Algorithmen auf Sequenzen

Vorlesung von Prof. Dr. Sven Rahmann
im Sommersemester 2008

Kapitel 9
Suffixbäume und Suffixarrays

Webseite zur Vorlesung
http://ls11-www.cs.tu-dortmund.de/people/rahmann/teaching/ss2008/AlgorithmenAufSequenzen

Sprechstunde
Mo 16-17 in OH14, R214

Motivation

Linearzeitalgorithmen zur Suche eines Musters P in einem Text T
sind dann effizient, wenn vorher weder P noch T bekannt sind.
Für k Suchanfragen mit jeweils $|P|=m$ und $|T|=n$ braucht man $O(k(n+m))$ Zeit.

Weiß man vorher, dass man viele verschiedene Muster P, P', P'', ...
in demselben Text sucht, ist es vorteilhaft, einen Index von T zu erstellen,
so dass sich jede einzelne Suchanfrage in $O(m)$, unabhängig von $|T|$,
beantworten lässt.

Wir werden sehen, dass entsprechende Index-Datenstrukturen
in $O(n)$ Zeit erstellt werden können.
Dies führt bei k Suchanfragen auf $O(n + km)$ Zeit insgesamt.

Solche Datenstrukturen sind Suffixbäume oder (Enhanced) Suffix-Arrays.

Der Suffixbaum eines Strings

Im Folgenden sei stets $T = T_1 \ldots T_n$ ein Text
über einem geordneten Alphabet A mit $|A| = O(1)$.
Aus technischen Gründen betrachten wir im Folgenden den Suffixbaum von T\$,
wobei \$ ein String-Ende-Zeichen ist, das in A nicht vorkommt.
So ist gewährleistet, dass kein Suffix noch an anderer Stelle in T vorkommt.

Der **Suffixbaum** von T\$ ist ein gerichteter Baum mit folgenden Eigenschaften:

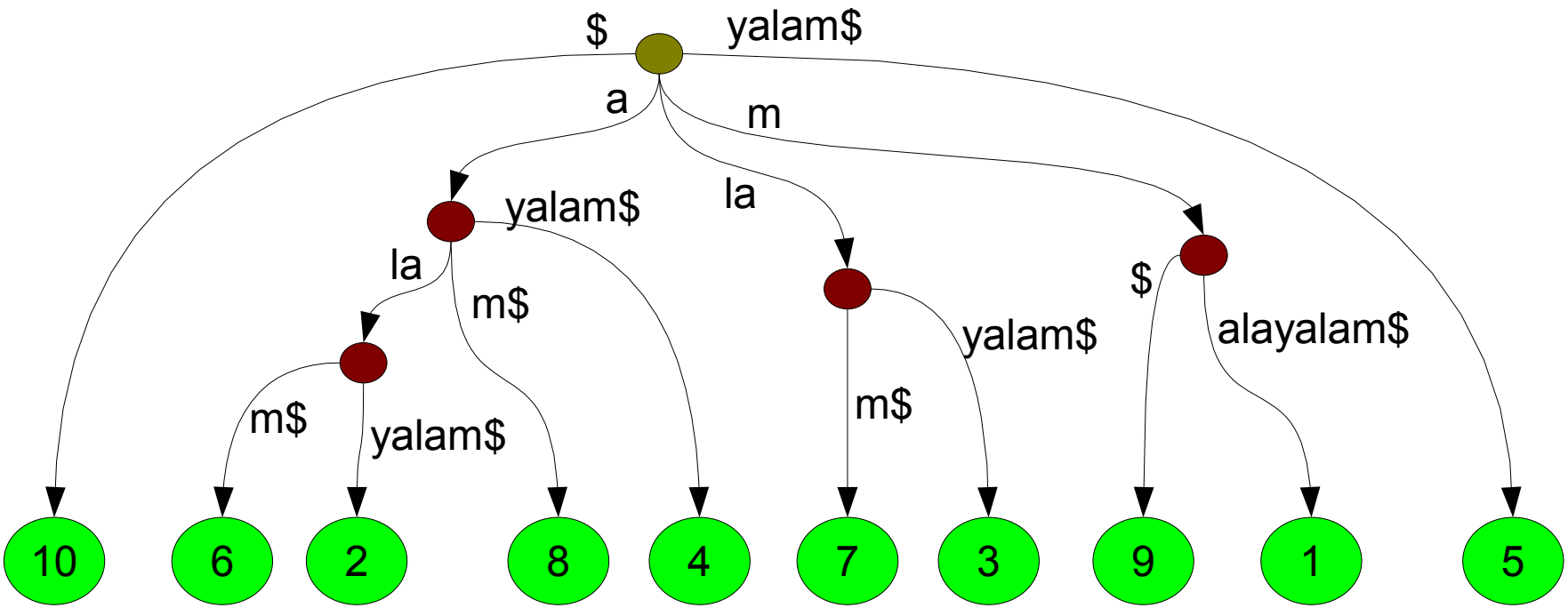
- Jedes Blatt ist bijektiv einem Suffix von T\$ zugeordnet.
- Jeder innere Knoten ist bijektiv einem **rechtsverzweigenden** Teilstring zugeordnet
  (d.h., einem Teilstring x, so dass es verschiedene Zeichen a,b gibt,
  für die sowohl xa als auch xb Teilstring von T sind).
  Die Wurzel ist dabei dem leeren String zugeordnet.
- Die ausgehenden Kanten eines inneren Knoten sind mit nichtleeren Teilstrings von T beschriftet, so dass die Teilstrings verschiedener Kanten mit verschiedenen Buchstaben beginnen.
  Die Kanten sind dabei nach der Alphabetordnung geordnet.
- Die Zuordnung eines Knoten v zu einem Teilstring bzw. Suffix ergibt sich aus der Konkatenation der Beschriftungen auf dem Pfad von der Wurzel zu v.

Beispiel: Malayalam$

`malayalam$` bezeichnet einen indischen Dialekt.
`1234567890` [Positionsangabe mod 10]

Suffixbaum

Die Wurzel hat 5 ausgehende Kanten,
deren Beschriftungen jeweils mit ($, a, l, m, y) beginnen.



Einfache Folgerungen aus der Definition

Definition des Suffixbaums

- Jedes Blatt ist bijektiv einem Suffix von *T*\$ zugeordnet.
- Jeder innere Knoten ist bijektiv einem **rechtsverzweigenden** Teilstring zugeordnet
  (d.h., einem Teilstring *x*, so dass es verschiedene Zeichen *a*,*b* gibt,
  für die sowohl *xa* als auch *xb* Teilstring von *T* sind).
  Die Wurzel ist dabei dem leeren String zugeordnet.
- Die ausgehenden Kanten eines inneren Knoten sind mit nichtleeren Teilstrings von T beschriftet, so dass die Teilstrings verschiedener Kanten mit verschiedenen Buchstaben beginnen.
- Die Zuordnung eines Knoten *v* zu einem Teilstring bzw. Suffix ergibt sich aus der Konkatenation der Beschriftungen auf dem Pfad von der Wurzel zu *v*.

