Algorithmic Challenges: From Suffix Array to Suffix Tree

Michael Levin

Department of Computer Science and Engineering
University of California, San Diego

String Processing and Pattern Matching Algorithms
Algorithms and Data Structures

Outline

Construct suffix Tree

Input: String S

Output: Suffix tree of S

- You already know how to construct suffix tree
- But $O(|S|^2)$ will only work for short strings
- You will learn to build it in $O(|S| \log |S|)$ which enables very long texts!

General Plan

- Construct suffix array in $O(|S| \log |S|)$
- Compute additional information in $O(|S|)$
- Construct suffix tree from suffix array and additional information in $O(|S|)$

Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

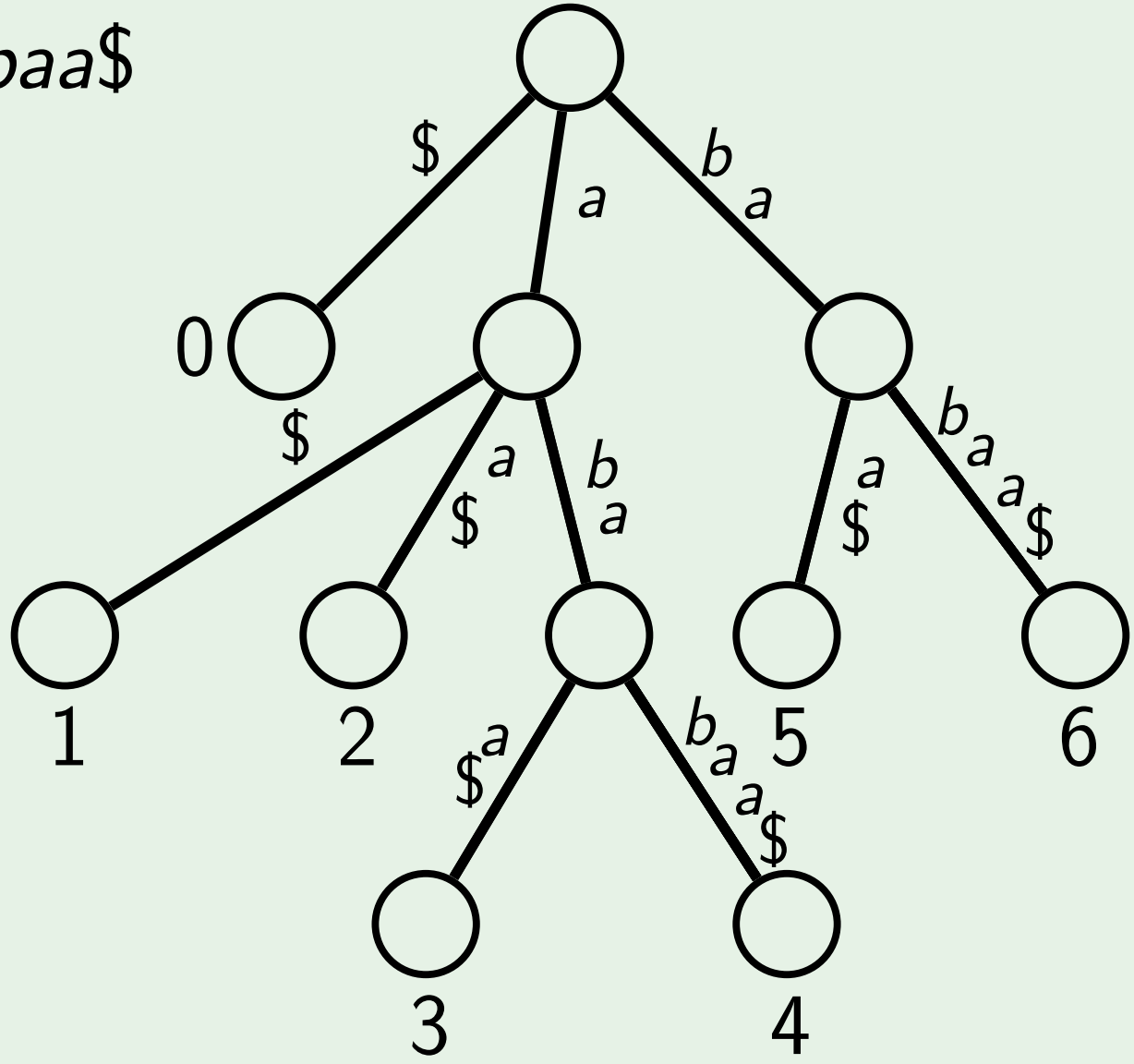
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

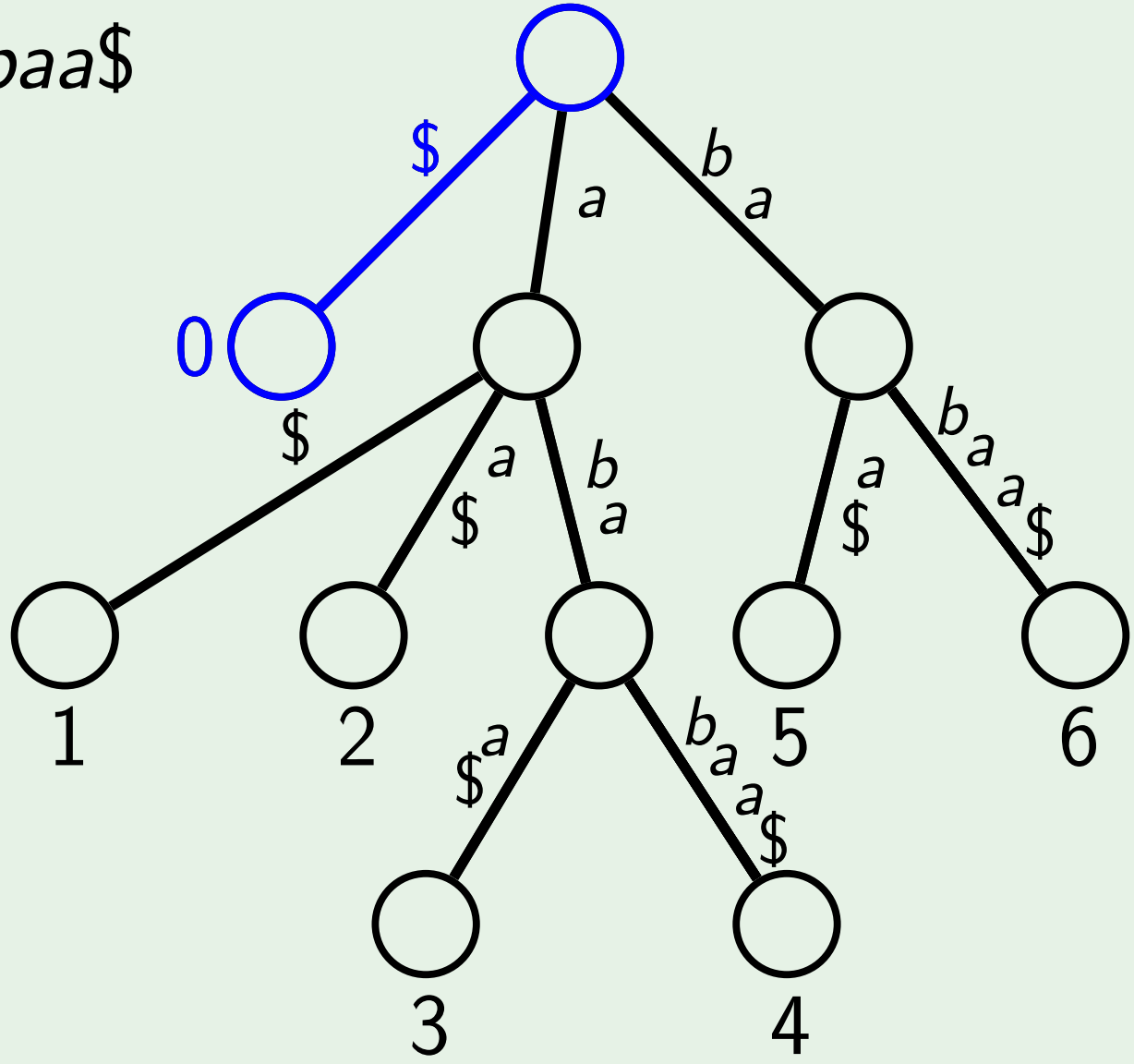
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

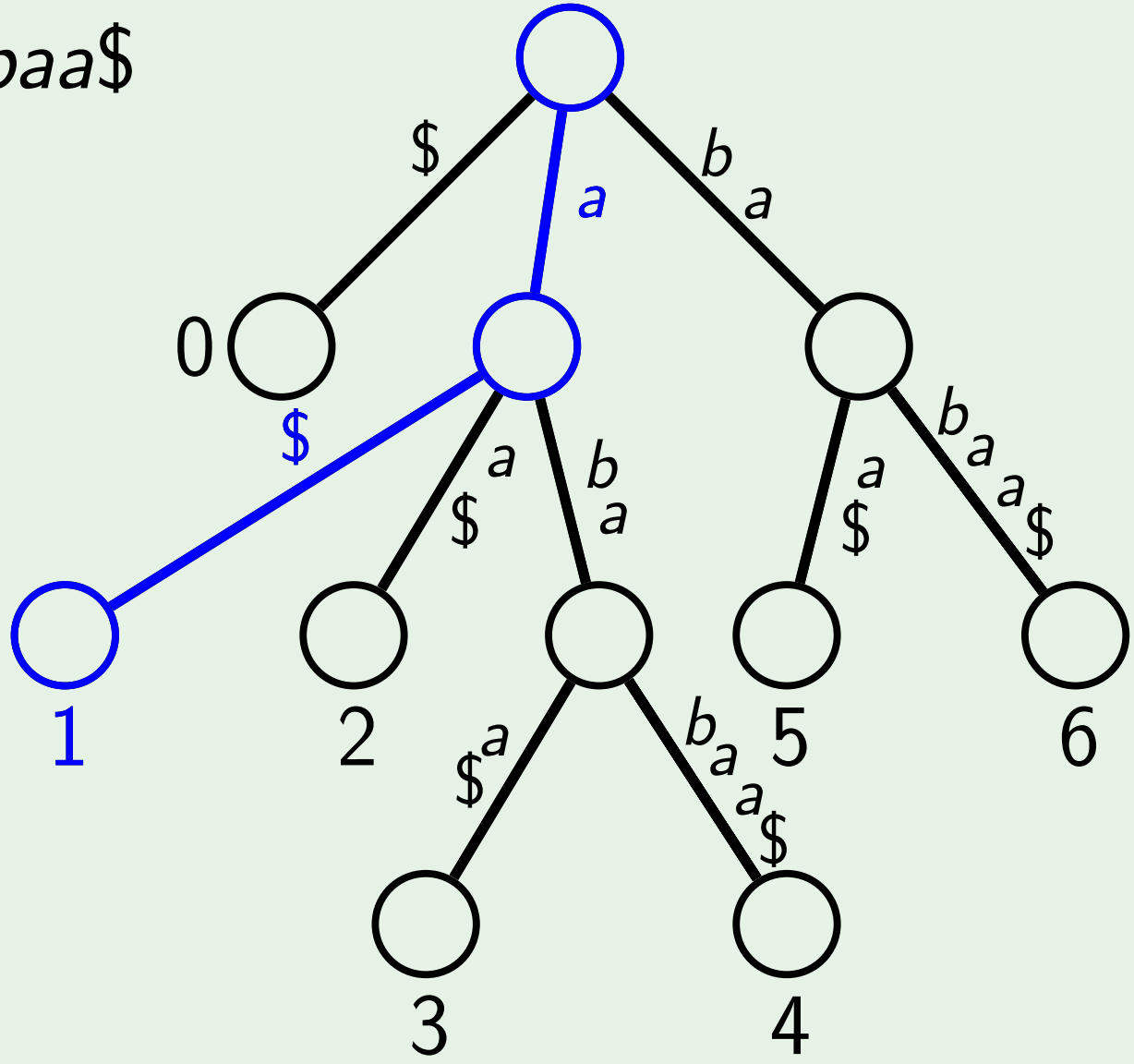
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

