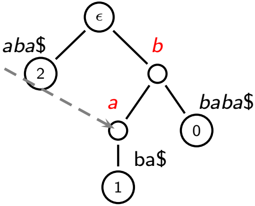
| | |
|---|---|
| i | 2 |
| S[i+1..n] | ba$ |
| head(i) | $\epsilon$ |
| tail(i) | aba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i{+}1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

| | |
|---|---|
| i | 2 |
| S[i+1..n] | ba$ |
| head(i) | $\epsilon$ |
| tail(i) | aba$ |

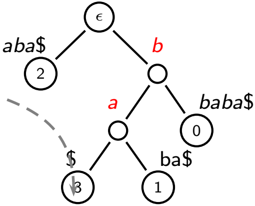


# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]

**for** $i$ in 0 to $n-2$ **do**
  **if** head($i$) $= \epsilon$ **then**
    head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
    **if** head($i$+1) ends in edge **then**
      addNode(head($i$+1))
    **end if**
    addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **else**
    $p$ := magic(head($i$))
    **if** $p$ ends in edge **then**
      head($i$+1) $= p$
    **else if** $p$ ends in vertex **then**
      head($i$+1) = scan(ST(p), tail($i$))
    **end if**
    **if** $head(i+1)$ ends in edge **then**
      addNode(head($i$+1))
    **end if**
    addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **end if**
**end for**

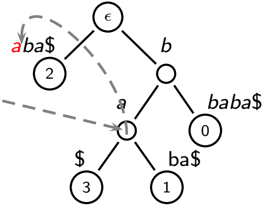
| | |
|---|---|
| i | 2 |
| S[i+1..n] | ba$ |
| head(i) | $\epsilon$ |
| tail(i) | aba$ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i{+}1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) $= p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

| | |
|---|---|
| i | 3 |
| S[i+1..n] | a$ |
| head(i) | ba |
| tail(i) | $ |

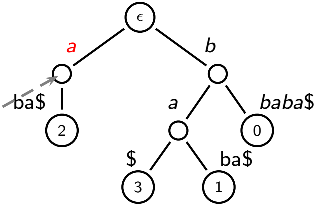



$S\$[i+1..\ell-1]$=a in tree $\Rightarrow$
p points to $a$-edge of root

# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
 **for** $i$ in 0 to $n-2$ **do**
  **if** head($i$) $= \epsilon$ **then**
   head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
   **if** head($i$+1) ends in edge **then**
    addNode(head($i$+1))
   **end if**
   addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **else**
   $p$ := magic(head($i$))
   **if** $p$ ends in edge **then**
    head($i$+1) = $p$
   **else if** $p$ ends in vertex **then**
    head($i$+1) = scan(ST(p), tail($i$))
   **end if**
   **if** $head(i+1)$ ends in edge **then**
    addNode(head($i$+1))
   **end if**
   addLeaf(head($i$+1), tail($i$+1), $i$+1)
  **end if**
 **end for**

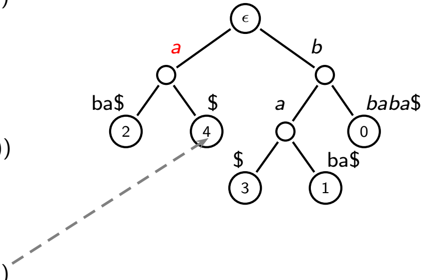
| | |
|---|---|
| i | 3 |
| S[i+1..n] | a$ |
| head(i) | ba |
| tail(i) | $ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** $head(i+1)$ ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

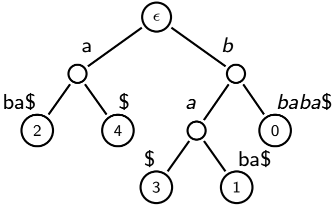
| | |
|---|---|
| i | 3 |
| S[i+1..n] | a$ |
| head(i) | ba |
| tail(i) | $ |



# linear construction: algorithm M

**Require:** tree for the suffix S$[0..n-1]
  **for** $i$ in 0 to $n-2$ **do**
    **if** head($i$) $= \epsilon$ **then**
      head($i$+1) := scan($ST(\epsilon)$, $S\$[i+1..n]$)
      **if** head($i$+1) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **else**
      $p$ := magic(head($i$))
      **if** $p$ ends in edge **then**
        head($i$+1) = $p$
      **else if** $p$ ends in vertex **then**
        head($i$+1) = scan(ST(p), tail($i$))
      **end if**
      **if** *head*($i+1$) ends in edge **then**
        addNode(head($i$+1))
      **end if**
      addLeaf(head($i$+1), tail($i$+1), $i$+1)
    **end if**
  **end for**

| | |
|---|---|
| i | 4 |
| S[i+1..n] | $ |
| head(i) | a |
| tail(i) | $ |




What will happen now?

# magic

For some head(i)=$S\$[i..i+\ell]$, the function **magic(head(i))** delivers a position $p \in ST(S)$ with $p$ representing the substring $S\$[i+1..i+\ell-1]$. It is realized using **suffix links**:

- suffix links are only defined on nodes
- to use suffix link, go back to parent node
- while going back remember all edge labels
- jump to another node via suffix link
- rematch the edge labels
- done. we reached the location for $S\$[i+1..i+\ell-1]$.

Suffix links can be updated during the construction of the suffix tree with algorithm M.

# Suffix trees: a stringology toolbox

1. string search
2. longest common substrings (two strings)
3. longest repeated substrings (one string)
4. for each suffix of a pattern, get length of the longest match
5. shortest unique substrings
6. ...

# Abschlussveranstaltung SWT-Praktikum 2011

Im SWT-Praktikum stellen die studentischen Teams im 4. Semester ihre Fähigkeiten unter Beweis, ein größeres Software-Projekt im Umfang von etwa 1 000 Mannstunden „nach den Regeln der Kunst“ gemeinschaftlich zu planen und umzusetzen.
In der **Abschlussveranstaltung**

**am 7. Juli 2011, 9:15 bis 10:45 Uhr im Hs 10**

präsentieren die Teams in Vorträgen und Demonstrationen von je etwa 10 Minuten die Ergebnisse ihrer Arbeit der Öffentlichkeit.

*Dazu sind alle Studenten des ersten Studienjahres, die sich bereits jetzt über die Anforderungen und Ergebnisse des SWT-Praktikums im 4. Semester informieren wollen, herzlich eingeladen.*