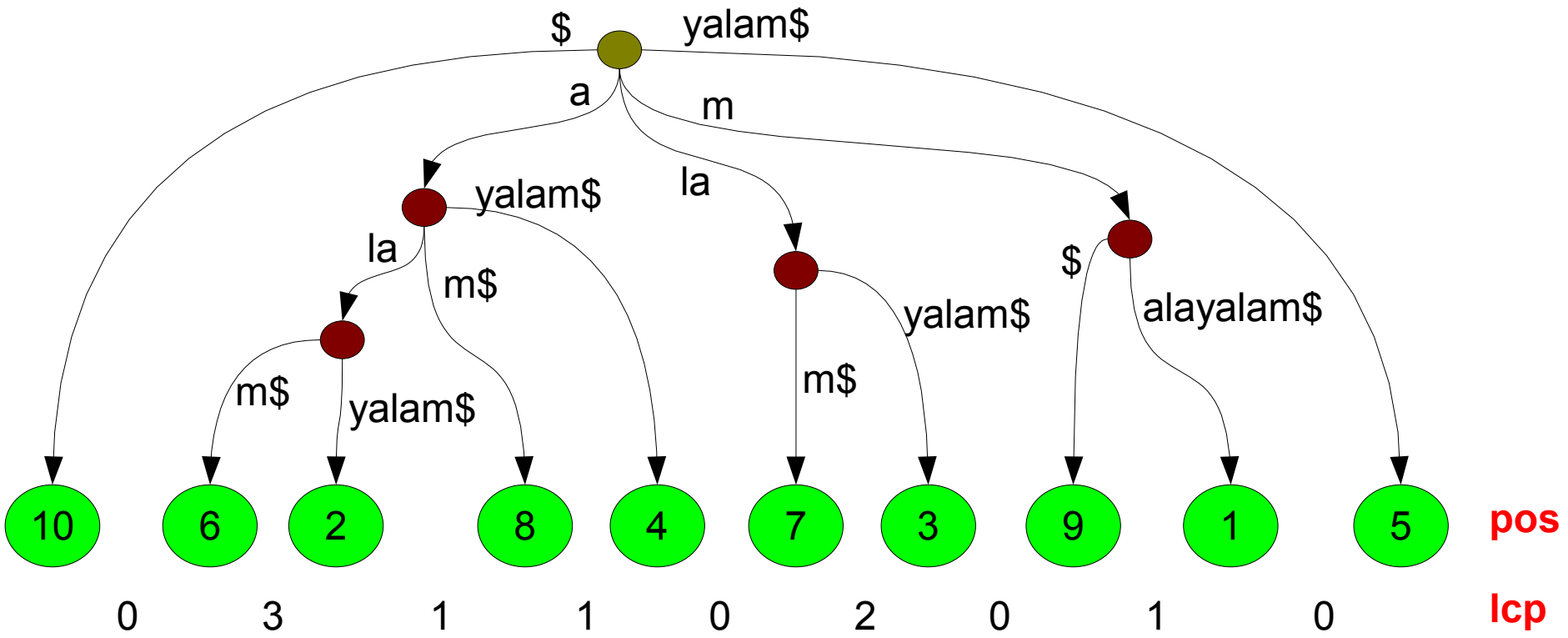
Folgerungen

- Ein innerer Knoten hat mindestens zwei ausgehende Kanten.
- Die Anzahl der inneren Knoten + Wurzel ist höchstens $|T|$.
- Ist ein innerer Knoten einem String *ax* zugeordnet (*a* ein Zeichen),
  so gibt es einen weiteren inneren Knoten, der *x* zugeordnet ist.
- Ist *x* ein Teilstring von *T*, so gibt es ein Blatt mit der Beschriftung *xy* (*y* ein String).
- Der Suffixbaum benötigt $O(|T|)$ Speicher,
  wenn man die Kanten mit Teilstring-Indizes statt den Teilstrings selbst annotiert.

(Enhanced) Suffix-Array

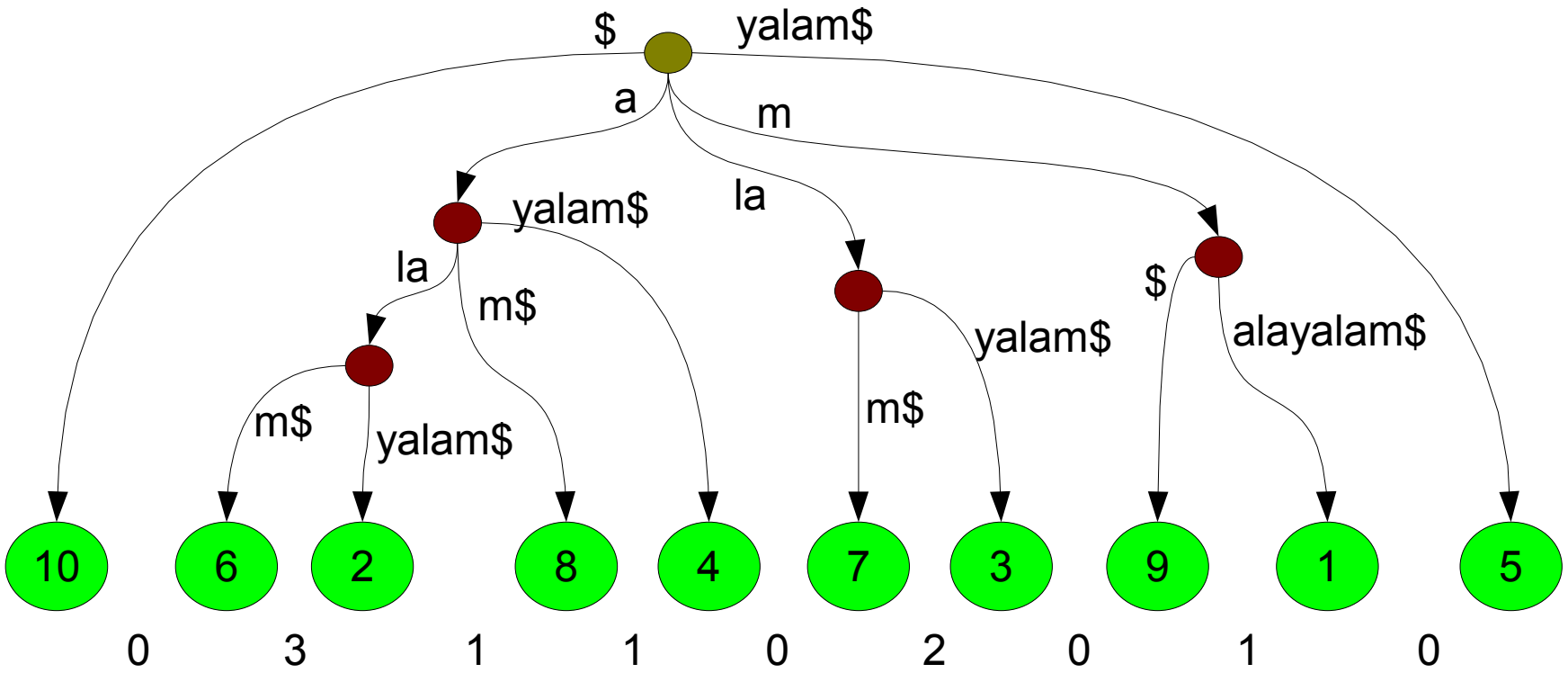
Motivation

Obwohl man den Suffixbaum in O(|*T*|) Platz speichern kann,
ist der konstante Faktor für die Baumstruktur relativ groß
Im Prinzip genügen nur zwei Arrays, um die Struktur des Baumes abzubilden:



Das lcp-Array enthält die **Stringtiefe** des tiefsten Knoten zwischen den jeweiligen benachbarten Blättern (längster gemeinsamer Präfix der zugeordneten Suffixe).

Enhanced Suffix-Array: Definitionen

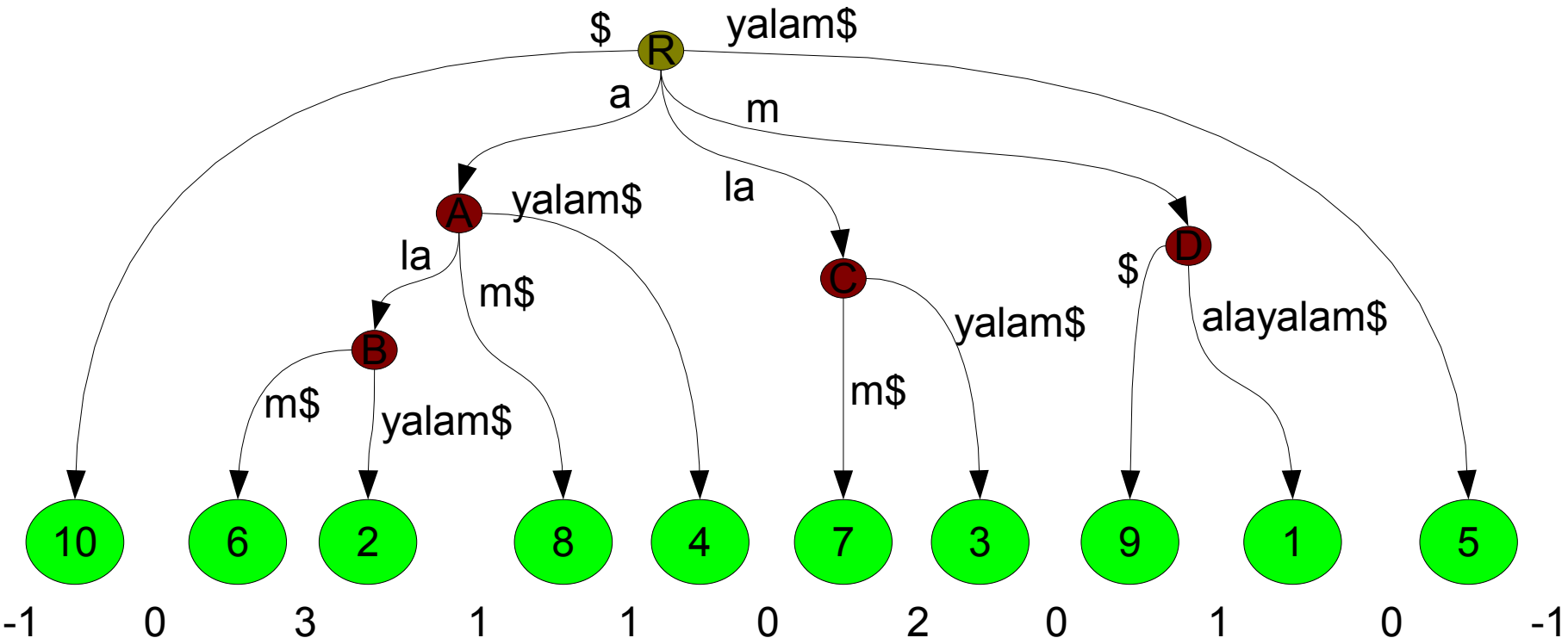


- Das **Suffix-Array pos** von T\$ ist eine Permutation von {1, ..., |T|+1};
  pos[*r*] ist die Startposition des lexikographisch *r*-t kleinsten Suffixes.

