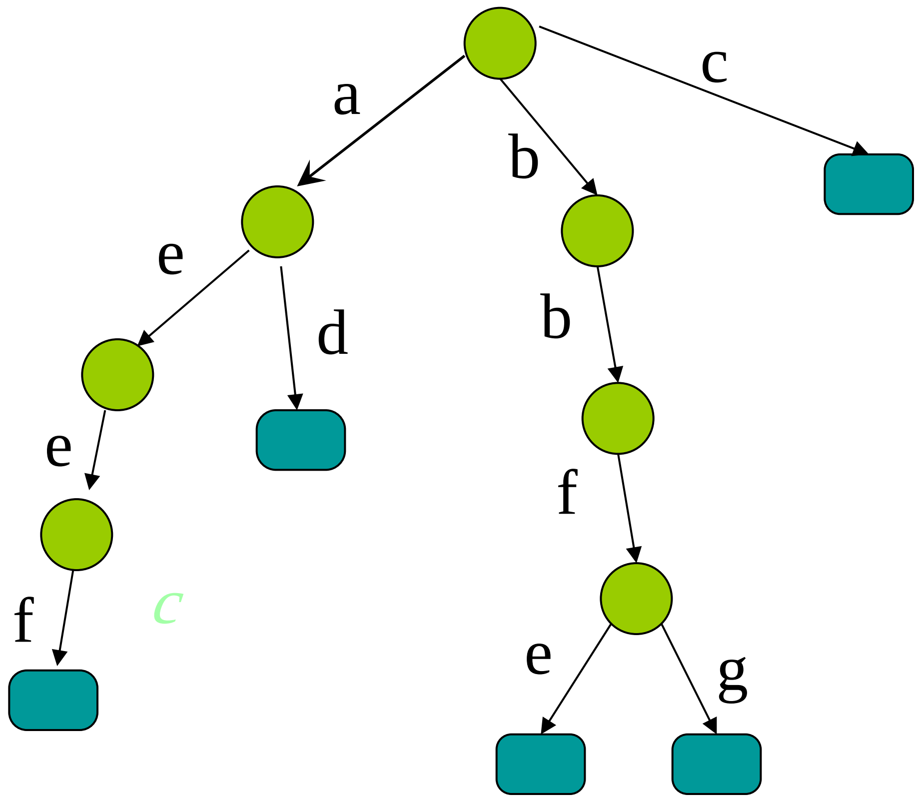
Suffix trees

Trie

- A tree representing a set of strings.

{
aeef
ad
bbfe
bbfg
c
}

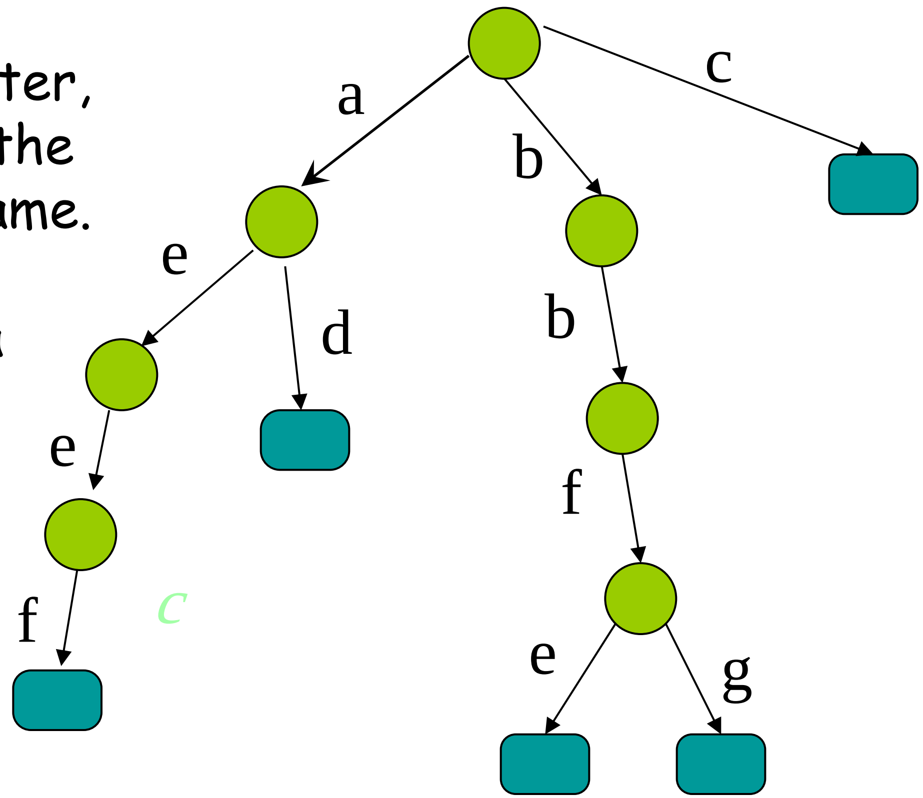


Trie (Cont)