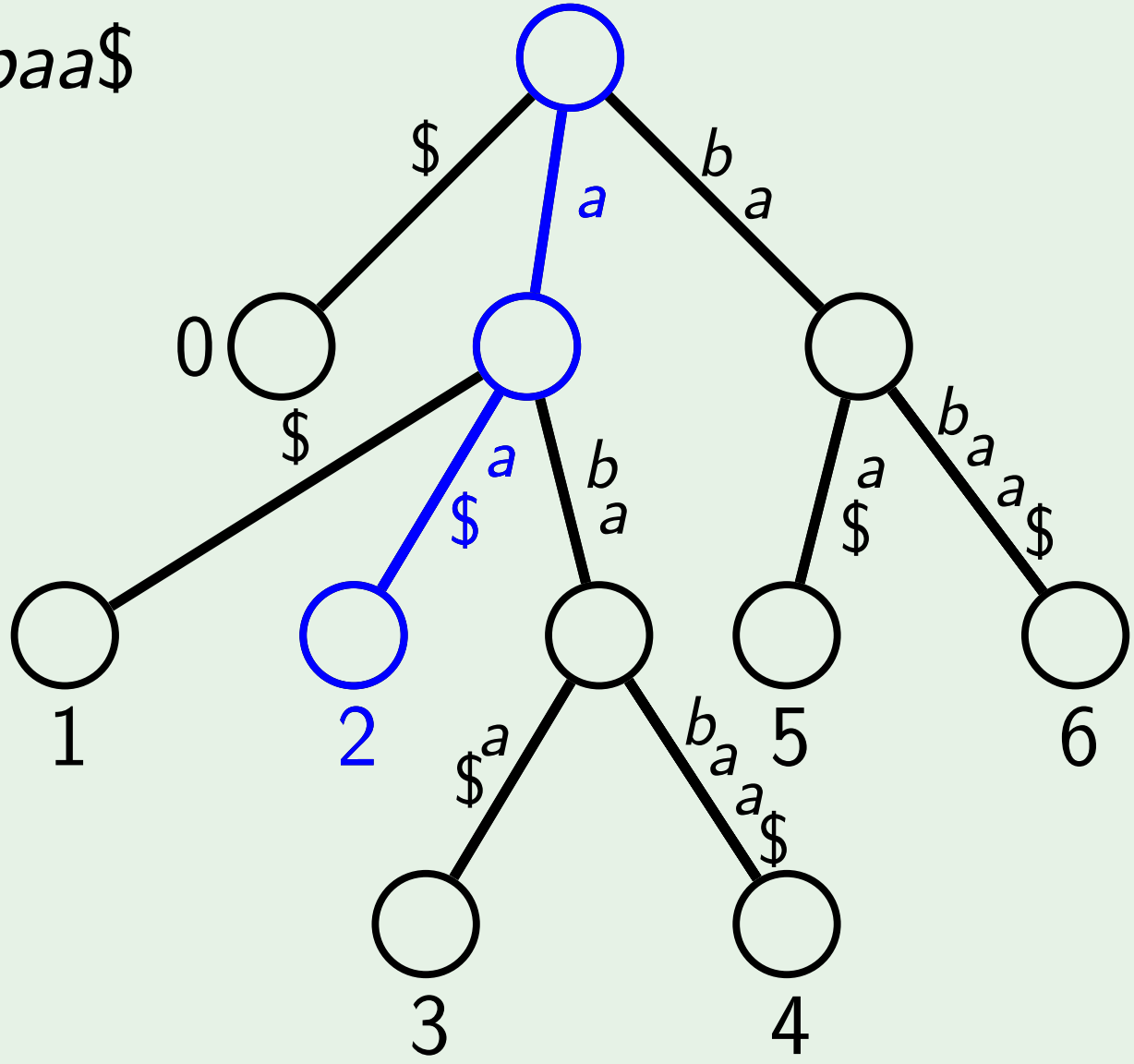
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

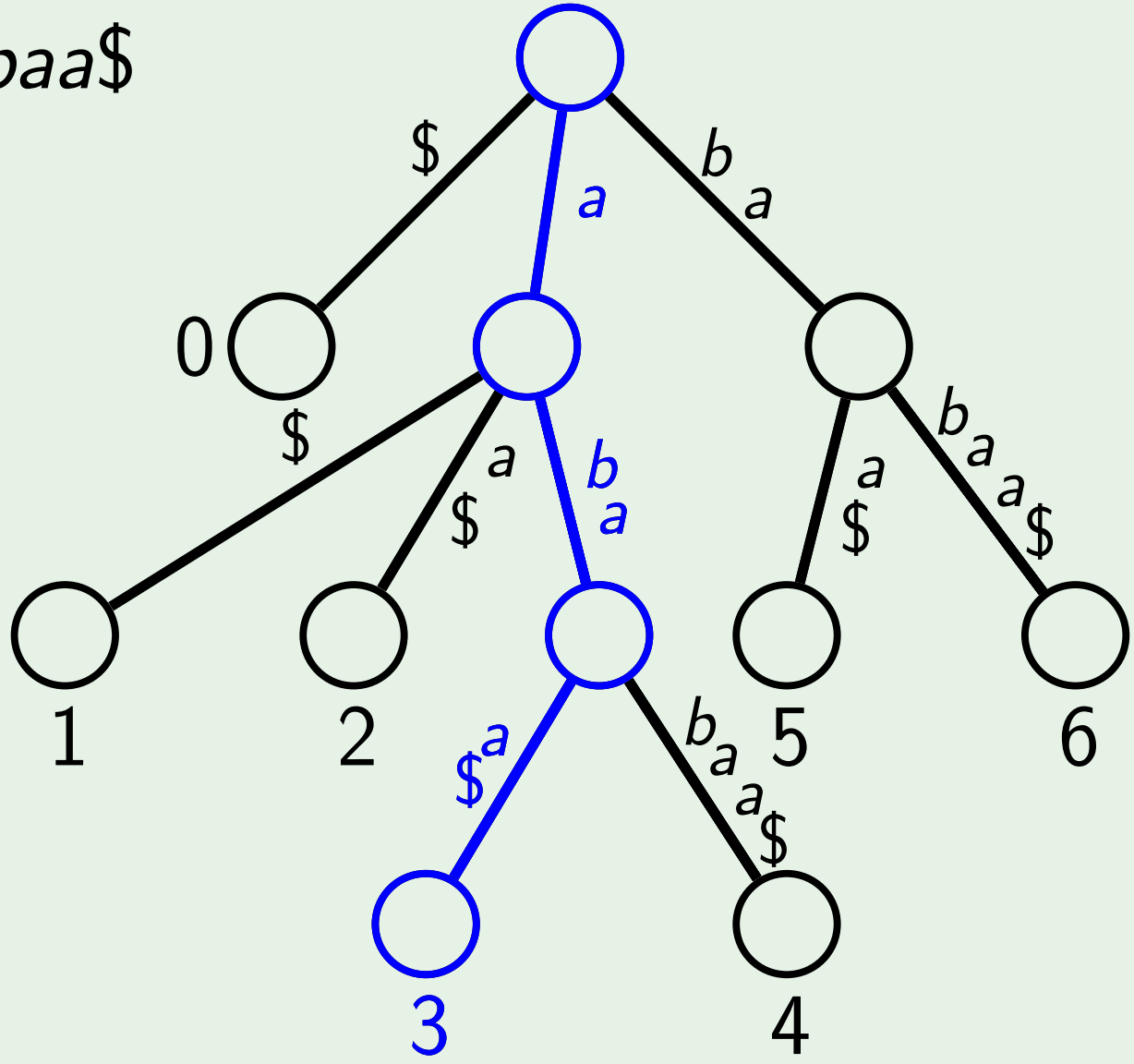
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

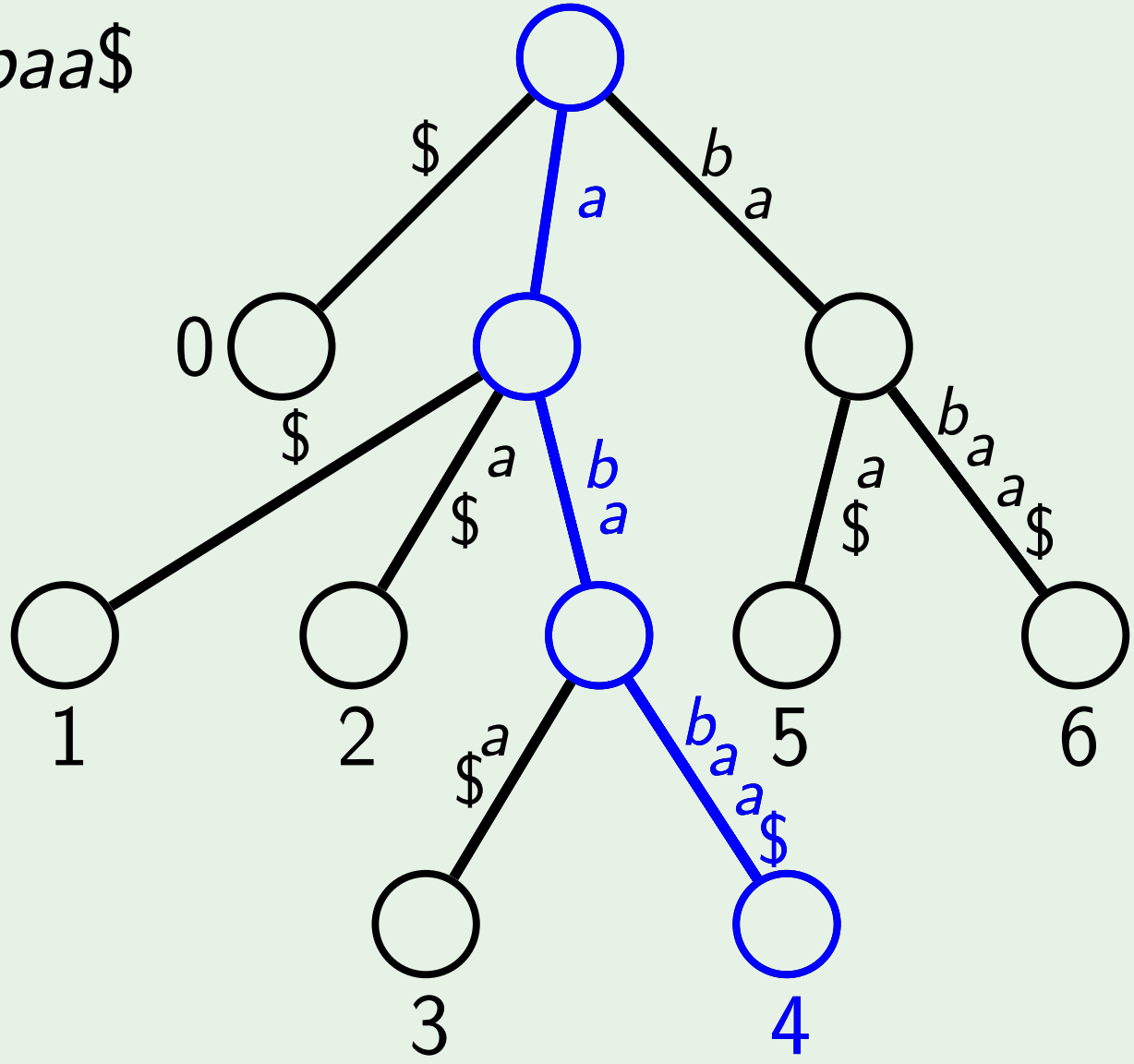
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

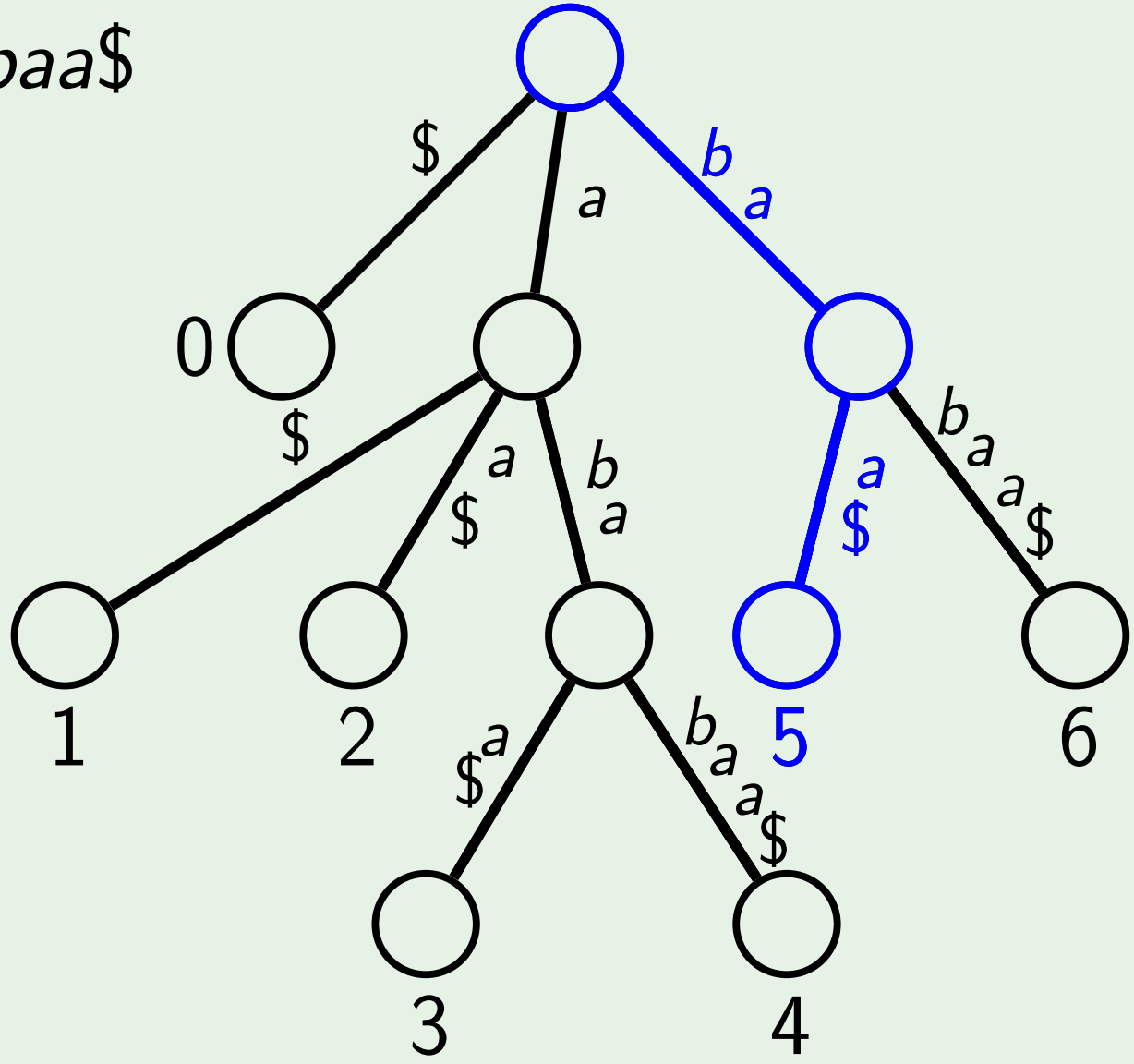
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array and suffix tree