- **lcp[*r*]** ist die Länge des längsten gemeinsamen Präfix der Suffixe $T^{pos[r-1]}$ und $T^{pos[r]}$.

- **rank** ist die inverse Permutation zu pos: rank[pos[*r*]]=*r* und pos[rank[*p*]]=*p* für alle *r*, *p*.
  rank[*p*] gibt den lexikographischen Rang des Suffixes T^p an.

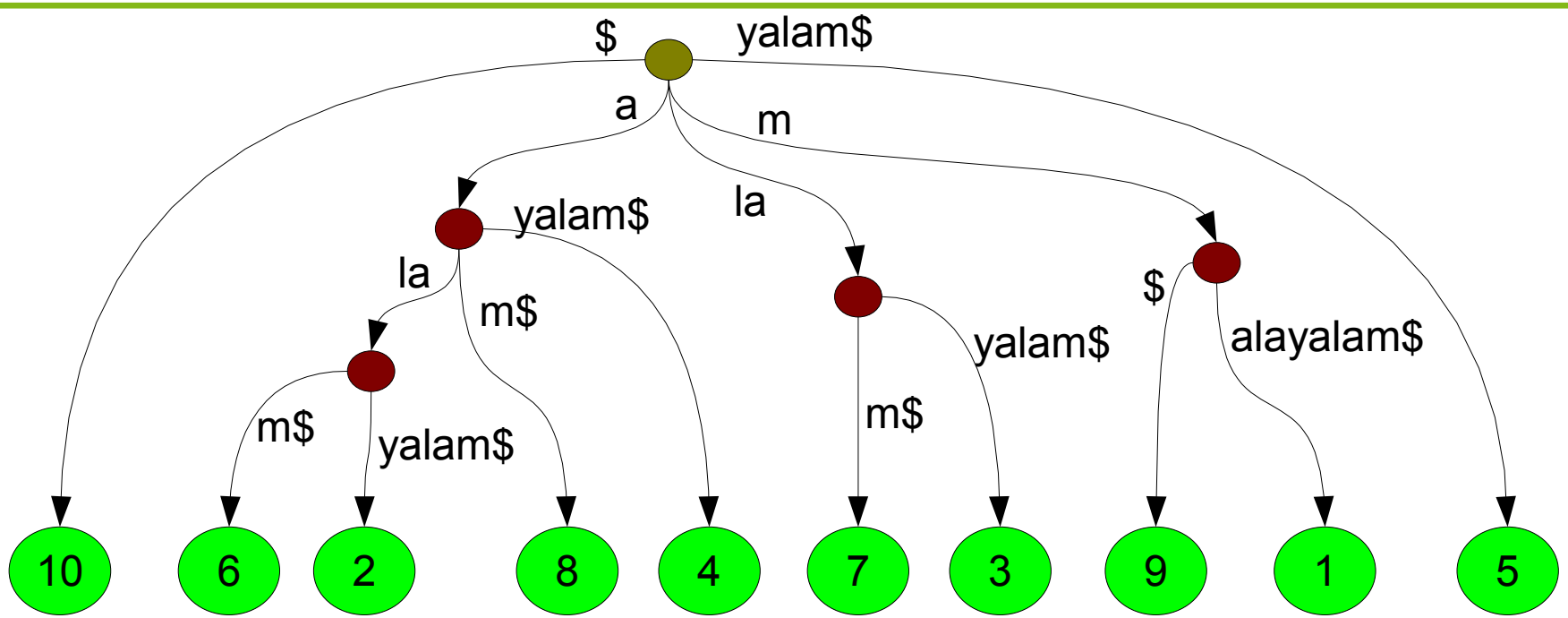
Enhanced Suffix-Array: Definitionen



- Wir sagen, [*r*, *r'*] mit *r* < *r'* ist ein ***d*-Intervall** der Länge *r'*-*r*+1, wenn
  lcp[*r*] < *d*, lcp[*r'*+1] < *d*, min{ lcp[*q*] : *r* < *q* <= *r'* } = *d*.
  Implizit setzen wir lcp[1] := -1 und lcp[|*T*|+2] := -1.
  Ein *d*-Intervall mit entspricht genau einem inneren Knoten der Stringtiefe *d*.

Intervalle im Beispiel in der Form d-[r,r']:
0-[1,10] (R); 1-[2,5] (A mit a); 3-[2,3] (B mit ala); 2-[6,7] (C mit la); 1-[8,9] (D mit m)

Einfache Mustersuche im Suffixbaum



Man kann in O(|*P*|) Zeit feststellen, ob ein String *P* Teilstring von *T* ist:

- Starte in der Wurzel
- Folge der Kante, deren Beschriftung mit dem ersten Buchstaben von *P* beginnt.
- Prüfe die Zeichen entlang dieser Kante auf Übereinstimmung mit *P*.
- In einem Knoten, folge der nächsten *P* weiter buchstabierenden Kante...
- ... bis entweder keine weiteren Zeichen übereinstimmen oder *P* gefunden ist.

Die Positionen von P stehen in dem Unterbaum,
dessen Wurzel der nächsttiefere (oder aktuelle) innere Knoten ist.

Beispiele: yal, la, alma, ala

Einfache Mustersuche im Suffixbaum

Algorithmus

- Starte in der Wurzel
- Folge der Kante, deren Beschriftung mit dem ersten Buchstaben von *P* beginnt.
- Prüfe die Zeichen entlang dieser Kante auf Übereinstimmung mit *P*.
- In einem Knoten, folge wieder der entsprechenden Kante...
- ... bis entweder keine weiteren Zeichen übereinstimmen oder *P* gefunden ist.

Laufzeitanalyse

O(|*P*|), wenn in einem Knoten die richtige Kante in O(1) Zeit gefunden wird.
Durch die Voraussetzung |*A*| = O(1) ist das in jedem Fall so.

Sonst je nach Datenstruktur innerhalb der Knoten:

- Verkettete Liste: O(|*A*|) Zeit, O(|Kanten|) Platz pro Knoten
- direktes Array: O(1) Zeit, O(|*A*|) Platz pro Knoten
- sortiertes Array (statisch) oder balancierter Baum (dynamisch):
  O(log |*A*|) Zeit, O(|Kanten|) Platz pro Knoten

Einfache Mustersuche im Suffix-Array

p	1	2	3	4	5	6	7	8	9	0
T	m	a	l	a	y	a	l	a	m	$

r	1	2	3	4	5	6	7	8	9	10
pos	10	6	2	8	4	7	3	9	1	5
lcp	-1	0	3	1	1	0	2	0	1	0

Gesucht ist das *d*-Intervall [*r*, *r'*] mit *d* >= |*P*|,
so dass *P* gemeinsames Präfix von allen Suffixen $T^{pos[q]}$ mit *r* <= *q* <= *r'*
und nicht Präfix von Suffixen außerhalb dieses Intervalls ist.

Erster Algorithmus

2x binäre Suche auf pos. Suche nach:

- *r* := min {*q*: die ersten |*P*| Zeichen von $T^{pos[q]}$ sind lexikographisch >= *P*}
- *r'* := max{*q*: die ersten |*P*| Zeichen von $T^{pos[q]}$ sind lexikographisch <= *P*}

Wenn *r* > *r'*, dann kommt *P* in *T* nicht vor.
Wenn *r* = *r'*, dann kommt *P* genau 1x in *T* vor, an Position pos[*r*].
Sonst ist [*r*, *r'*] das gesuchte Intervall.

Laufzeitanalyse

O(log |*T*|) Schritte, in jedem Schritt ein Stringvergleich mit O(|*P*|) Zeit.
Also O(|*P*| log |*T*|) Zeit insgesamt.
Aber: lcp-Informationen werden nicht genutzt.