- Assume no string is a prefix of another

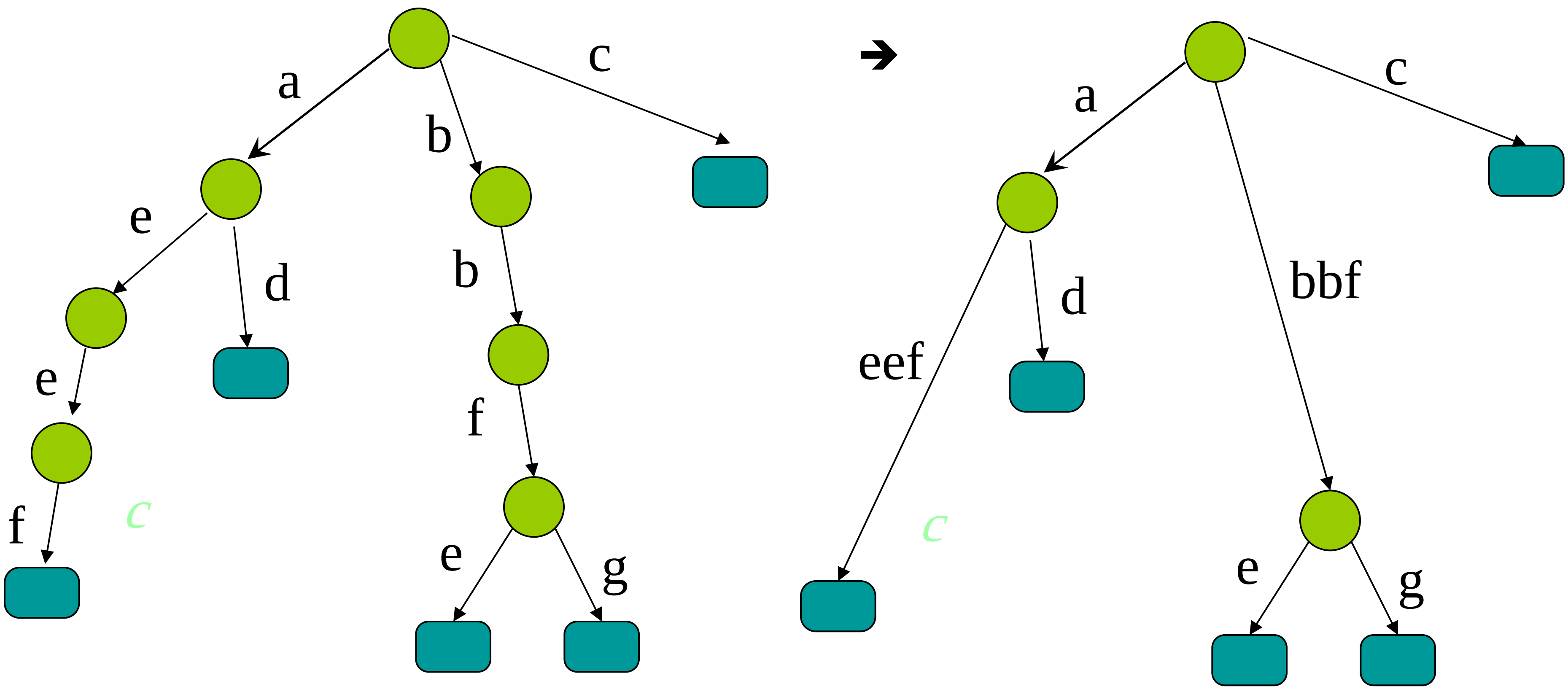
Each edge is labeled by a letter, no two edges outgoing from the same node are labeled the same.

Each string corresponds to a leaf.



Compressed Trie

- Compress unary nodes, label edges by strings



Suffix tree

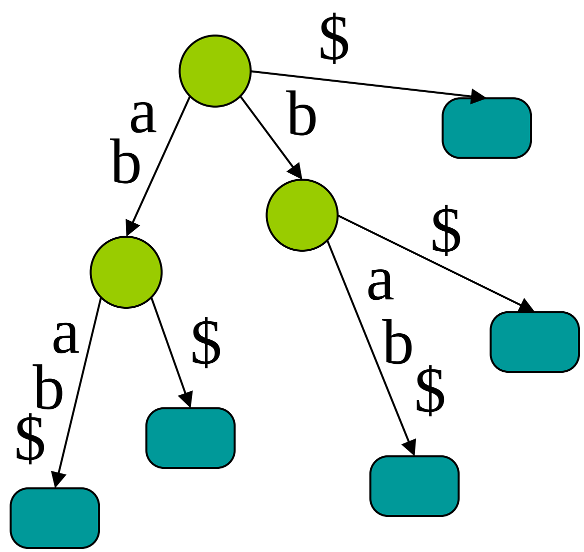
Given a string s a suffix tree of s is a compressed trie of all suffixes of s

To make these suffixes prefix-free we add a special character, say $, at the end of s

Suffix tree (Example)

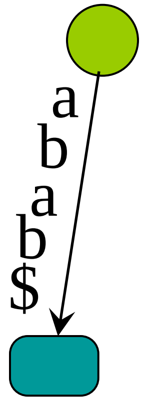
Let s=abab, a suffix tree of s is a compressed trie of all suffixes of s=abab$

{
$
b$
ab$
bab$
abab$
}

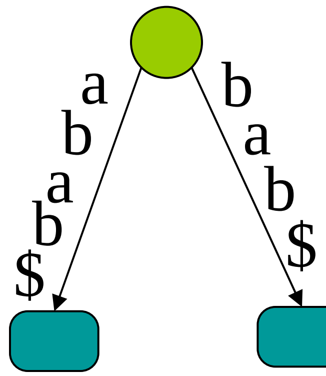


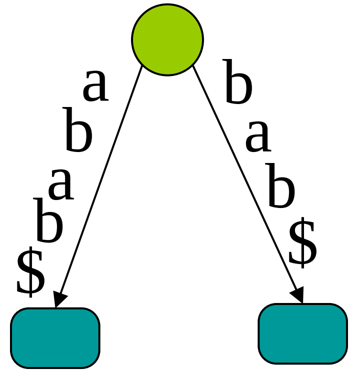
Trivial algorithm to build a Suffix tree