$S = ababaa\$$

0 $

1 $a\$$

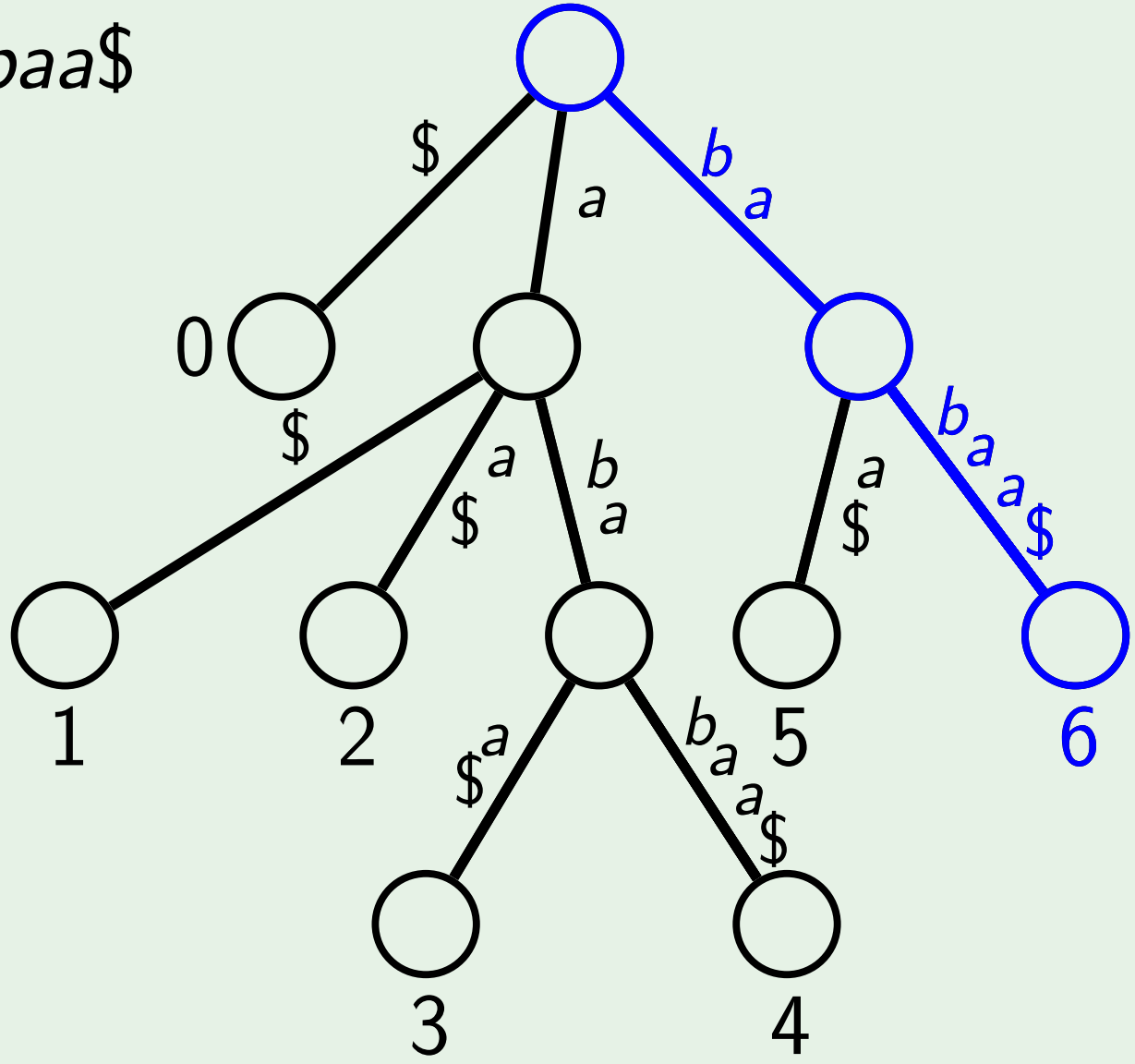
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Definition

The longest common prefix (or just "lcp") of two strings S and T is the longest such string u that u is both a prefix of S and T. We denote by $LCP(S, T)$ the length of the "lcp" of S and T.

Example

$LCP(\text{“ababc”}, \text{“abc”}) = 2$

$LCP(\text{“a”}, \text{“b”}) = 0$

Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 $a\$$

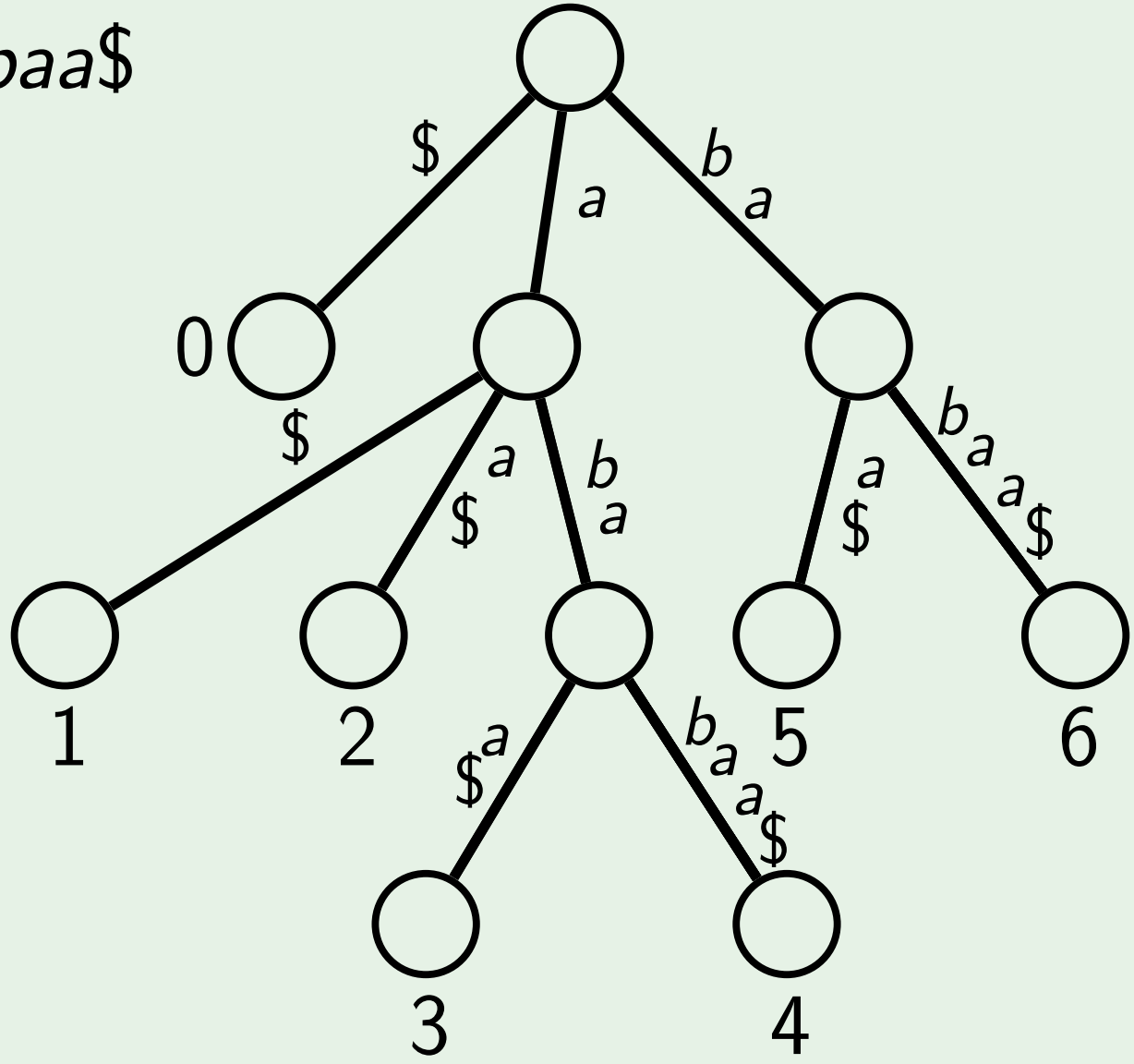
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 $a\$$

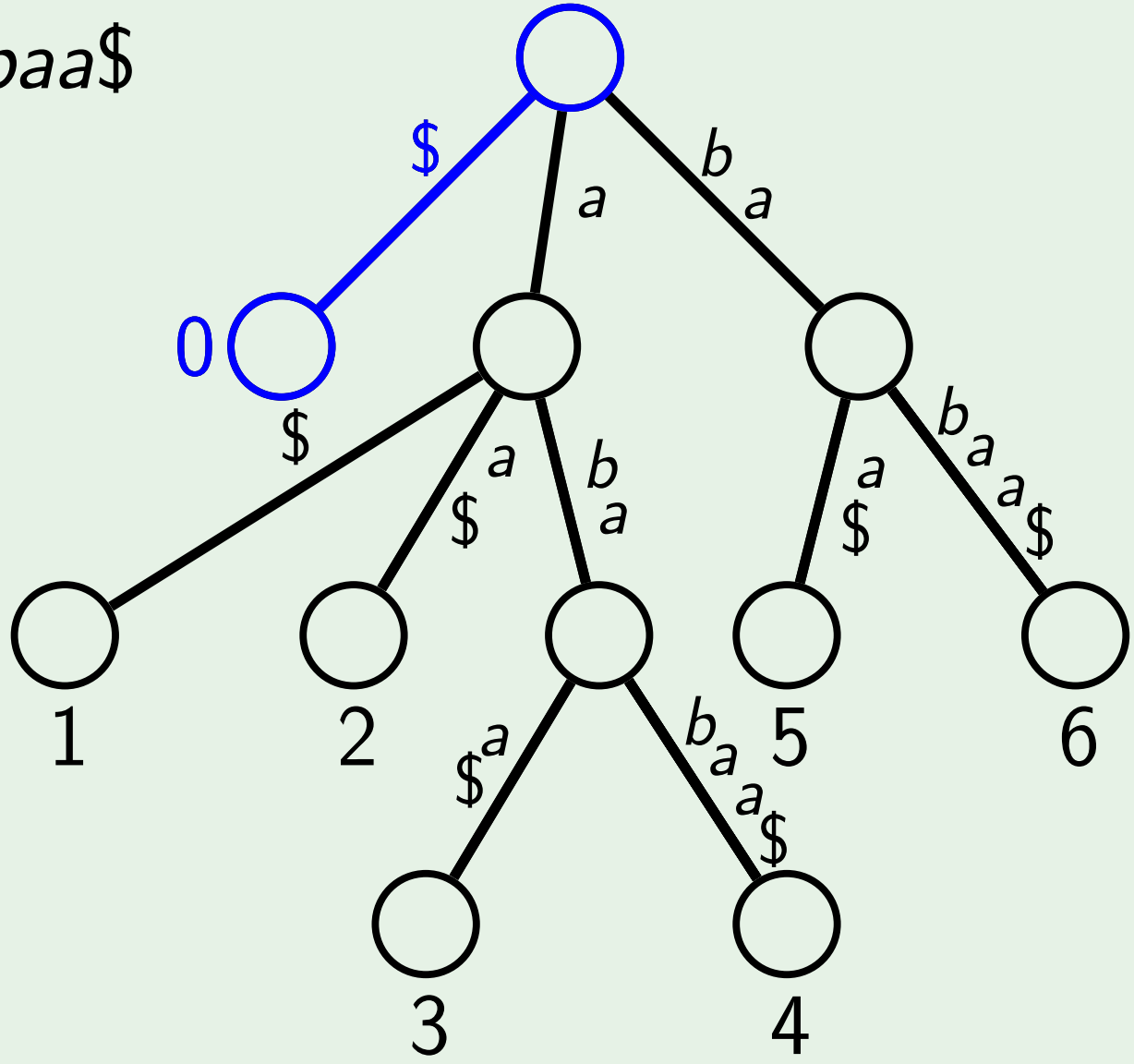
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 a$