Einfache Mustersuche im Suffix-Array

p	1	2	3	4	5	6	7	8	9	0
T	m	a	l	a	y	a	l	a	m	$

r	1	2	3	4	5	6	7	8	9	10
pos	10	6	2	8	4	7	3	9	1	5
lcp	-1	0	3	1	1	0	2	0	1	0

Erster Algorithmus

2x binäre Suche auf pos nach *r*, *r'*

Zweiter Algorithmus: Verbesserung durch gemeinsame Präfixe

Während der Suche nach r verfügt man stets über ein Intervall [*L*,*R*] mit *r* in [*L*,*R*].
Wiederholte Operation: *M* := (*L*+*R*)/2; vergleiche $T^{pos[M]}$ mit *P*.
Verbesserung: Speichere die gemeinsamen Präfixlängen von ($T^{pos[L]}$, *P*) und ($T^{pos[R]}$, *P*).
Bereits als gemeinsam bekannte Präfixe müssen nicht mehr verglichen werden.

Der Suffix-Array-Artikel von Manber und Myers (1990) behauptet:
Man braucht zusätzlich von allen Tripeln (*L*,*M*,*R*), die bei der Suche auftreten können, die gemeinsamen Präfixlängen von (*L*,*M*) und (*M*,*R*).
Es gibt nur linear viele solcher Tripel; man kann alle vorberechnen.
Damit kann die Suchzeit auf O(log |*T*| + |*P*|) reduziert werden.
Aber ist diese Vorberechnung wirklich nötig; reicht obiges nicht aus?

Suffixbaum vs. Suffix-Array

Äquivalenz

- Aus dem Suffixbaum lassen sich durch eine Traversieung pos und lcp in Linearzeit berechnen.
- Aus pos und lcp lässt sich ebenso der Baum bottom-up in Linearzeit rekonstruieren: Verwende einen Stack, der zu jedem Zeitpunkt die „offenen“ *d*-Intervalle enthält. [Empfohlene Übung!]

Unterschiede

- Der Suffixbaum erlaubt eine Top-Down-Traversierung, daher Laufzeit O(|*P*|) für die Suche nach *P*.
- Mit pos und lcp kann eine Bottom-Up-Traversierung simuliert werden, aber ohne weitere Informationen keine Top-Down-Traversierung.
- Man kann aber Informationen über den durch die *d*-Intervalle gegebenen Baum in weiteren Tabellen speichern, so dass Top-Down möglich wird.
- Vorteil der Arrays: Man kann eine Vielzahl von Tabellen vorberechnen, persistent Speichern, und je nach Problemstellung nur die benötigten benutzen.
- Enhanced Suffix Array: pos + lcp + weitere Tabellen

Konstruktion von Suffixbäumen

Einfacher quadratischer Algorithmus

Füge jedes Suffix nacheinander in den Baum ein.
Reihenfolge der Suffixe spielt keine Rolle.

- Solange ein Präfix des aktuellen Suffix bereits als Pfad von der Wurzel aus existiert, verfahre wie bei der Mustersuche nach diesem Präfix.
- Sobald das nächste Zeichen nicht mehr gefunden wird, erweitere den Baum: ggf. muss auch ein innerer Knoten an der aktuellen Position eingefügt werden, in jedem Fall ein Blatt und eine Kante.
- Tafelbeispiel: `malayalam$`

Beobachtungen

- Problematisch ist das Suchen der Stelle, an der der Baum erweitert werden muss: Das Verfolgen des Pfades von der Wurzel zur Einfügestelle dauert $O(|T|)$ pro Position.
- Das Einfügen kann in O(1) Zeit durchgeführt werden, da an die neue Kante nur die Indizes des verbleibenden Teilstrings geschrieben werden.
- Ein Linearzeit-Algorithmus ist machbar, wenn man die Einfügestelle in O(1) findet.
- Dabei helfen ein paar Tricks.

Ukkonen's on-line Linearzeit-Algorithmus

Trick 0: Einfügereihenfolge: On-line Algorithmus
- Suffixe von links nach rechts einfügen (längstes zuerst)
- Suffixe sind nur bis zum aktuell bearbeiteten Zeichen bekannt
- String kann im Laufe des Algorithmus verlängert werden (on-line)
- Da die Stringlänge noch nicht bekannt sein muss,
  wird die Position des Stringendes symbolisch durch E (Ende) markiert.

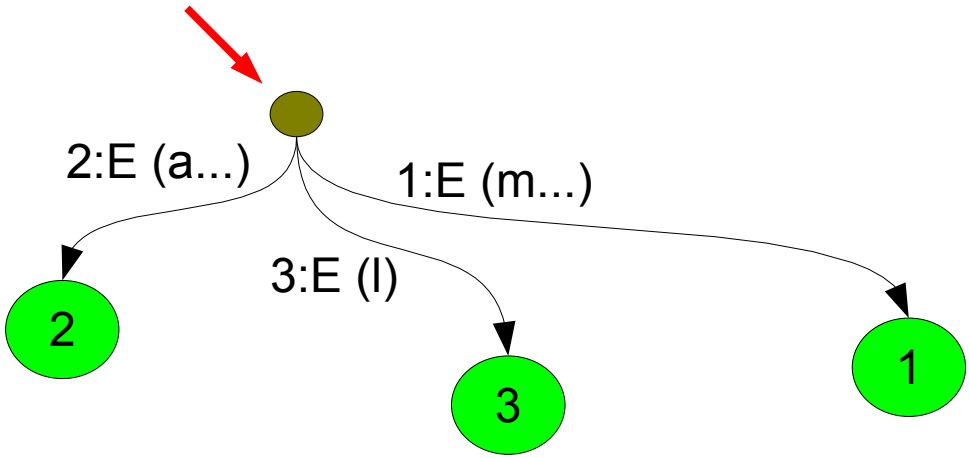
Trick 1: Suffixlinks
- Gibt es einen Knoten *v*, der zu *ax* gehört, gibt es auch einen Knoten *w* zu *x*.
  Eine solche Verbindung heißt **Suffixlink**.
  Wir speichern zu jedem Knoten *v* den Suffixlink suf(*v*).

Trick 2: Verfolgen und Anlegen von Suffixlinks
- Da man weiß, dass das Suffix *x* zu *ax* existiert, muss man auf dem Weg dahin
  keine Zeichen entlang Kanten vergleichen, sondern kann die Zeichen abzählen.

Beispiel

```
p  1234567890
T  malayalam$
```

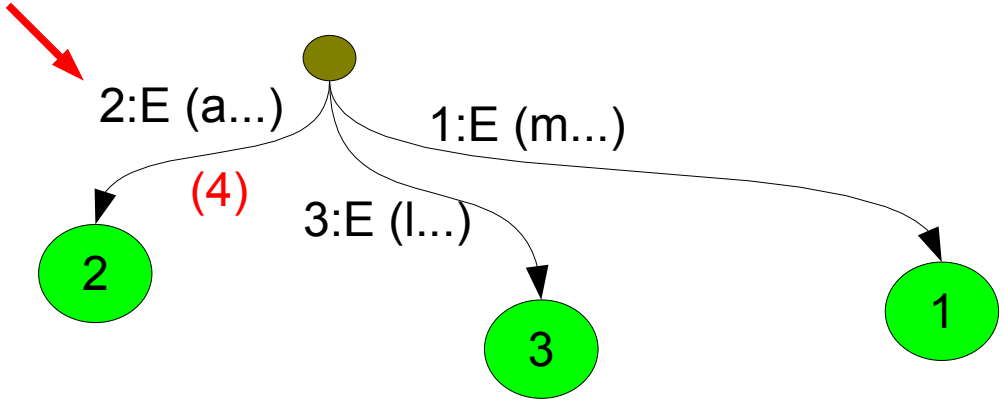
Suffixe 1,2,3: jeweils neuer Startbuchstabe