Put the largest suffix in

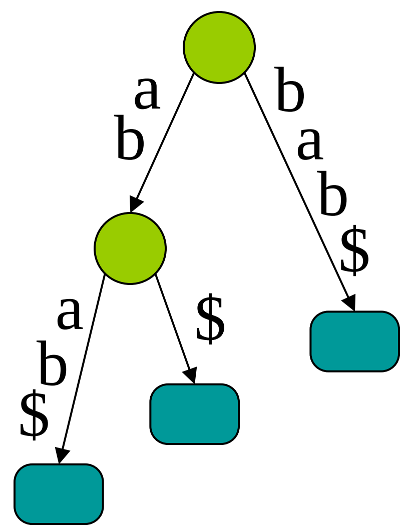


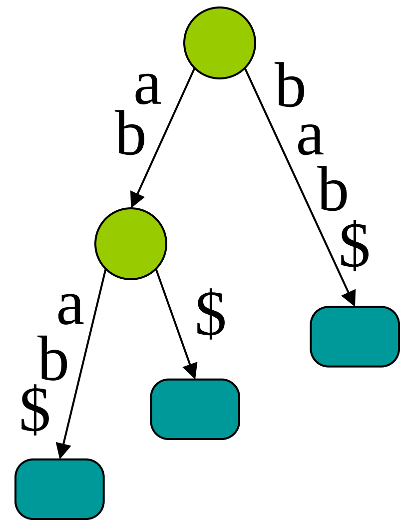
Put the suffix bab$ in



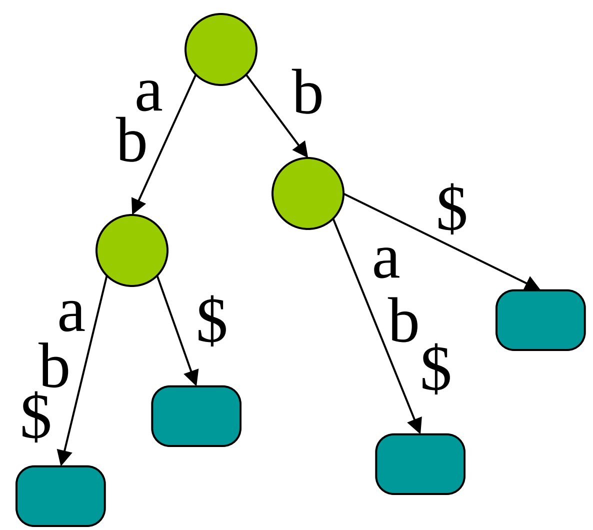


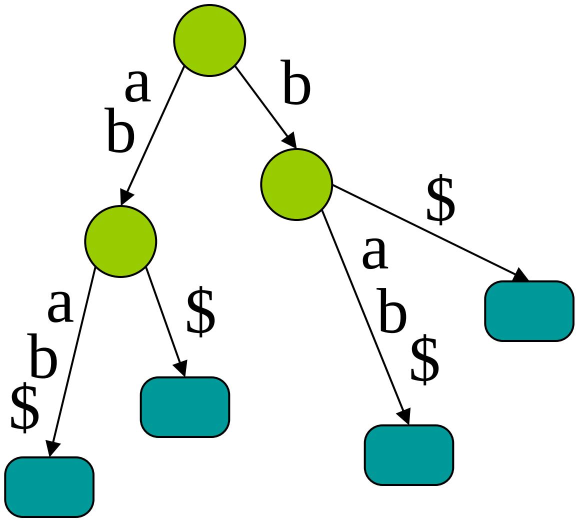
Put the suffix ab$ in



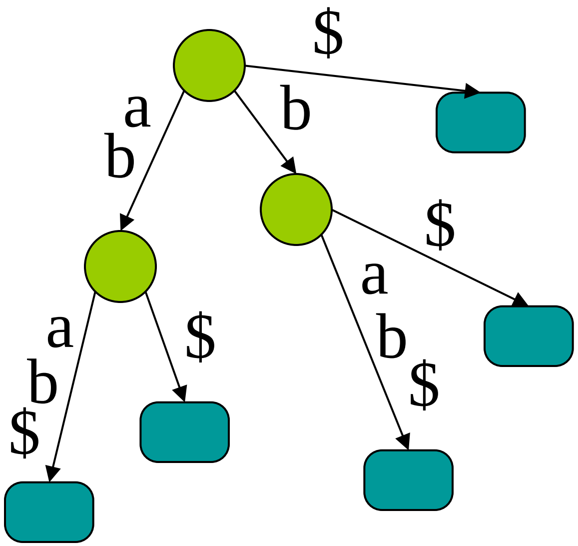


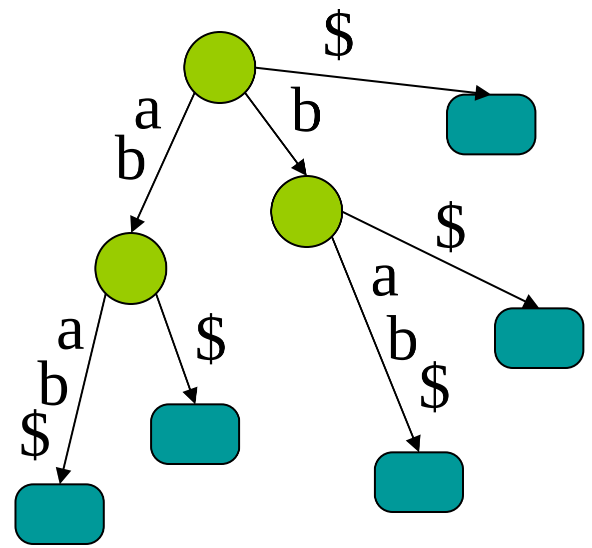
Put the suffix b$ in



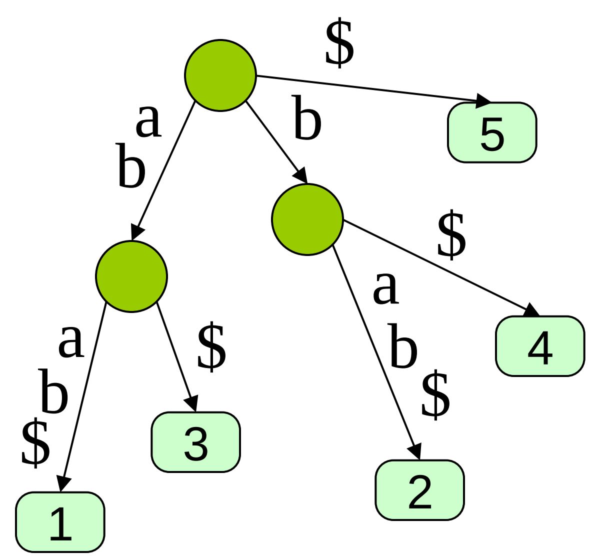


Put the suffix $ in





We will **also** label each leaf with the starting point of the corres. suffix.



Analysis

Takes $O(n^2)$ time to build.

We will see how to do it in $O(n)$ time

What can we do with it ?