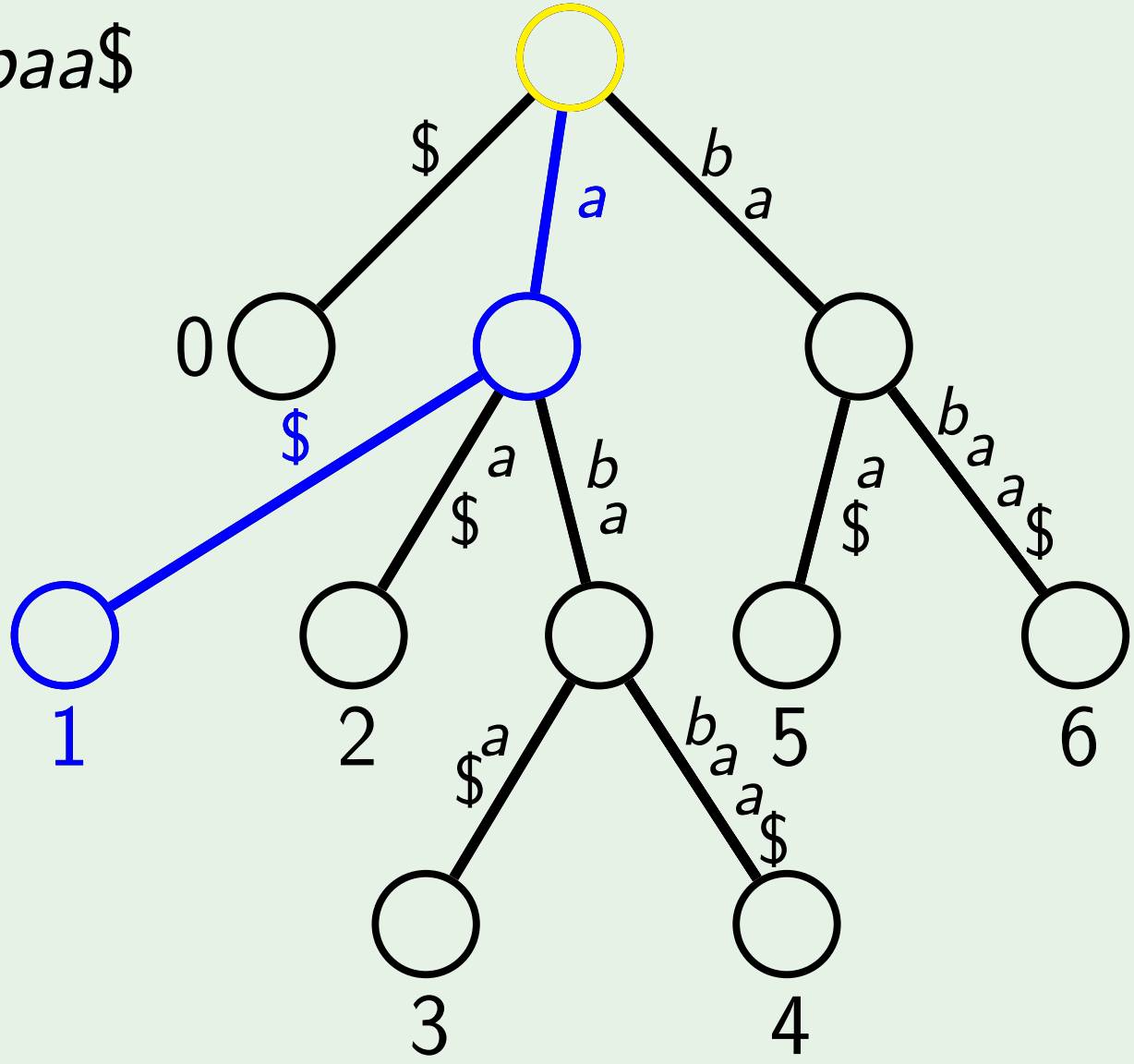
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 a$

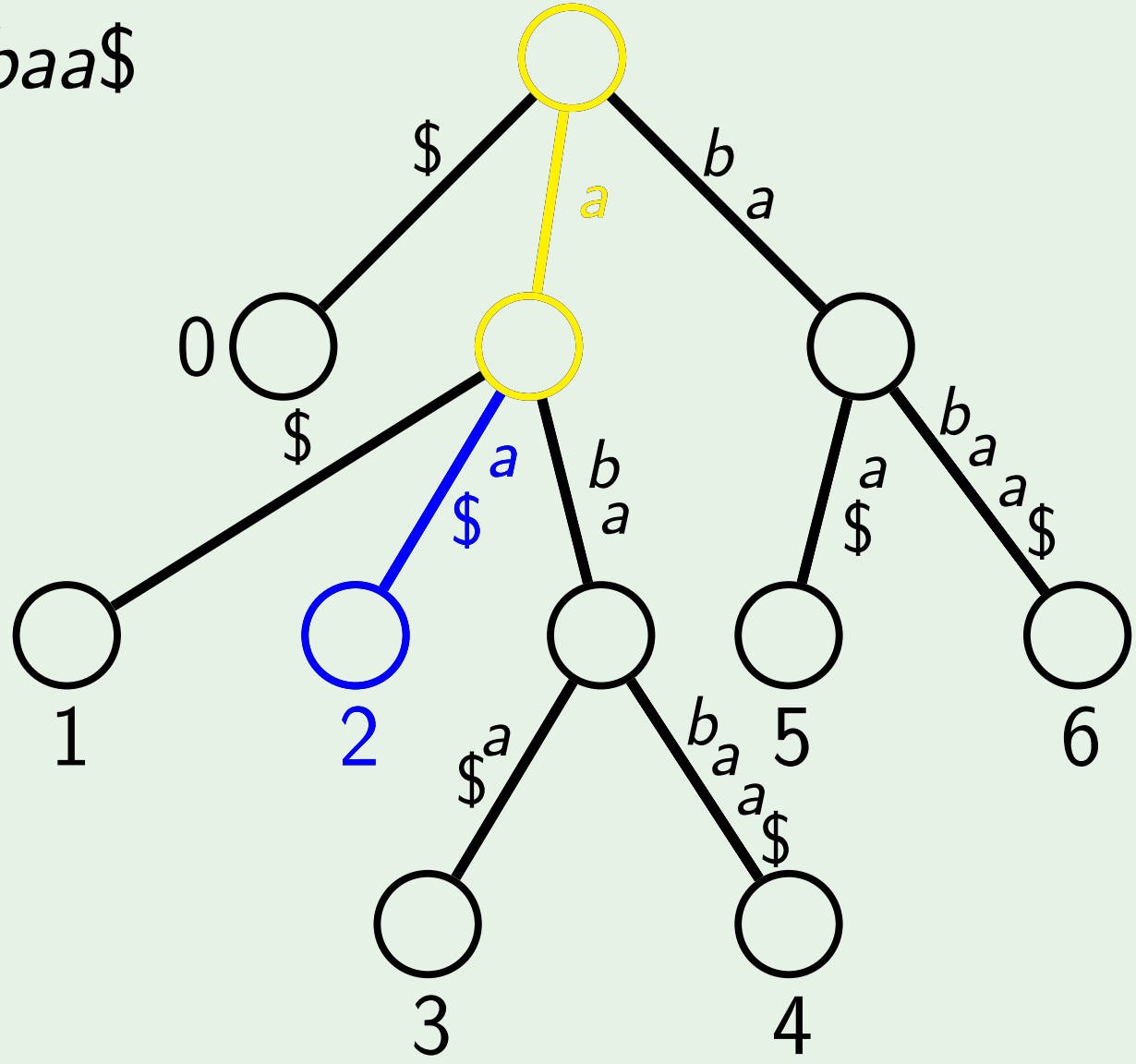
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 a$

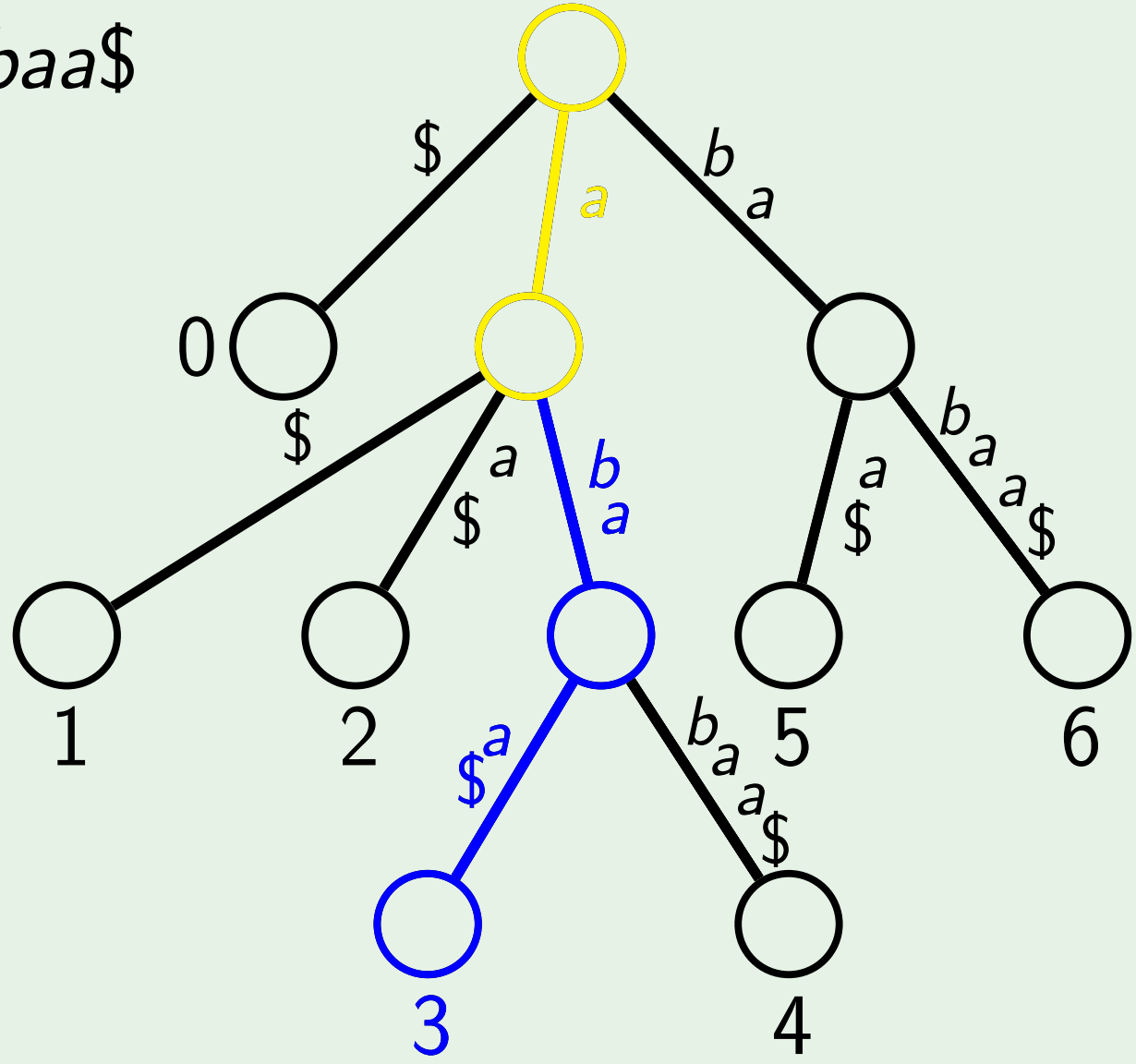
2 aa$

3 abaa$

4 ababaa$

5 baa$

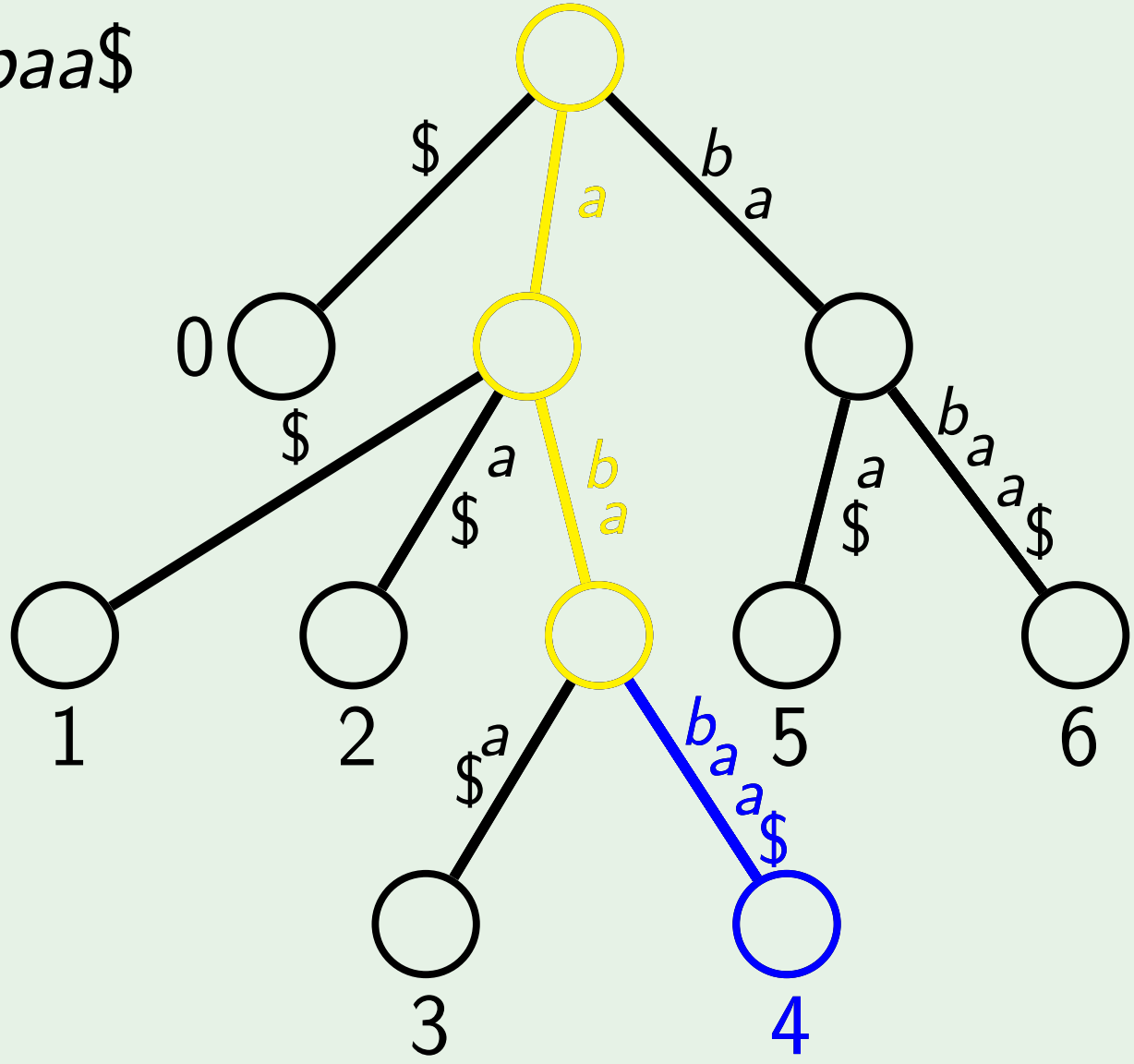
6 babaa$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0	$
1	a$
2	aa$
3	aba$
4	ababaa$
5	baa$
6	babaa$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 $a\$$