Beispiel

```
p   1234567890
T   malayalam$
```

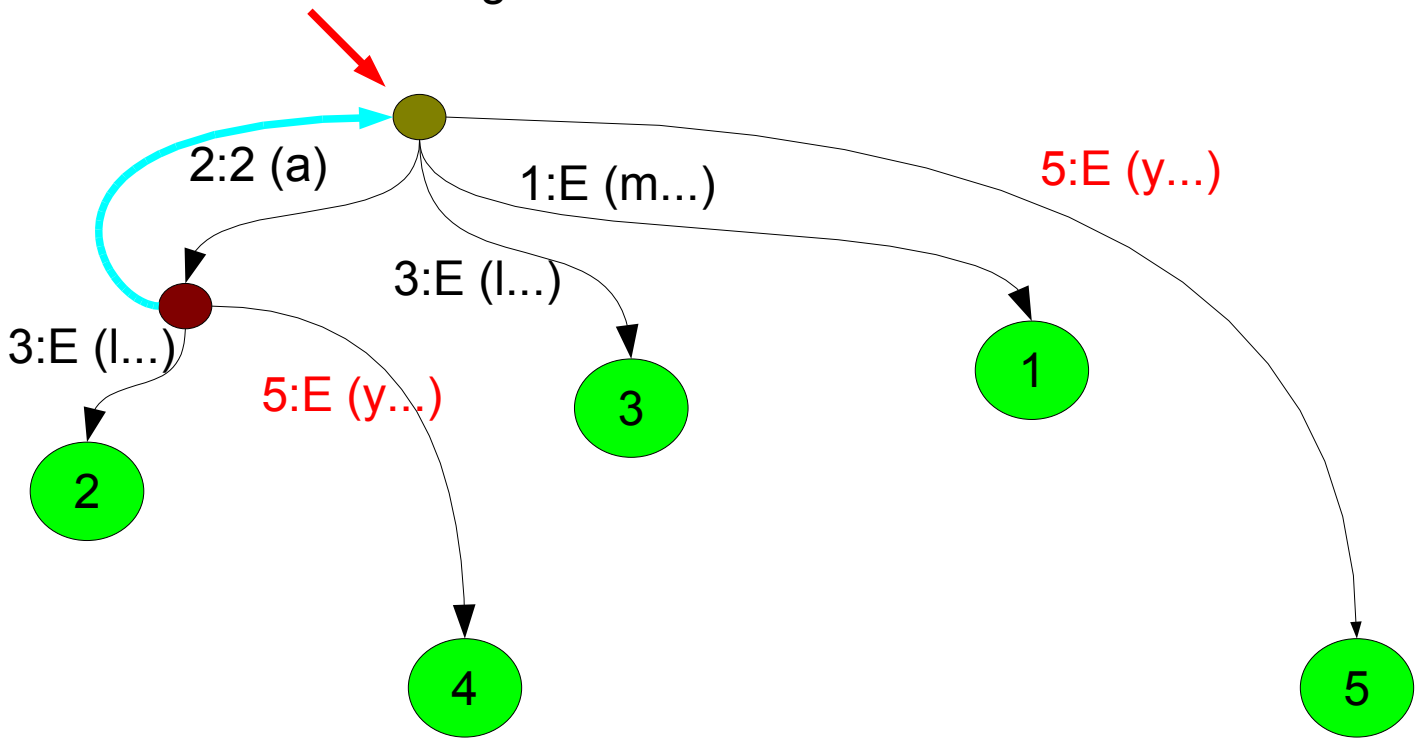
Suffix 4: a
Suche von der Wurzel aus findet „a“!
Warte ab, was bei 5 passiert (wenn l kommt, weiter)



Beispiel

```
p   1234567890
T   malayalam$
```

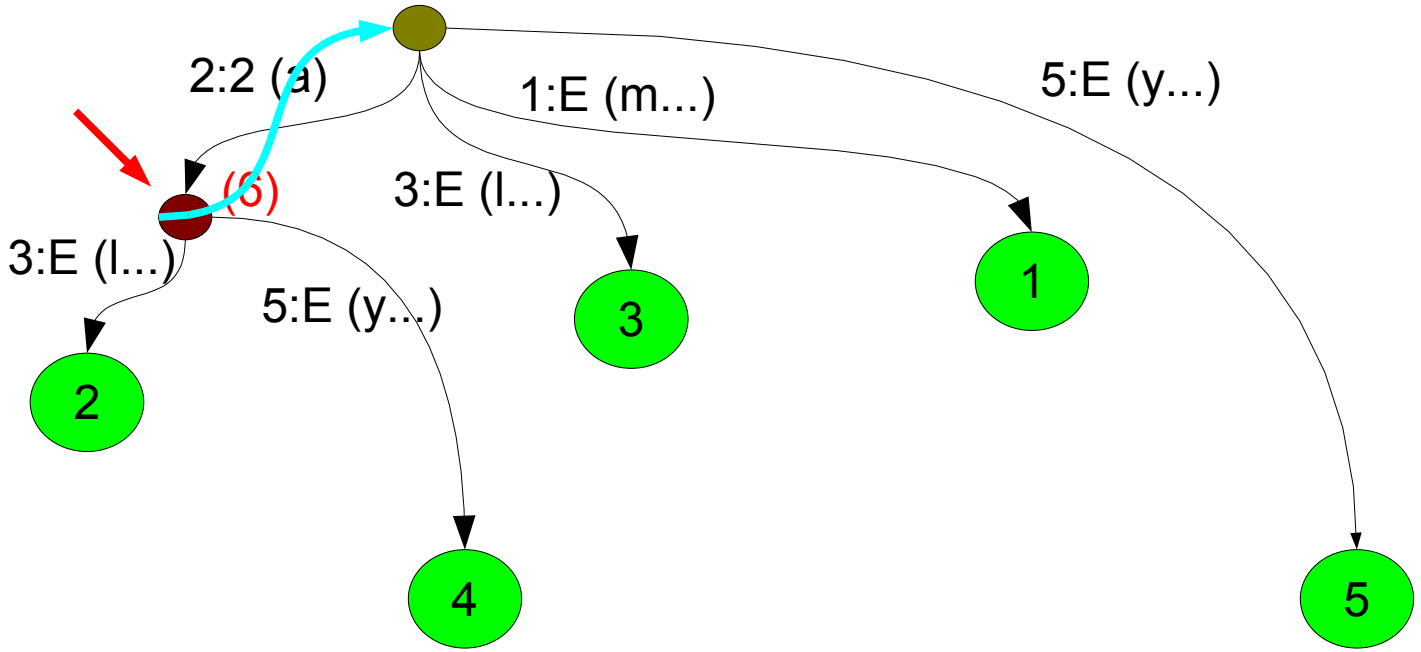
Suffix 5: y passt nicht zur aktuellen Kante.
neuer innerer Knoten; neue Kante mit y zu 4.
Suffixlink geht zur Wurzel. Von dort aus Kante zu 5.