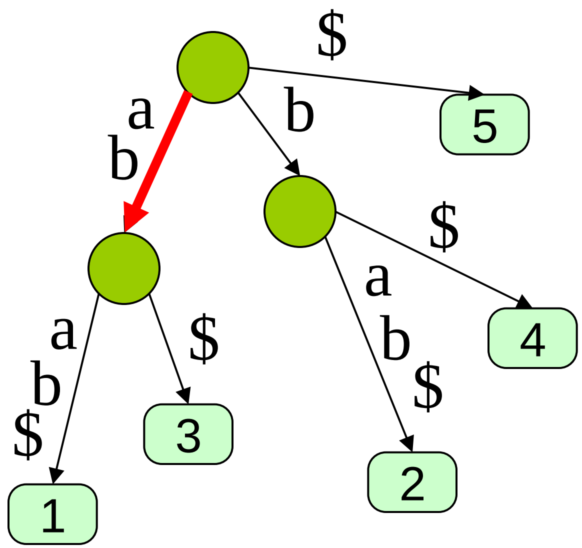
Exact string matching:

Given a Text T, $|T| = n$, preprocess it such that when a pattern P, $|P|=m$, arrives you can quickly decide when it occurs in T.

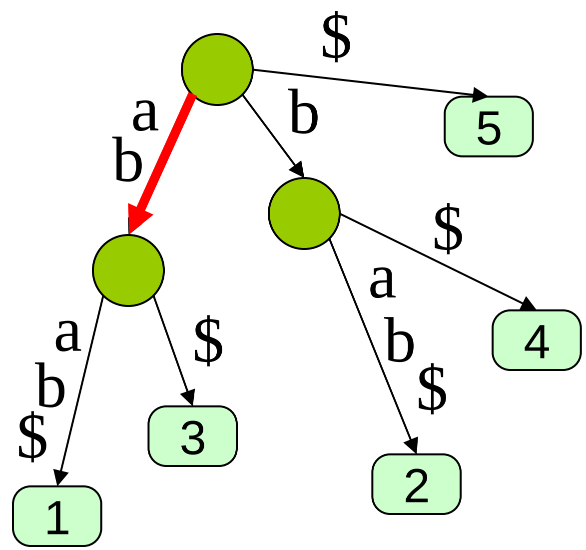
We may also want to find all occurrences of P in T

Exact string matching

In preprocessing we just build a suffix tree in O(n) time



Given a pattern P = ab we traverse the tree according to the pattern.



If we did not get stuck traversing the pattern then the pattern occurs in the text.

Each leaf in the subtree below the node we reach corresponds to an occurrence.