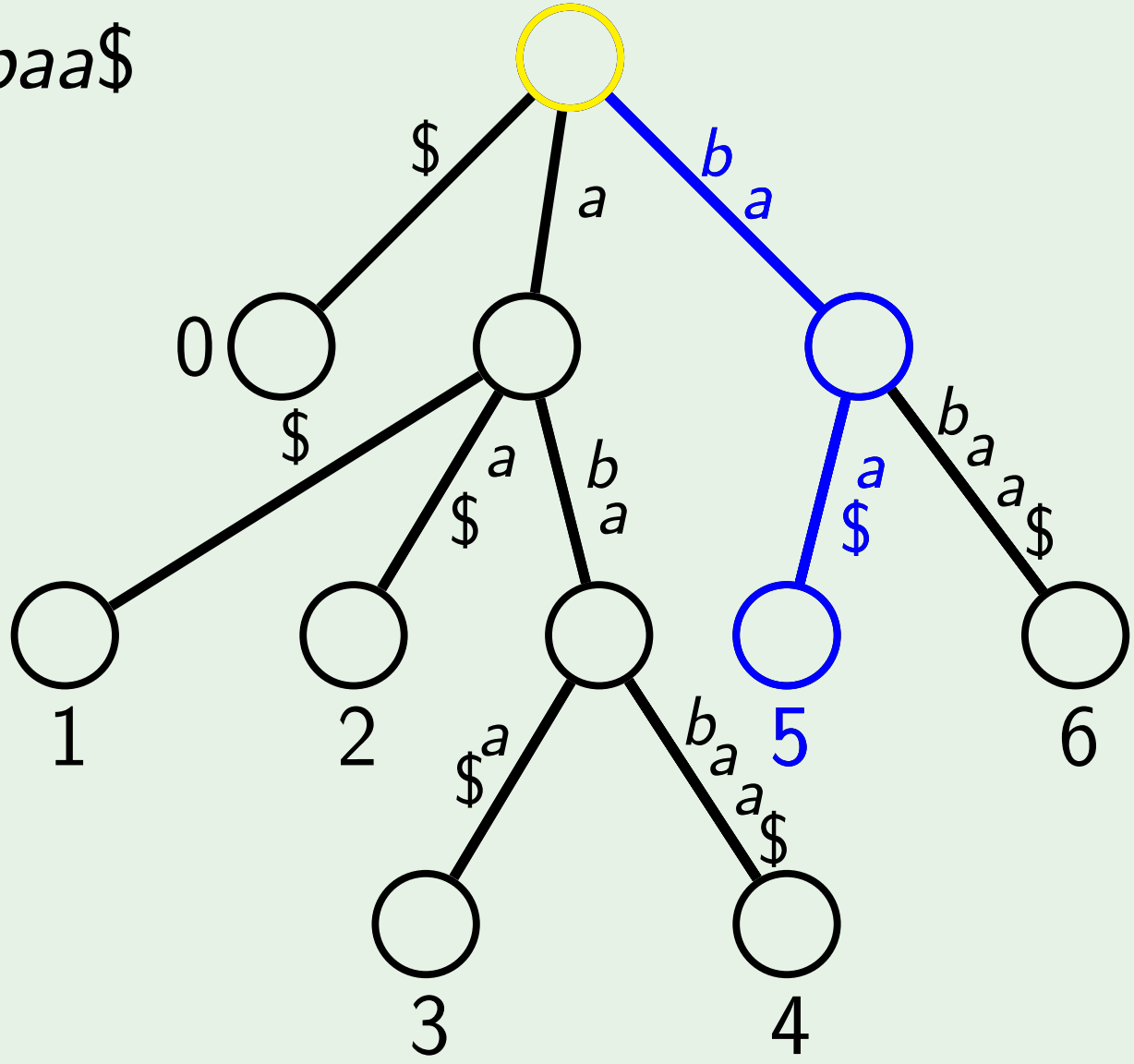
2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$



Suffix array, suffix tree and lcp

$S = ababaa\$$

0 $

1 a$

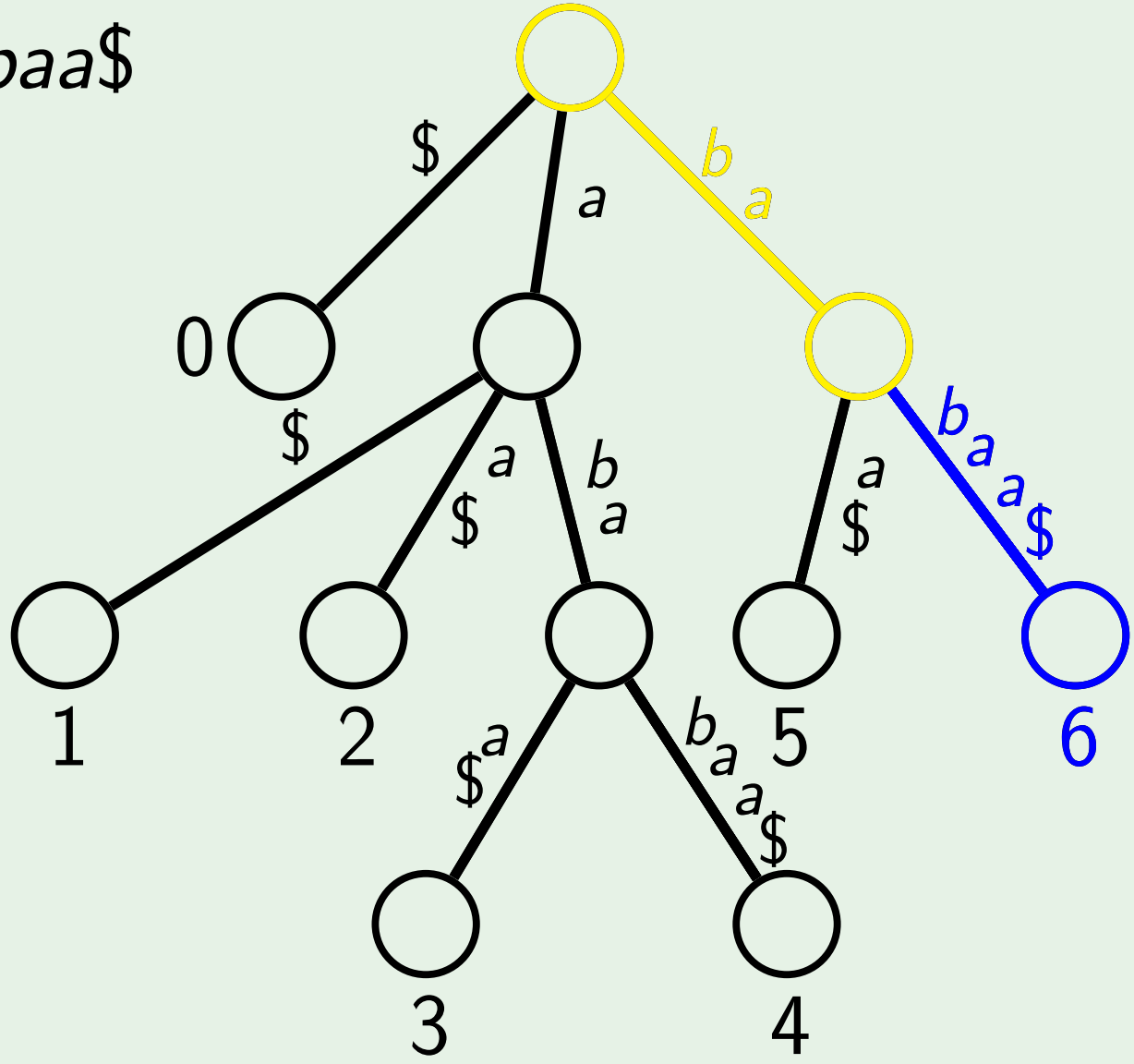
2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$



Outline

LCP array

Definition

Consider suffix array A of string S in the raw form, that is
$A[0] < A[1] < A[2] < \cdots < A[|S| - 1]$ are all the suffixes of S in lexicographic order.
LCP array of string S is the array lcp of size $|S| - 1$ such that for each i such that $0 \leq i \leq |S| - 2$,

$$lcp[i] = LCP(A[i], A[i + 1])$$

LCP array

$S = ababaa\$$

0 $

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [\ ,\ ,\ ,\ ,\ ,\ ]$

LCP array

$S = ababaa\$$

0 $

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [\ ,\ ,\ ,\ ,\ ,\ ]$

LCP array

$S = ababaa\$$

0	$
1	a$
2	aa$
3	abaa$
4	ababaa$
5	baa$
6	babaa$

$lcp = [0, \ , \ , \ , \ , \ ]$

LCP array

$S = ababaa\$$

0 $

1 a$

2 aa$

3 abaa$

4 ababaa$

5 baa$

6 babaa$

$lcp = [0, 1, \ , \ , \ , \ ]$

LCP array

$S = ababaa\$$

0 $

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [0, 1, 1, \ , \ , \ ]$

LCP array

$S = ababaa\$$

0	$
1	a
2	aa
3	aba
4	$abab$
5	baa
6	$babaa$

$lcp = [0, 1, 1, 3, \ , \ ]$

LCP array

$S = ababaa\$$

0 $

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [0, 1, 1, 3, 0, \ ]$

LCP array

$S = ababaa\$$

0 $

1 $a\$$

2 $aa\$$

3 $abaa\$$

4 $ababaa\$$

5 $baa\$$

6 $babaa\$$

$lcp = [0, 1, 1, 3, 0, 2]$

LCP array property

Lemma

For any $i < j$, $LCP(A[i], A[j]) \leq lcp[i]$ and $LCP(A[i], A[j]) \leq lcp[j-1]$.

Proof

$\ldots$

i $ababababa$

$i+1$ $xxxxxxxxx$

$\ldots$

j $abbcabab$

If $LCP(A[i], A[j]) > LCP(A[i], A[i+1])$

Proof

$\ldots$

i $ababababa$

$i+1$ $xxxxxxxxx$ $k = 1$

$\ldots$

j $abbcababab$

If $LCP(A[i], A[j]) > LCP(A[i], A[i+1])$
Consider $k = LCP(A[i], A[i+1])$

Proof

$\ldots$

i ab*abababa*

$i+1$ $a\_$ $k = 1$

$\ldots$

j ab*bcabab*

If $k \geq |A[i+1]|$, then $A[i+1] < A[i]$ – contradiction

Proof

$\ldots$

i $ababababa$

$i+1$ $axxxxxxxx$ $k = 1$

$\ldots$

j $abbcabab$

Otherwise $A[j][k] = A[i][k] \neq A[i+1][k]$

Proof

$\ldots$

i $ababababa$

$i+1$ $acxxxxxxx$ $k = 1$

$\ldots \bigvee$

j $abbcabab$

If $A[j][k] = A[i][k] < A[i+1][k]$, then $A[j] < A[i+1]$ — contradiction

Proof

$\ldots$

i $ab$$abababa$

$\vee$

$i+1$ $aa$$xxxxxxxx$ $k = 1$

$\ldots$

j $ab$$bcabab$

If $A[i][k] > A[i+1][k]$, then $A[i] > A[i+1]$

— contradiction $\square$

Computing LCP array

- For each i, compute $LCP(A[i], A[i+1])$ via comparing $A[i]$ and $A[i+1]$ character-by-character
- $O(|S|)$ for each i, $O(|S|)$ different i — total time $O(|S|^2)$
- How to do this faster?

Outline

Idea

$S = abracadabra\$$

index	sorted suffixes	LCP
...	...	...
$i + 1 = 10$	a$	
7	abra$	
...	...	...
$j + 1 = 3$	acadabra$	
...	...	...
$i = 9$	ra$	
$j = 2$	racadabra$	

Idea

$S = abracadabra\$$

index	sorted suffixes	LCP
...	...	...
$i + 1 = 10$	a$	
7	abra$	
...	...	...
$j + 1 = 3$	acadabra$	
...	...	...
$i = 9$	ra$	$h = 2$
$j = 2$	racadabra$	

Idea

$S = abracadabra\$$

index	sorted suffixes	LCP
...	...	...
$i + 1 = 10$	a$	
7	abra$	
...	...	...
$j + 1 = 3$	acadabra$	
...	...	...
$i = 9$	ra$	$h = 2$
$j = 2$	racadabra$	

Idea

$S = abracadabra\$$

index	sorted suffixes	LCP
...	...	...
$i + 1 = 10$	a$	$\geq h - 1$
7	abra$	
...	...	...
$j + 1 = 3$	acadabra$	
...	...	...
$i = 9$	ra$	$h = 2$
$j = 2$	racadabra$	

Notation

Let $S_{a(i)}$ be the next suffix after S_i in the suffix array of string S.