Beispiel

```
p   1234567890
T   malayalam$
```

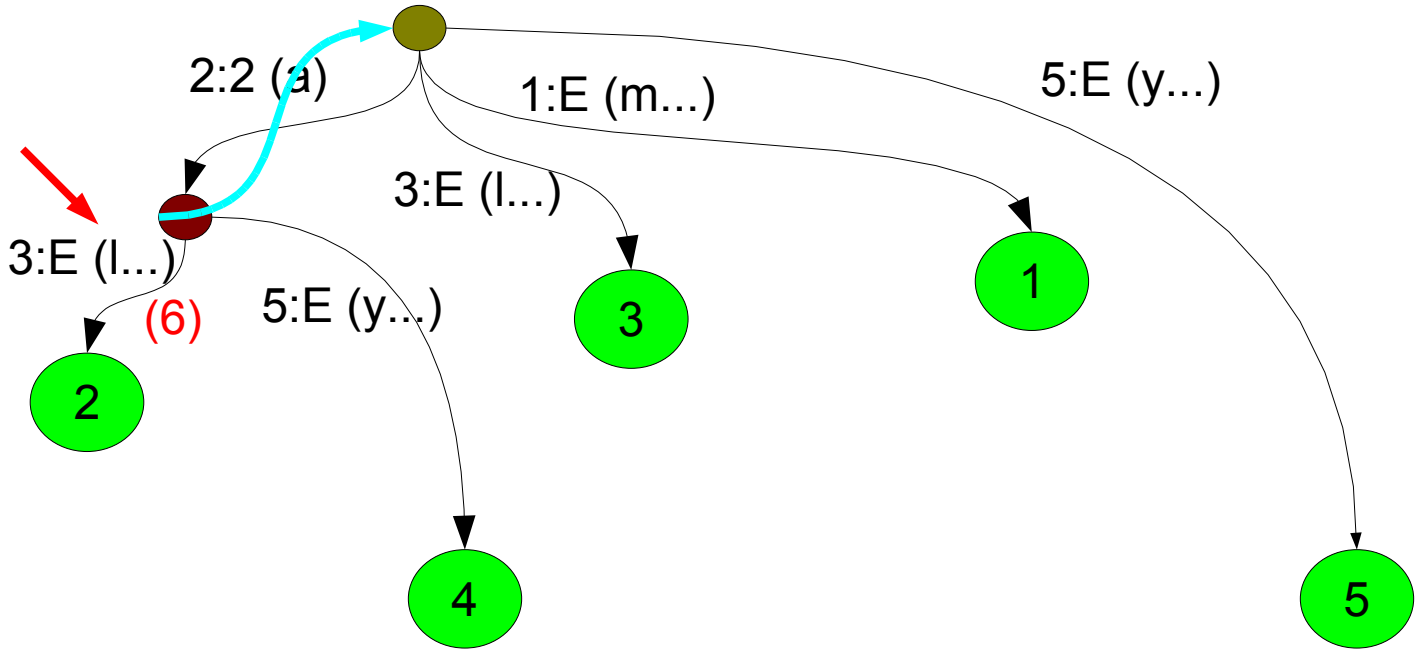
Suffix 6: a



Beispiel

```
p   1234567890
T   malayalam$
```

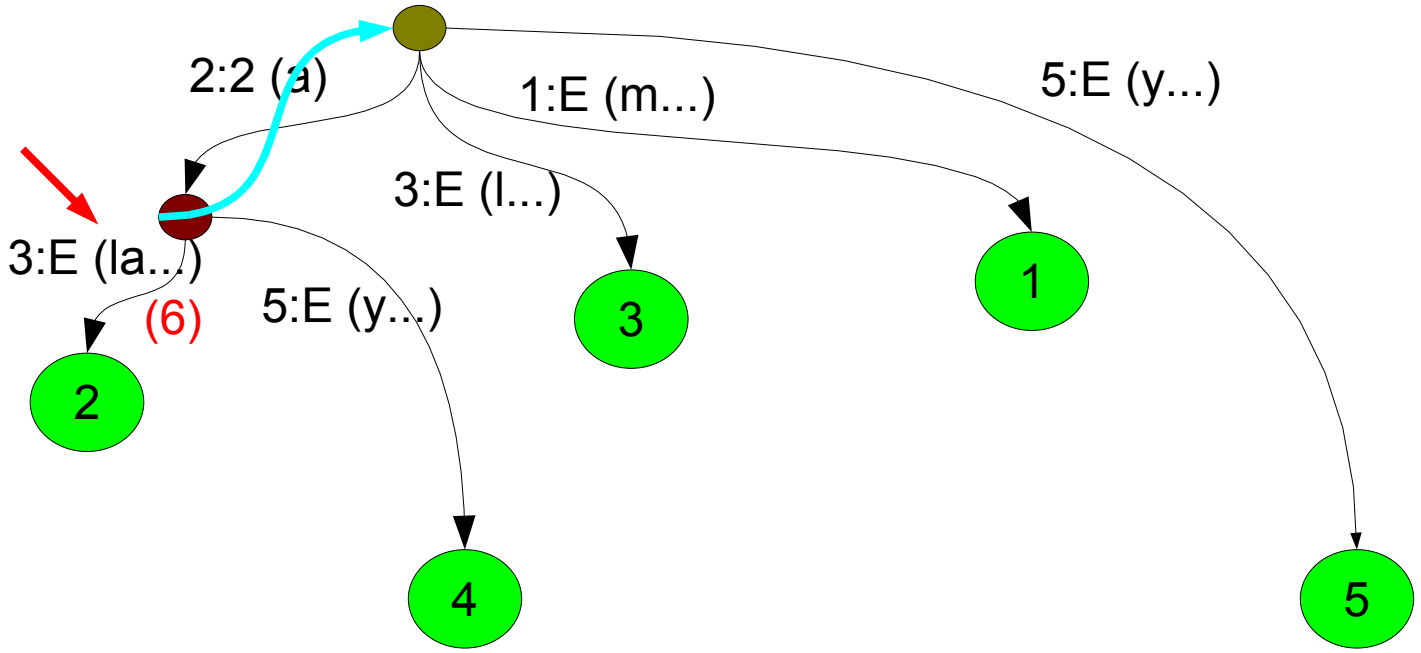
Suffixe 6,7: al, l



Beispiel

```
p   1234567890
T   malayalam$
```

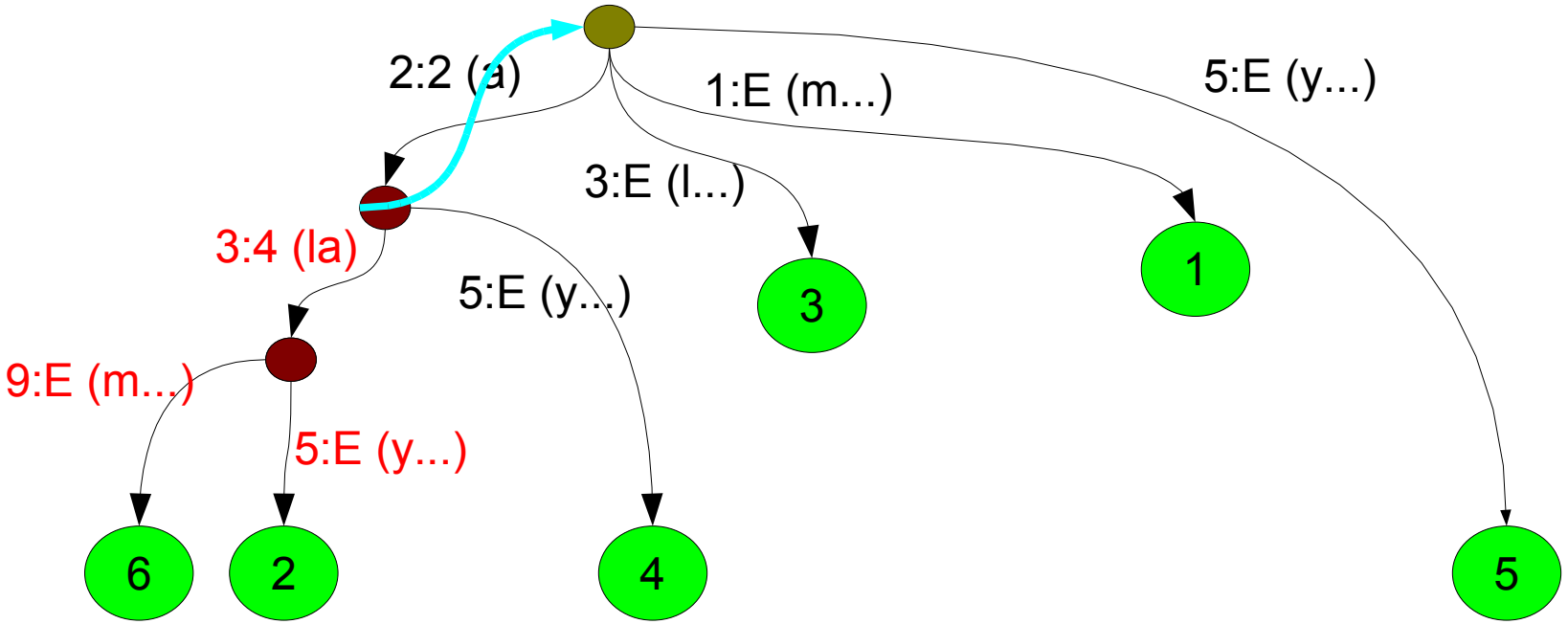
Suffixe 6,7,8: ala, la, a



Beispiel

```
p   1234567890
T   malayalam$
```

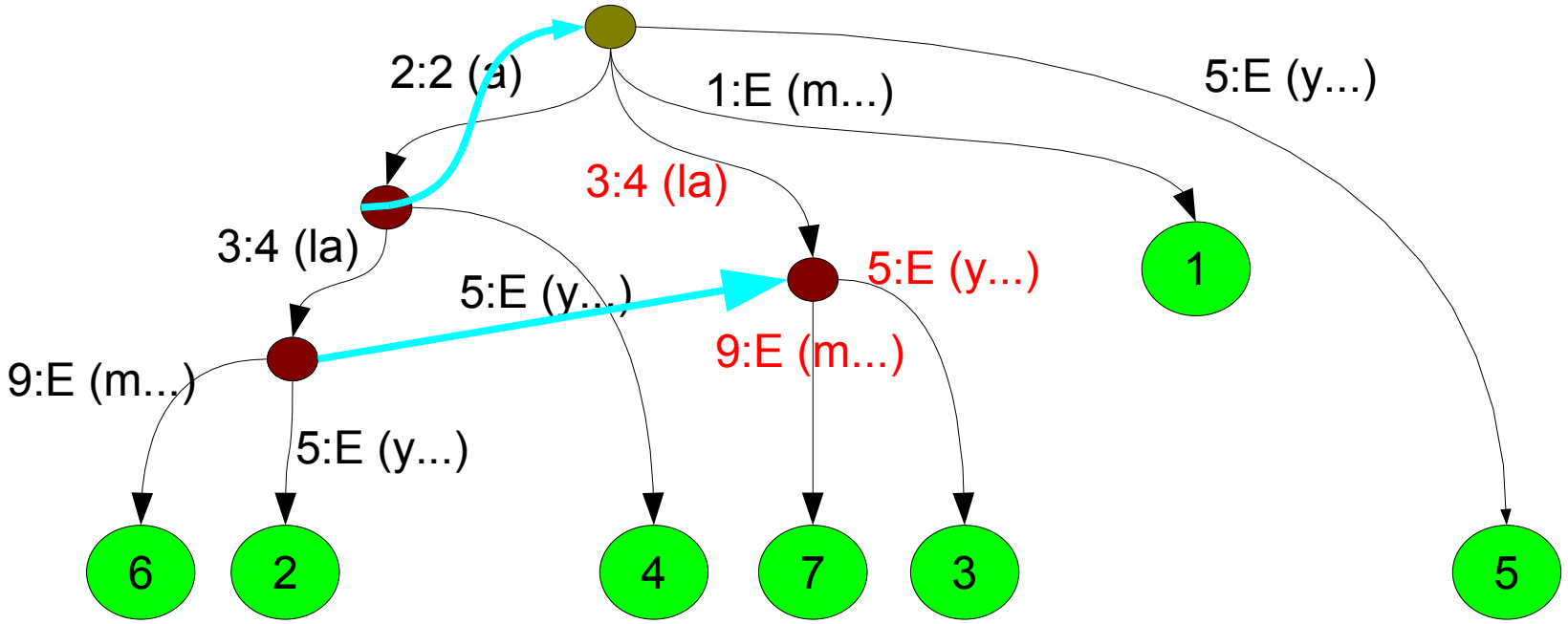
Suffixe 6,7,8,9: alam, lam, am, m
neuer inner Knoten („ala“), neue Kante davon („m“)
Es muss einen Knoten „la“ geben.
Ort effizient finden; Suffixlink setzen