By traversing this subtree we get all k occurrences in O(n+k) time

Generalized suffix tree

Given a set of strings S a *generalized suffix tree* of S is a compressed trie of all suffixes of $s \in S$

To associate each suffix with a unique string in S add a different special char to each s

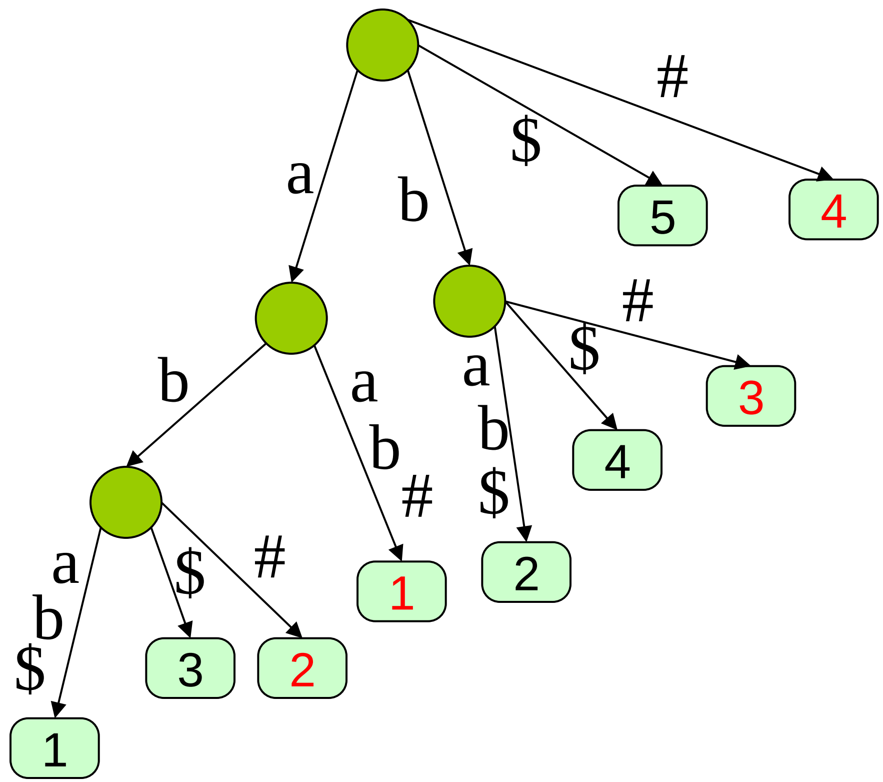
Generalized suffix tree (Example)

Let s_1=abab and s_2=aab here is a generalized suffix tree for s_1 and s_2

{

$	#
b$	b#
ab$	ab#
bab$	aab#
abab$	

}

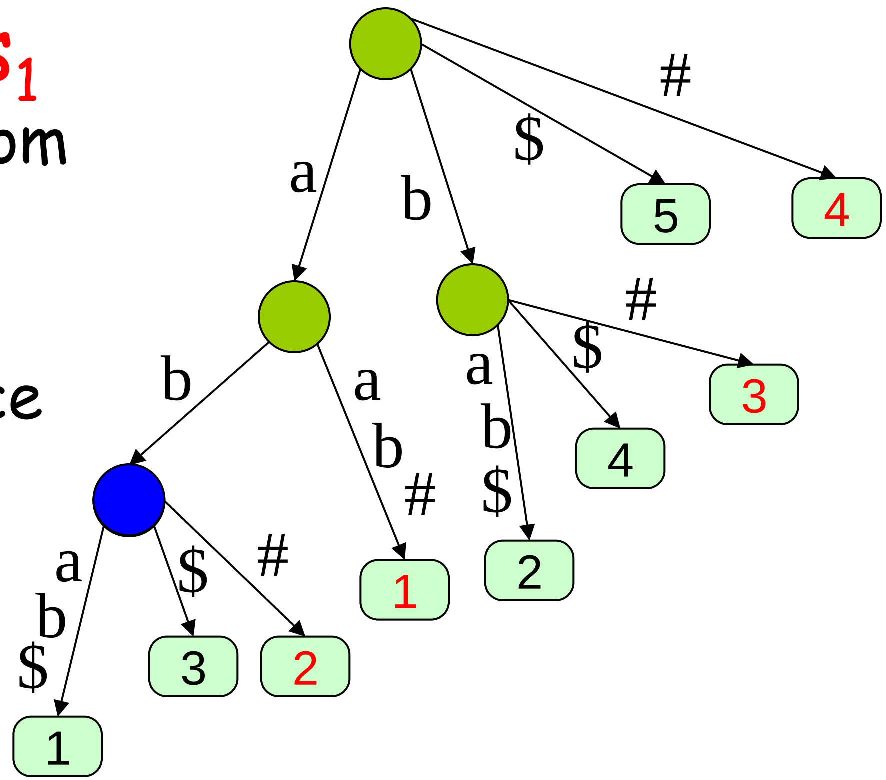


So what can we do with it ?

Matching a pattern against a database of strings

Longest common substring (of two strings)