Example

$S =$ abracadabra\$

$S_9 =$ ra\$, $S_{a(9)} = S_2 =$ racadabra\$

$S_{10} = a\$$, $S_{a(10)} = S_7 = abra\$$

Idea

Lemma

$$LCP(S_{i+1}, S_{a(i+1)}) \geq LCP(S_i, S_{a(i)}) - 1$$

Algorithm

- Let the smallest suffix be S_k
- Compute $LCP(S_k, S_{a(k)})$ directly, save result as *lcp*
- Then compute $LCP(S_{k+1}, S_{a(k+1)})$ directly, but start comparisons from position $lcp - 1$, as the first *lcp* characters are definitely equal
- Save the result as *lcp*
- Repeat with $LCP(S_{k+2}, S_{a(k+2)})$ and so on until all LCP values are computed

Outline

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

$lcp \leftarrow \max(0, equal)$

```
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

LCPOfSuffixes($S, i, j, equal$)

```
lcp ← max(0, equal)
while i + lcp < |S| and j + lcp < |S|:
    if S[i + lcp] == S[j + lcp]:
        lcp ← lcp + 1
    else:
        break
return lcp
```

InvertSuffixArray(*order*)

```
pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos
```

InvertSuffixArray(*order*)

```
pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos
```

InvertSuffixArray(*order*)

```
pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos
```

InvertSuffixArray(*order*)

```
pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos
```

InvertSuffixArray(*order*)

```
pos ← array of size |order|
for i from 0 to |pos| − 1:
    pos[order[i]] ← i
return pos
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

ComputeLCPArray(S, *order*)

```
lcpArray ← array of size |S| − 1
lcp ← 0
posInOrder ← InvertSuffixArray(order)
suffix ← order[0]
for i from 0 to |S| − 1:
    orderIndex ← posInOrder[suffix]
    if orderIndex == |S| − 1:
        lcp ← 0
        suffix ← (suffix + 1) mod |S|
        continue
    nextSuffix ← order[orderIndex + 1]
    lcp ← LCPOfSuffixes(S, suffix, nextSuffix, lcp − 1)
    lcpArray[orderIndex] ← lcp
    suffix ← (suffix + 1) mod |S|
return lcpArray
```

Analysis

Lemma

This algorithm computes LCP array in $O(|S|)$

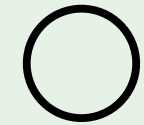
Proof

- Each comparison increases lcp
- $lcp \leq |S|$
- Each iteration lcp decreases by at most 1
- Number of comparisons is $O(|S|)$ □

Outline

Building suffix tree

$S = ababaa\$$



6 $

5 $a\$$

4 $aa\$$

2 $abaa\$$

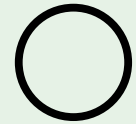
0 $ababaa\$$

3 $baa\$$

1 $babaa\$$

Building suffix tree

$S = ababaa\$$



6 $

5 $a\$$

4 $aa\$$

2 $abaa\$$

0 $ababaa\$$

3 $baa\$$

1 $babaa\$$

Building suffix tree

$S = ababaa\$$

6 $

5 $a\$$

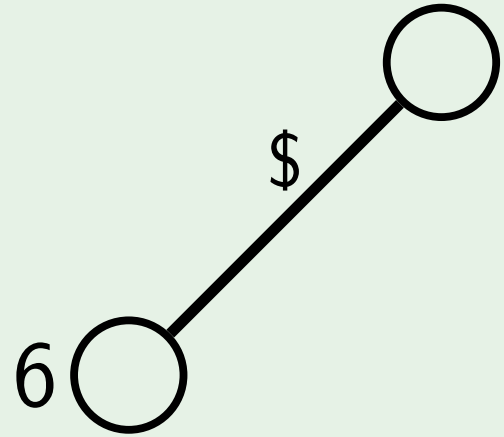
4 $aa\$$

2 $abaa\$$

0 $ababaa\$$

3 $baa\$$

1 $babaa\$$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

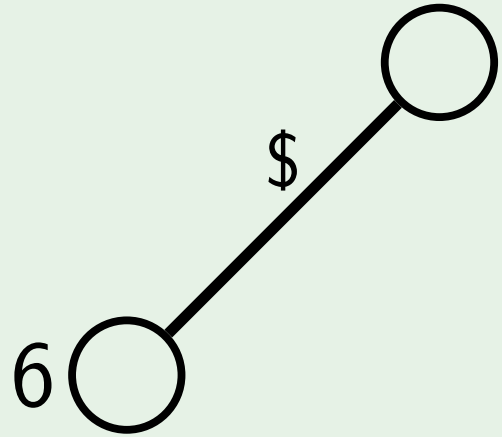
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

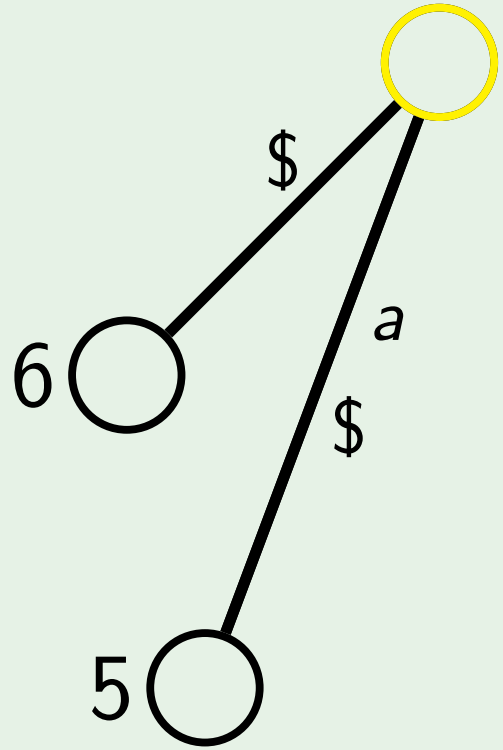
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a

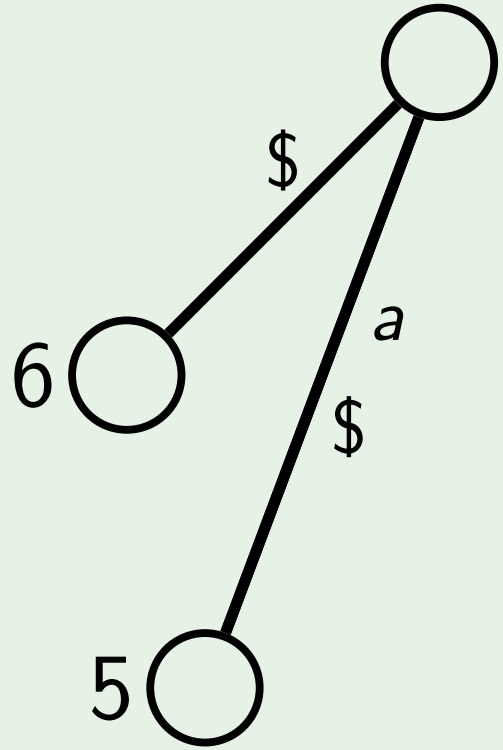
4 aa

2 $abaa$

0 $ababaa$

3 baa

1 $babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

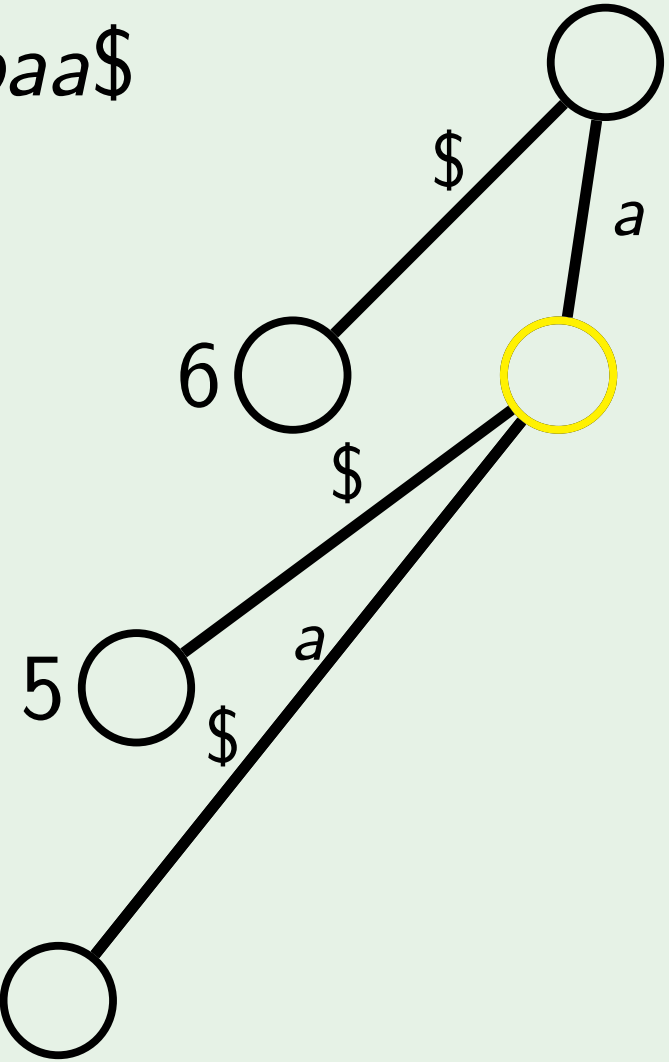
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

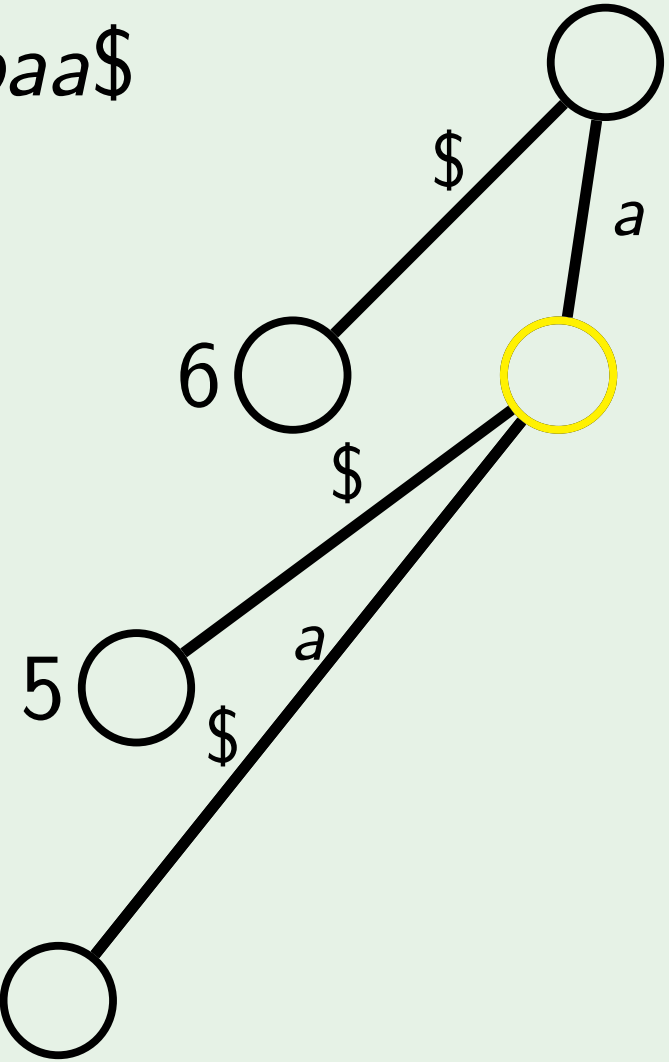
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

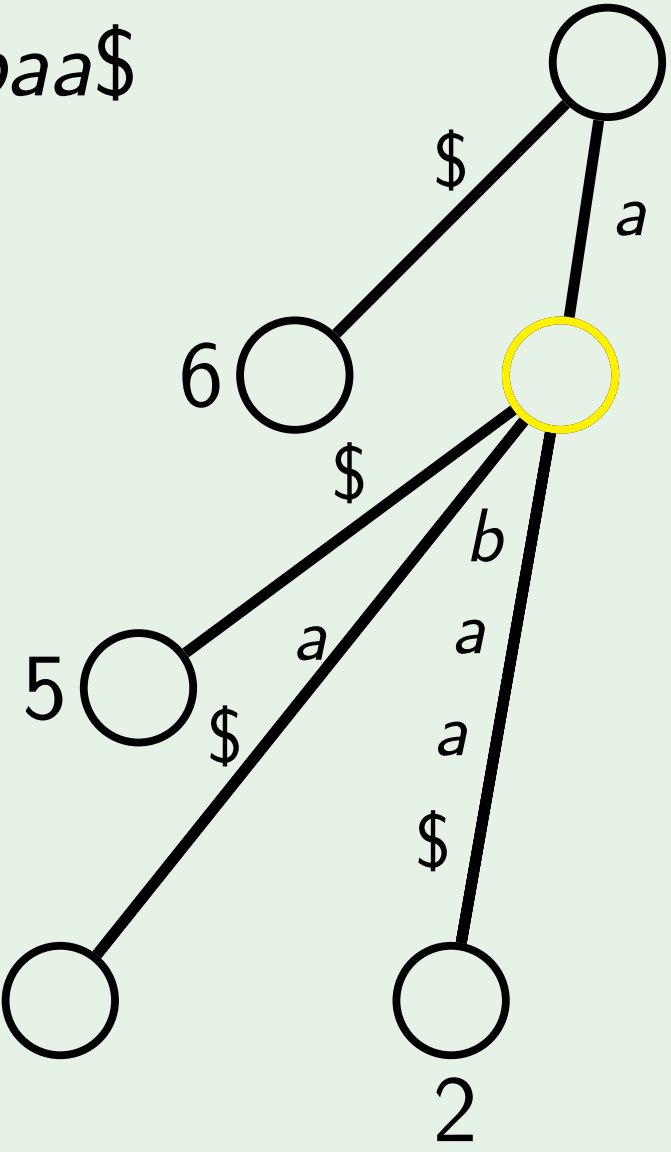
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