Beispiel

```
p   1234567890
T   malayalam$
```

Suffixe 6,7,8,9: alam, lam, am, m
neuer inner Knoten („la“), neue Kante davon („m“)
Es muss einen Knoten „a“ geben (gibt's schon);
Suffixlink setzen

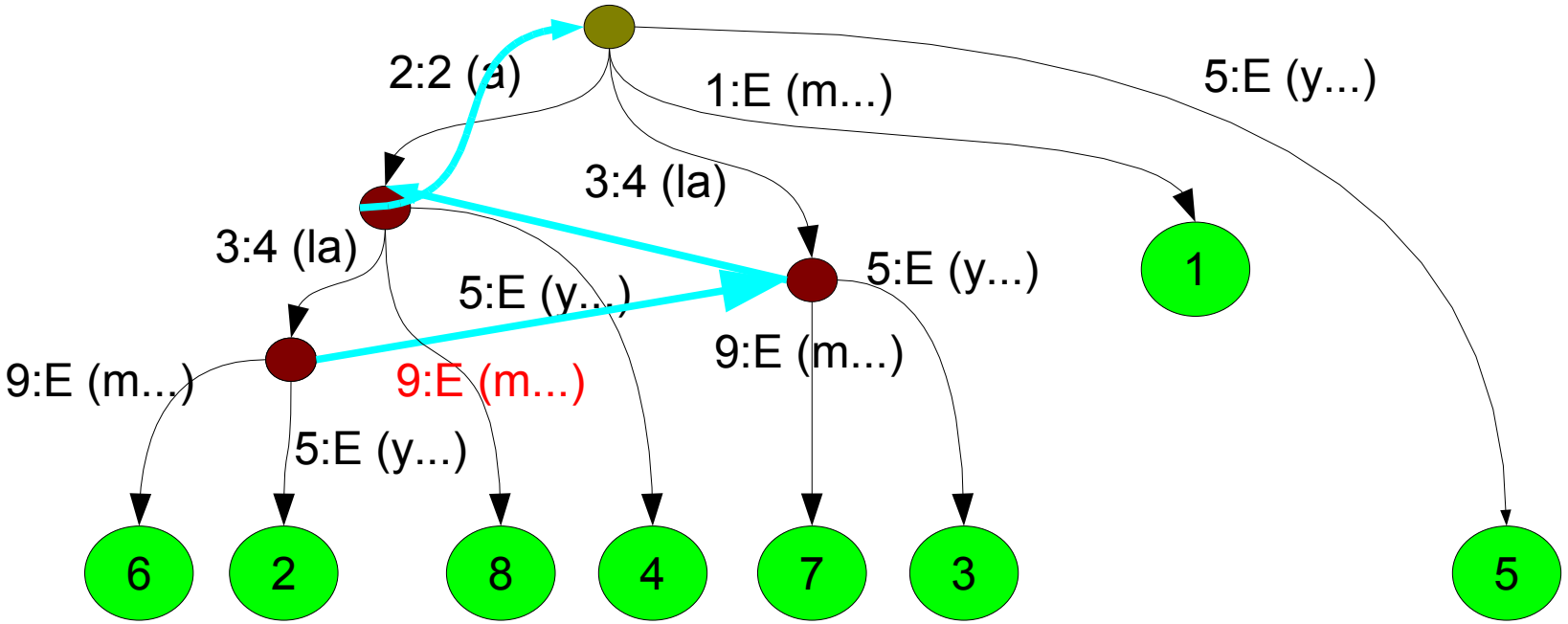


Wichtig hier: Nicht naiv den Pfad „la“ absuchen,
sondern existierende Eltern-Suffixlinks (hier: „a“ zur Wurzel) benutzen.
Man weiß, dass man von dort wieder 2 Zeichen nach unten gehen muss.

Beispiel

```
p   1234567890
T   malayalam$
```

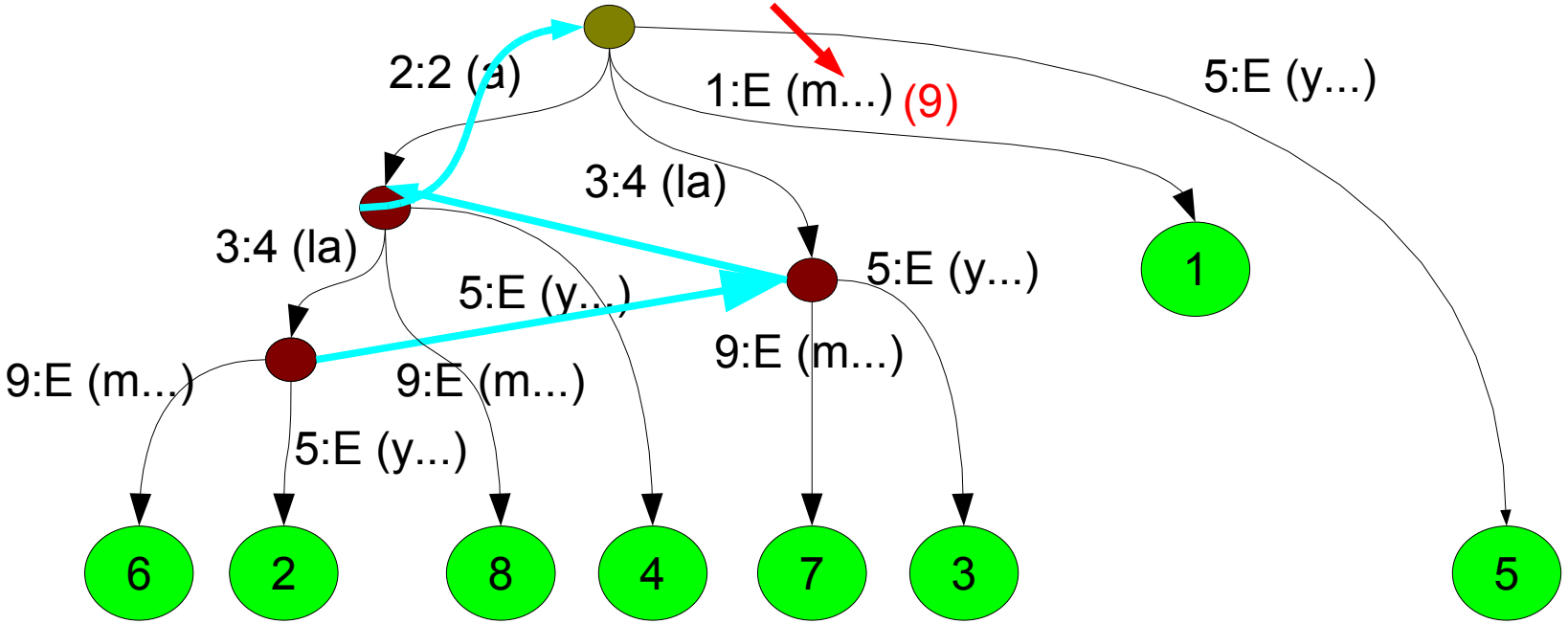
Suffixe 6,7,8,9: alam, lam, am, m
Von Knoten („a“) neue Kante („m“)
Suffixlink existiert bereits.