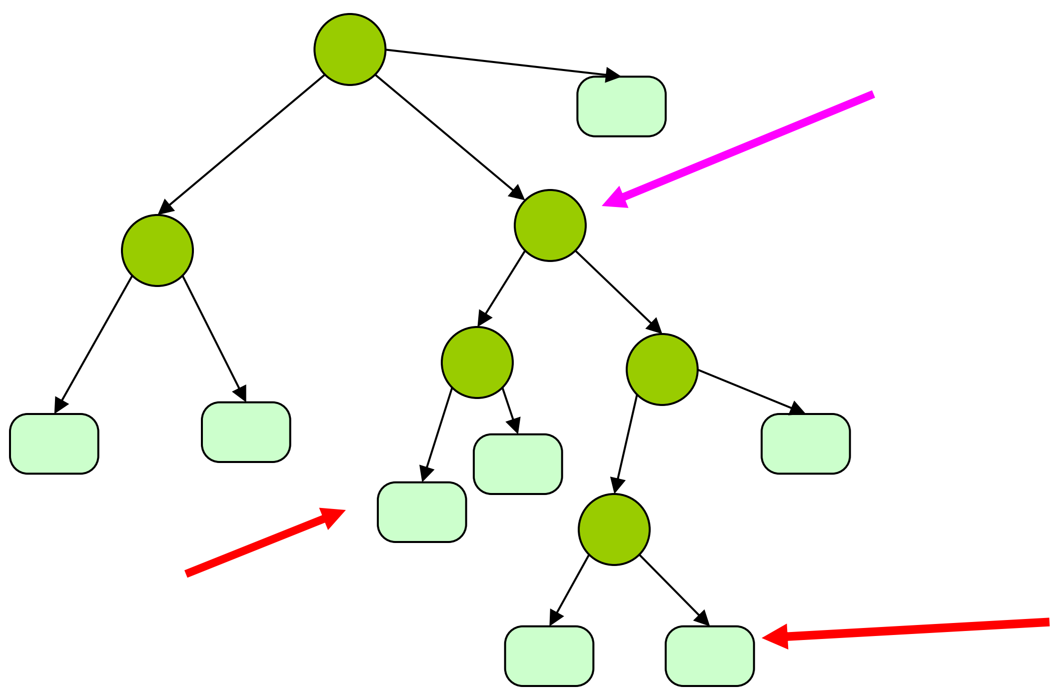
Every node with a leaf descendant from string S_1 and a leaf descendant from string S_2 represents a maximal common substring and vice versa.



Find such node with largest "string depth"

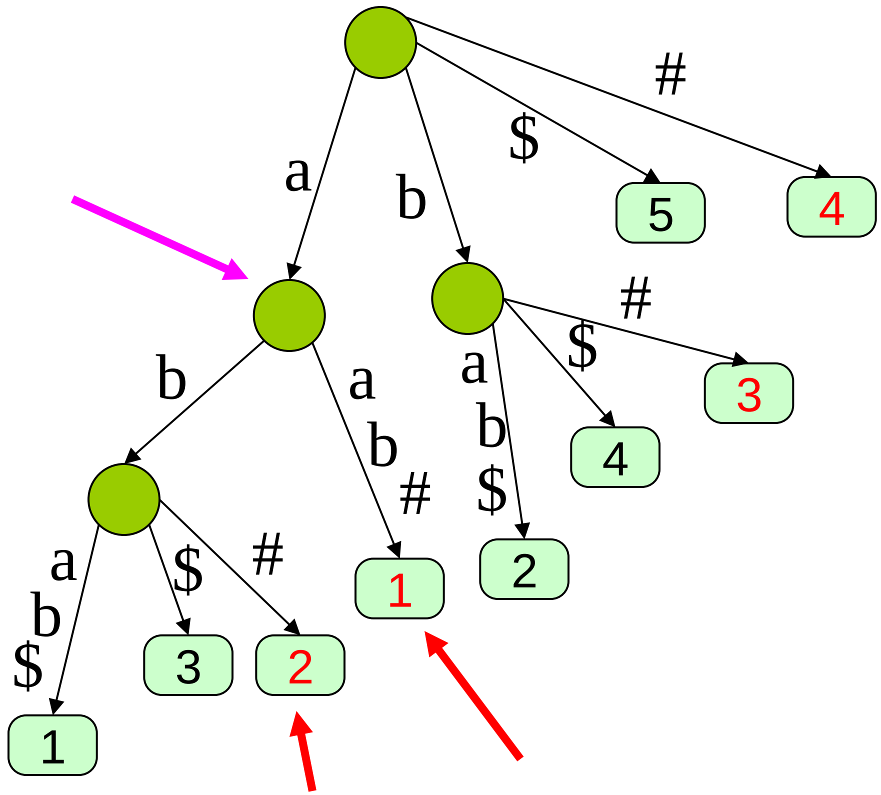
Lowest common ancetors

A lot more can be gained from the suffix tree if we preprocess it so that we can answer LCA queries on it



Why?

The LCA of two leaves represents the longest common prefix (LCP) of these 2 suffixes



Finding maximal palindromes

- A palindrome: caabaac, cbaabc
- Want to find all maximal palindromes in a string s