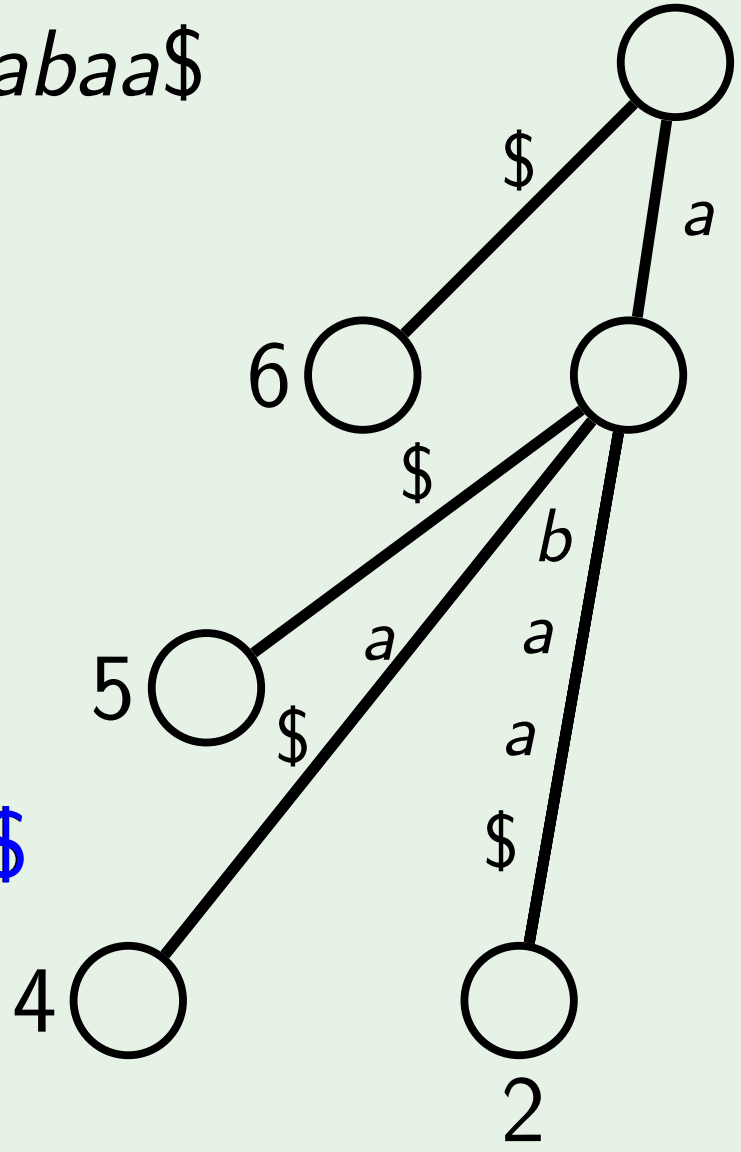
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

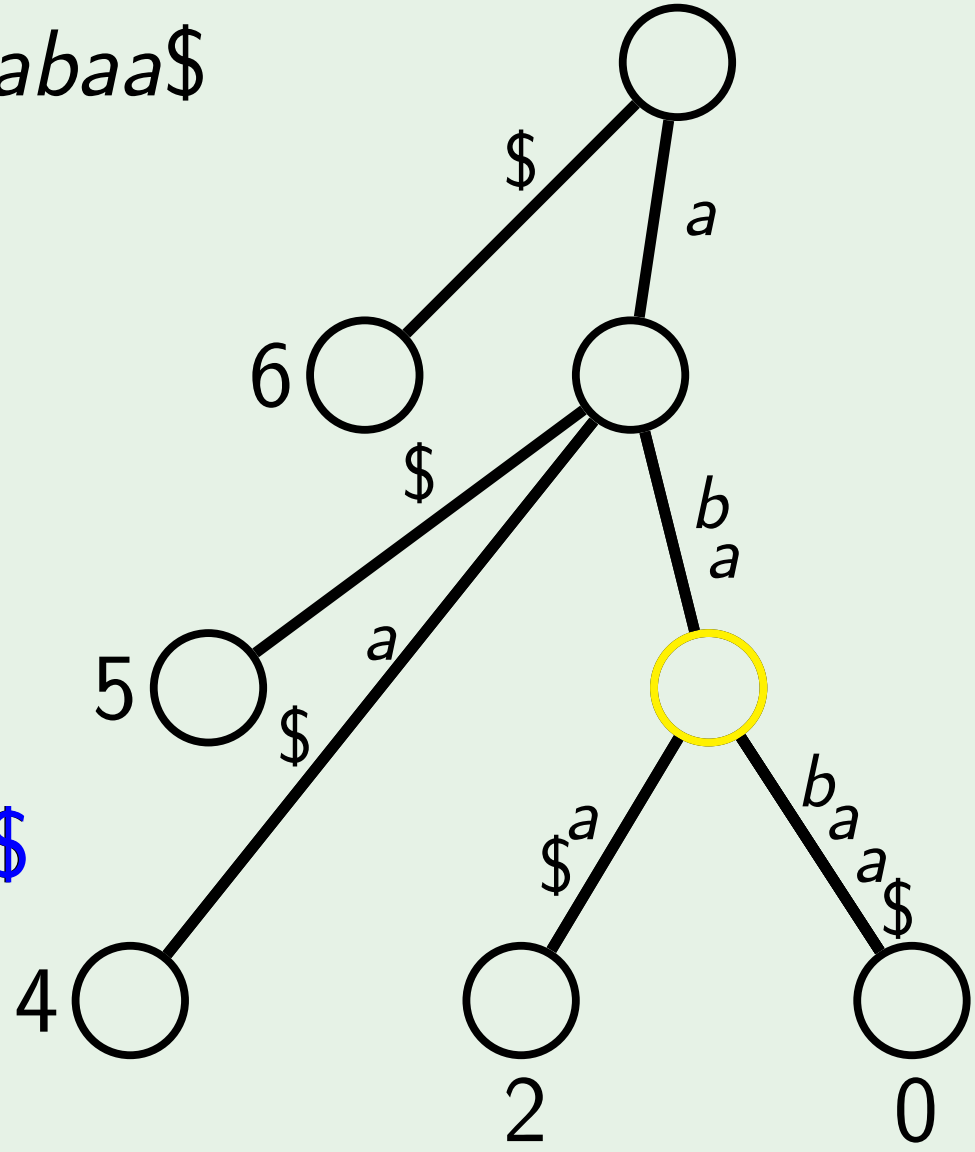
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

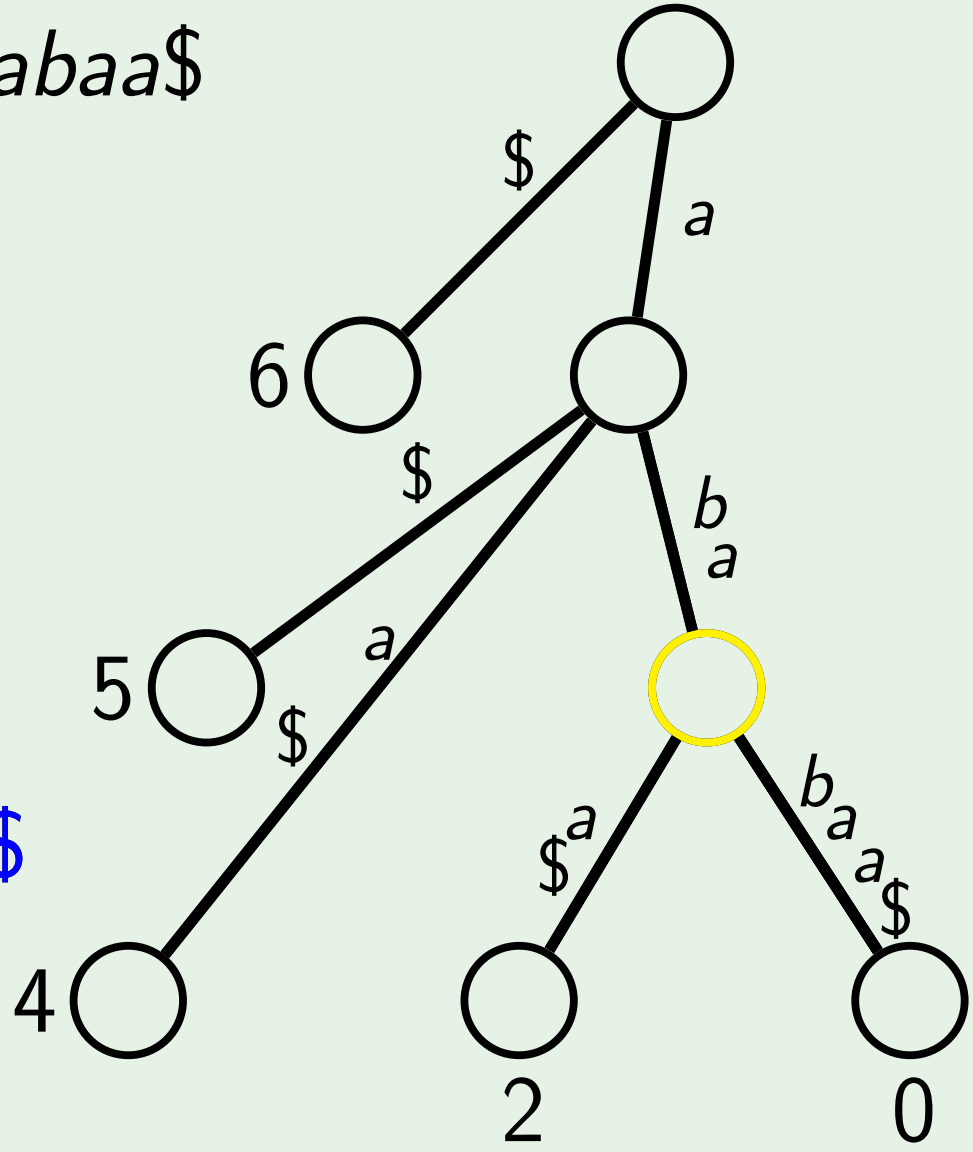
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 $a\$$