Beispiel

```
p   1234567890
T   malayalam$
```

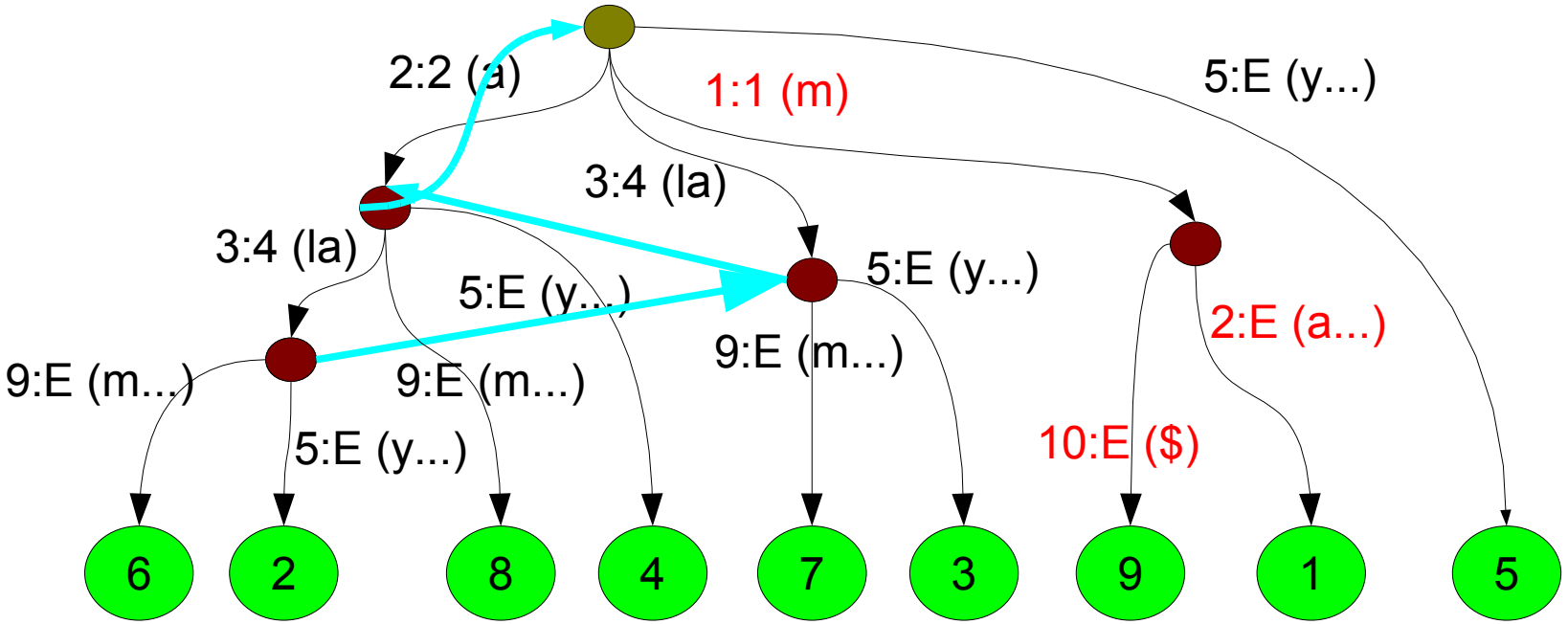
Suffixe 6,7,8,9: alam, lam, am, m
Von Knoten („a“) neue Kante („m“)
Suffixlink existiert bereits.



Beispiel

p 1234567890
T **malayalam$**

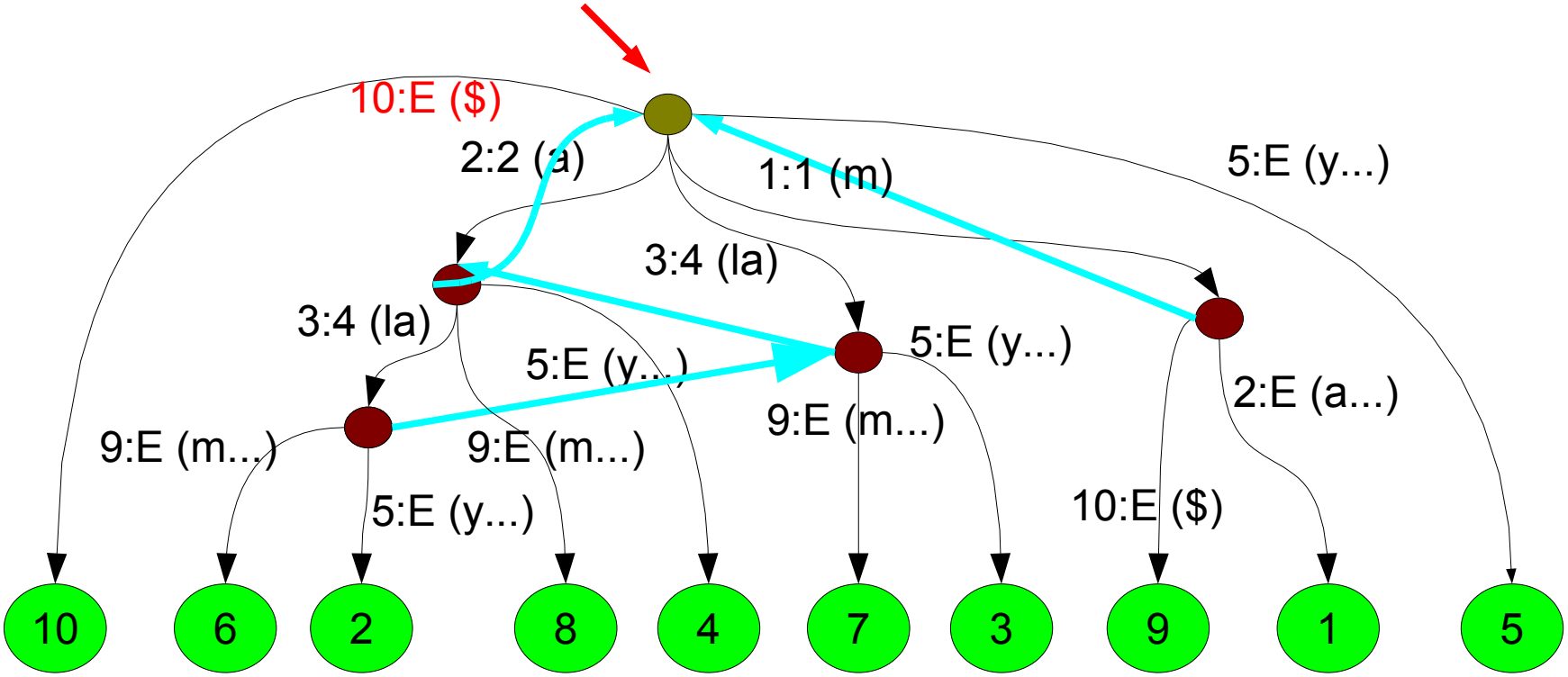
Suffixe 9, 10: m$, $
Von Knoten („m“) neue Kante („$“)
Suffixlink zur Wurzel



Beispiel

```
p  1234567890
T  malayalam$
```

Suffixe 9, 10: m$, $
Von der Wurzel neue Kante („$“)



Zum Abschluss einmal den Baum traversieren und alle E's durch 10 ersetzen.
Insgesamt amortisiert konstante Zeit pro Zeichen.

Diskussion

Linearzeit-Konstruktion des Baumes basiert essentiell auf Suffix-Links.

Suffix-Array und lcp können aus dem Baum leicht in Linearzeit gewonnen werden.
Das Ziel bei Arrays ist aber, weniger Speicher zu brauchen.
Wenn man doch erst den Baum erstellen muss, hat man wenig gewonnen.
Also: Existiert eine direkte Konstruktion für pos, lcp, ohne Suffixbaum?
(Ja, sogar auch in Linearzeit – ohne Suffixlinks!)

LCP-Berechnung in Linearzeit

Man kann aus pos das lcp-Array in Linearzeit berechnen.
(Der offensichtliche Weg nach der Definition hätte quadratische Laufzeit.)

Die Grundidee ist, lcp[r] nicht aufsteigend nach r,
sondern nach Positionen zu berechnen.
Wir nutzen aus:

lcp[rank[*p*]] >= lcp[rank[*p*-1]] – 1 (Suffixregel)

lcp-Berechnung

h := 0
p := 1
Solange *p* < |*T*|
  prev := pos[rank[*p*-1]] #Randfälle elegant abfangen
  h := *h* + gemeinsame Präfixlänge von T^{prev+h}, T^{p+h}
  lcp[rank[*p*]] := *h*
  if (*h*>0) *h*--
  p++