Let s = cbaaba

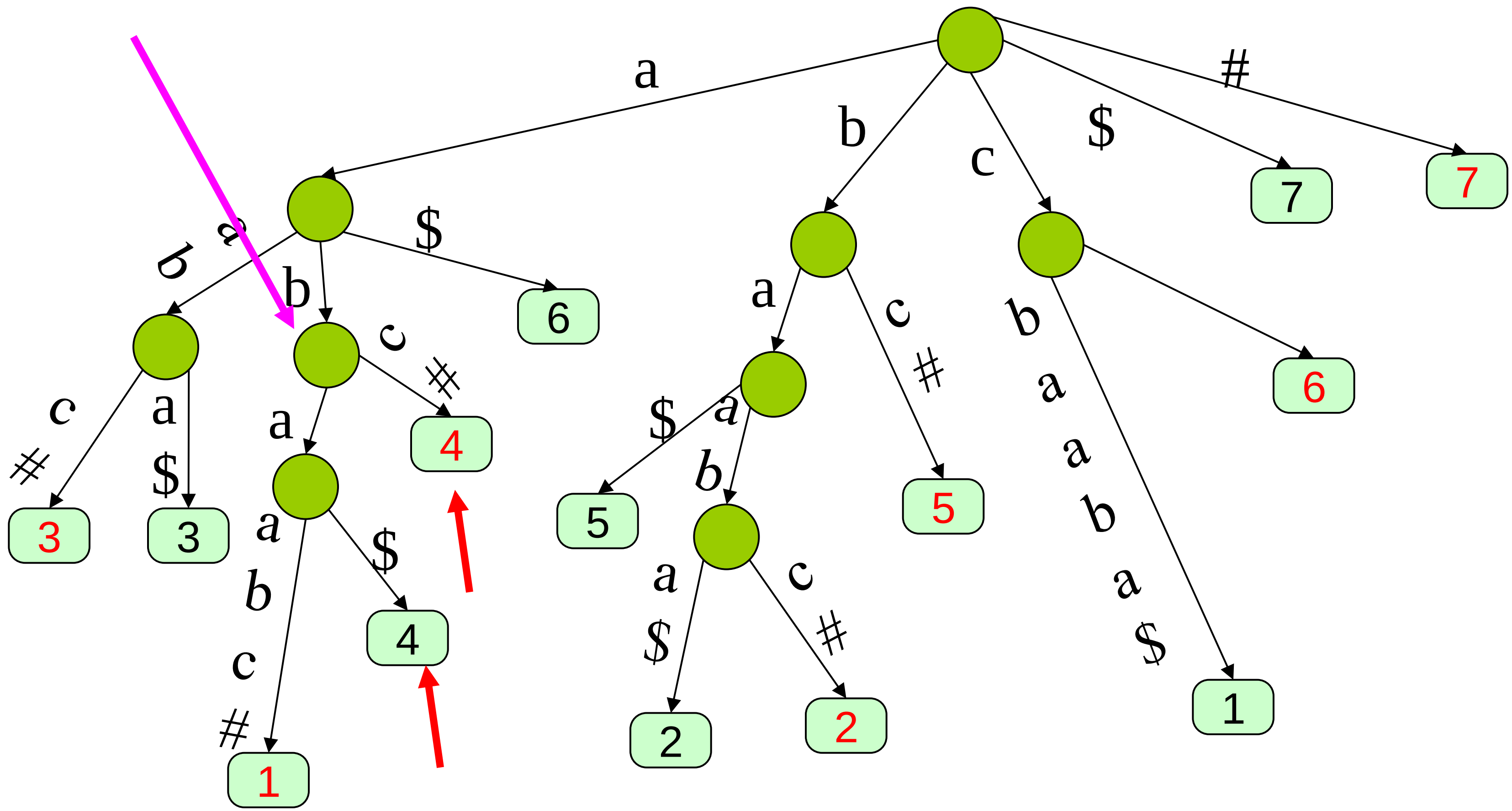
The maximal palindrome with center between i-1 and i is the LCA of the suffix at position i of s and the suffix at position m-i+1 of s^r

Maximal palindromes algorithm

Prepare a generalized suffix tree for
s = cbaaba$ and s^r = abaabc#

For every i find the LCA of suffix i of s and suffix m-i+1 of s^r

Let s = cbaaba$ then s^r = abaabc#



Analysis

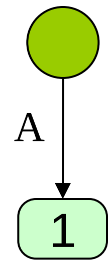
O(n) time to identify all palindromes

Can we construct a suffix tree in linear time ?