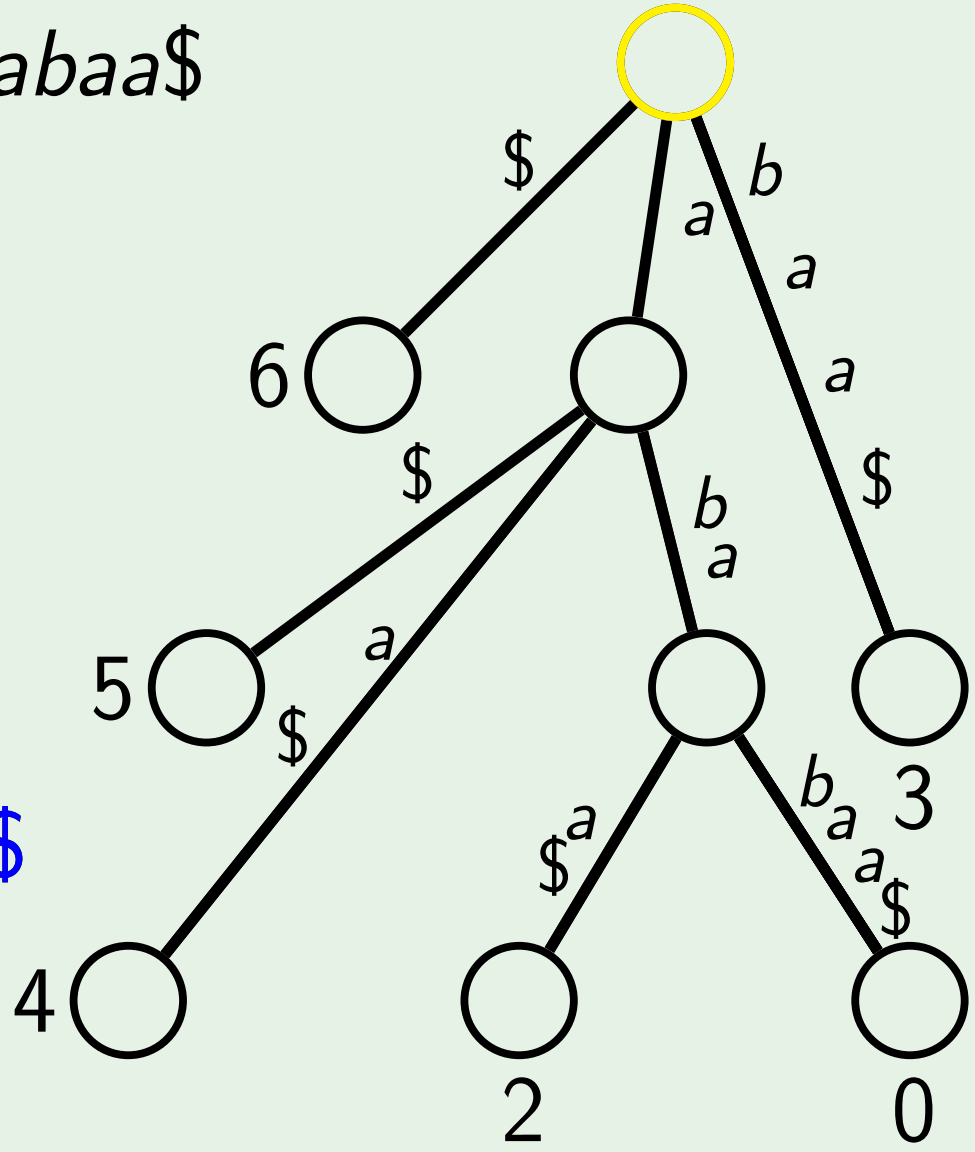
4 $aa\$$

2 $abaa\$$

0 $ababaa\$$

3 $baa\$$

1 $babaa\$$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

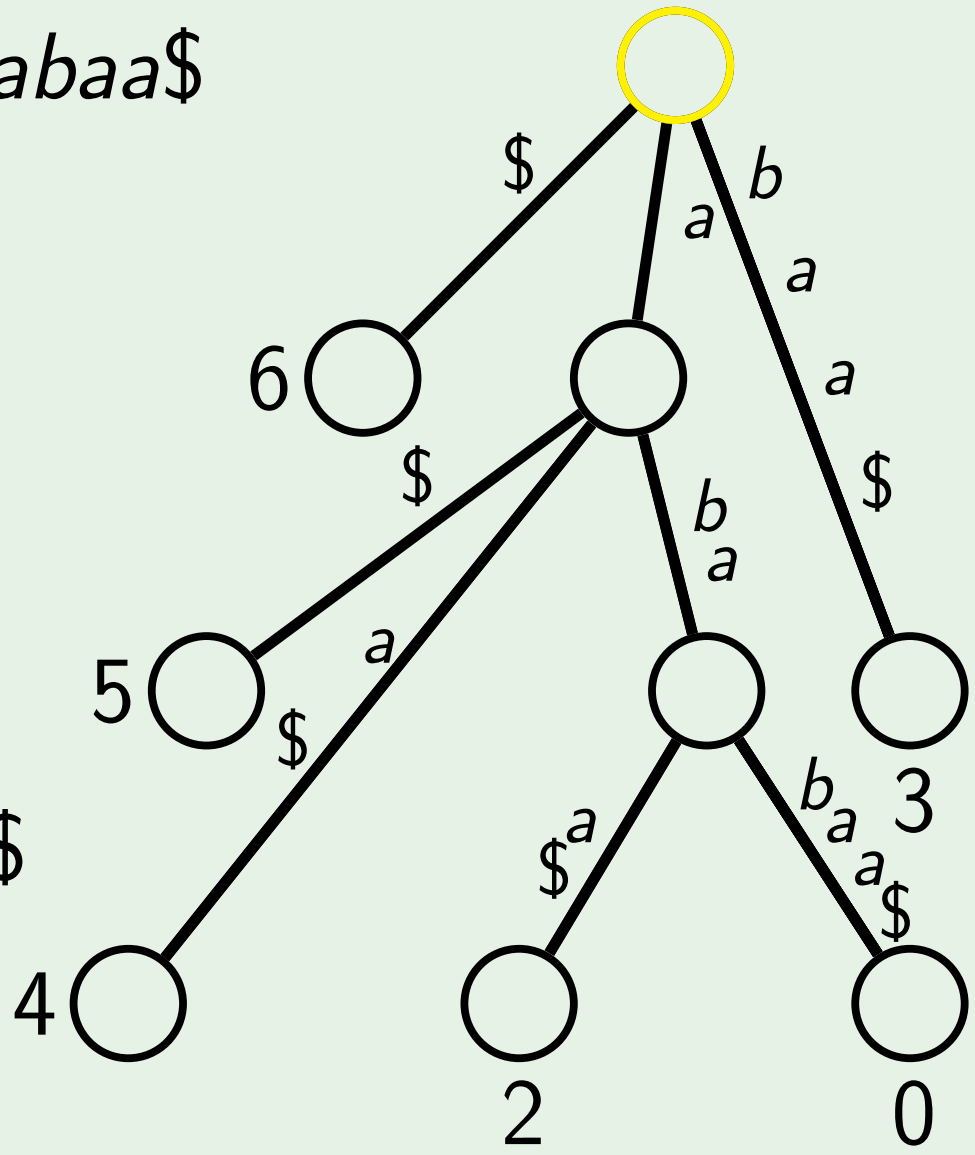
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Building suffix tree

$S = ababaa\$$

6 $

5 a$

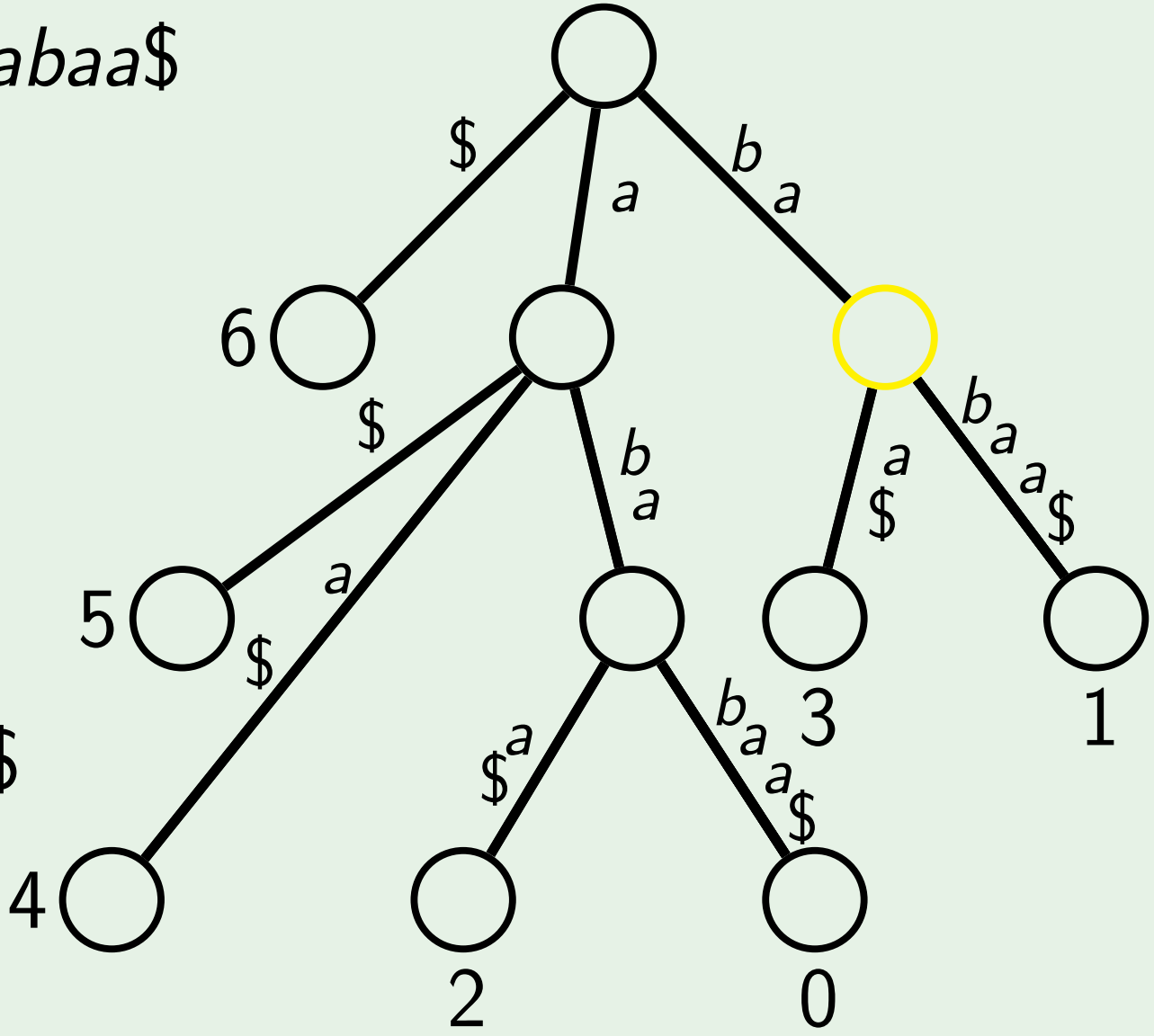
4 aa$

2 abaa$

0 ababaa$

3 baa$

1 babaa$



Algorithm

- Build suffix array and LCP array
- Start from only root vertex
- Grow first edge for the first suffix
- For each next suffix, go up from the leaf until LCP with previous is below
- Build a new edge for the new suffix

Outline

```
class SuffixTreeNode:
SuffixTreeNode parent
Map<char, SuffixTreeNode> children
integer stringDepth
integer edgeStart
integer edgeEnd
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

STFromSA(S, *order*, *lcpArray*)

```
root ← new SuffixTreeNode(
    children = {}, parent = nil, stringDepth = 0,
    edgeStart = −1, edgeEnd = −1)
lcpPrev ← 0
curNode ← root
for i from 0 to |S| − 1:
    suffix ← order[i]
    while curNode.stringDepth > lcpPrev:
        curNode ← curNode.parent
    if curNode.stringDepth == lcpPrev:
        curNode ← CreateNewLeaf(curNode, S, suffix)
    else:
        edgeStart ← order[i − 1] + curNode.stringDepth
        offset ← lcpPrev − curNode.stringDepth
        midNode ← BreakEdge(curNode, S, edgeStart, offset)
        curNode ← CreateNewLeaf(midNode, S, suffix)
    if i < |S| − 1:
        lcpPrev ← lcpArray[i]
return root
```

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$children = \{\},$
$parent = node,$
$stringDepth = |S| - suffix,$
$edgeStart = suffix + node.stringDepth,$
$edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$children = \{\}$,
$parent = node$,
$stringDepth = |S| - suffix$,
$edgeStart = suffix + node.stringDepth$,
$edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$children = \{\}$,
$parent = node$,
$stringDepth = |S| - suffix$,
$edgeStart = suffix + node.stringDepth$,
$edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

`CreateNewLeaf`($node, S, suffix$)

$leaf \leftarrow$ `new` $SuffixTreeNode($
$\quad children = \{\},$
$\quad parent = node,$
$\quad stringDepth = |S| - suffix,$
$\quad edgeStart = suffix + node.stringDepth,$
$\quad edgeEnd = |S| - 1)$
$node.children[S[leaf.edgeStart]] \leftarrow leaf$
`return` $leaf$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` *SuffixTreeNode*(
  $children = \{\}$,
  $parent = node$,
  $stringDepth = node.stringDepth + offset$,
  $edgeStart = start$,
  $edgeEnd = start + offset - 1$)
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` *midNode*

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge(*node*, *S*, *start*, *offset*)

```
startChar ← S[start]
midChar ← S[start + offset]
midNode ← new SuffixTreeNode(
    children = {},
    parent = node,
    stringDepth = node.stringDepth + offset,
    edgeStart = start,
    edgeEnd = start + offset − 1)
midNode.children[midChar] ← node.children[startChar]
node.children[startChar].parent ← midNode
node.children[startChar].edgeStart+ = offset
node.children[startChar] ← midNode
return midNode
```

BreakEdge(*node*, *S*, *start*, *offset*)

```
startChar ← S[start]
midChar ← S[start + offset]
midNode ← new SuffixTreeNode(
    children = {},
    parent = node,
    stringDepth = node.stringDepth + offset,
    edgeStart = start,
    edgeEnd = start + offset − 1)
midNode.children[midChar] ← node.children[startChar]
node.children[startChar].parent ← midNode
node.children[startChar].edgeStart+ = offset
node.children[startChar] ← midNode
return midNode
```

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge(*node*, *S*, *start*, *offset*)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` *SuffixTreeNode*(
  $children = \{\}$,
  $parent = node$,
  $stringDepth = node.stringDepth + offset$,
  $edgeStart = start$,
  $edgeEnd = start + offset - 1$)
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` *midNode*

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

BreakEdge($node, S, start, offset$)

$startChar \leftarrow S[start]$
$midChar \leftarrow S[start + offset]$
$midNode \leftarrow$ `new` $SuffixTreeNode($
  $children = \{\},$
  $parent = node,$
  $stringDepth = node.stringDepth + offset,$
  $edgeStart = start,$
  $edgeEnd = start + offset - 1)$
$midNode.children[midChar] \leftarrow node.children[startChar]$
$node.children[startChar].parent \leftarrow midNode$
$node.children[startChar].edgeStart += offset$
$node.children[startChar] \leftarrow midNode$
`return` $midNode$

Analysis

Lemma

This algorithm runs in $O(|S|)$

Proof

- Total number of edges in suffix tree is $O(|S|)$
- For each edge, we go at most once down and at most once up
- Constant time to create a new edge and possibly a new node

Conclusion

- Can build suffix tree from suffix array in linear time
- Can build suffix tree from scratch in time $O(|S| \log |S|)$