Ukkonen's *linear* time construction

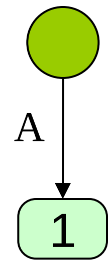
ACTCAATC

A



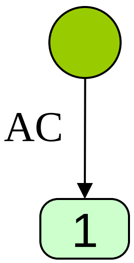
ACTAATC

AC



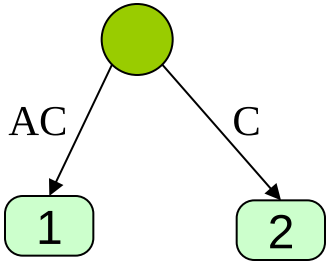
ACTAATC

AC



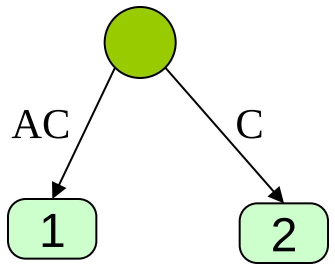
ACTAATC

AC



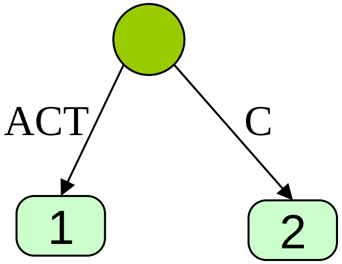
ACTAATC

ACT



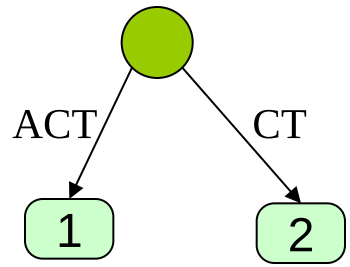
ACTAATC

ACT



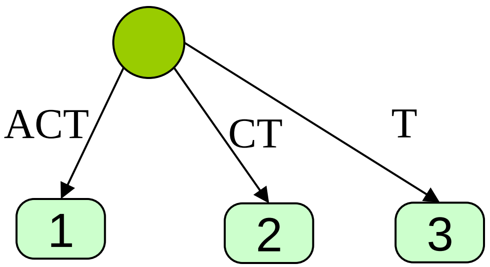
ACTAATC

ACT



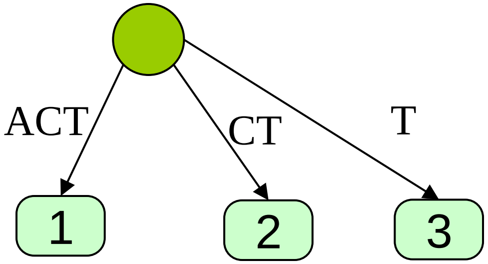
ACTAATC

ACT



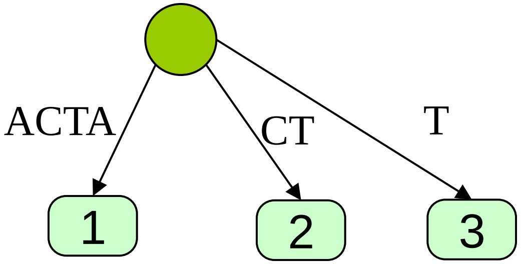
ACTAATC

ACTA



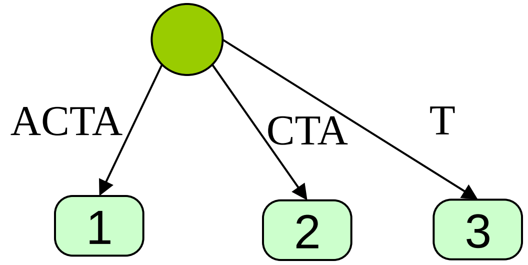
ACTAATC

ACTA



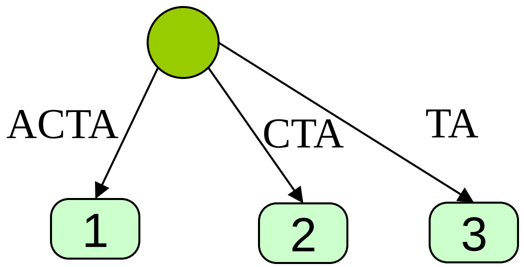
ACTAATC

ACTA



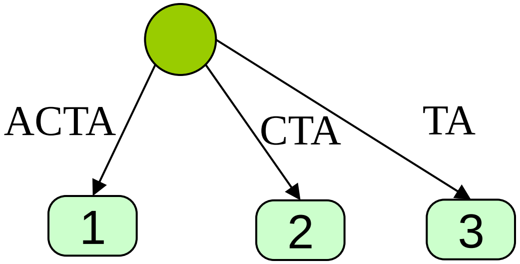
ACTAATC

ACTA



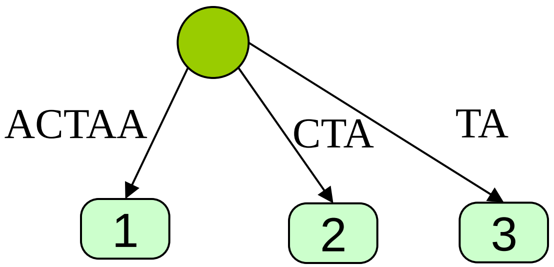
ACTAATC

ACTAA



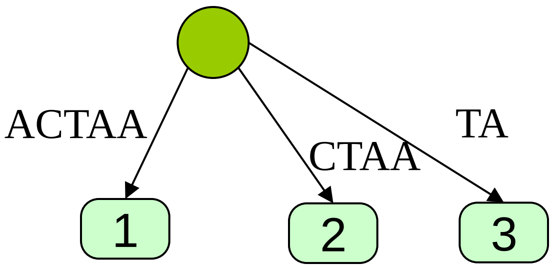
ACTAATC

ACTAA



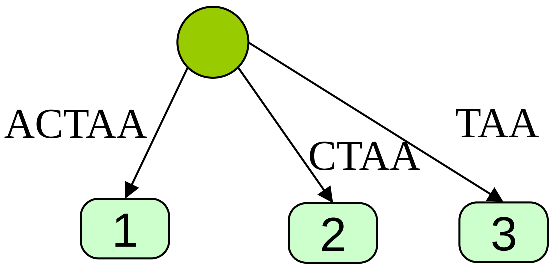
ACTAATC

ACTAA



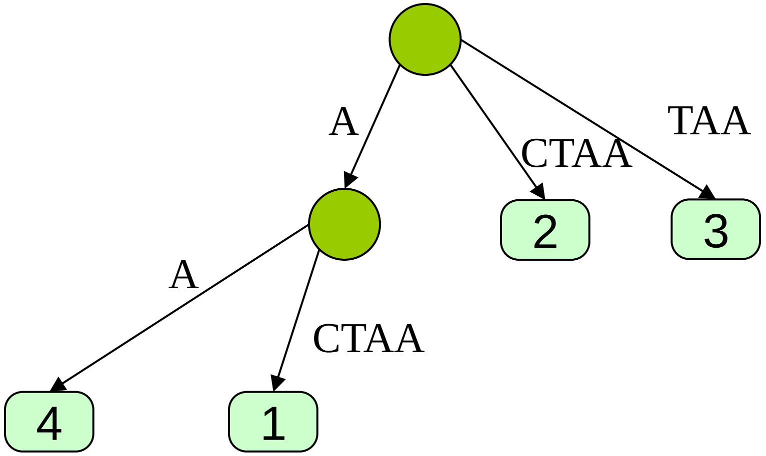
ACTAATC

ACTAA



ACTAATC

ACTAA



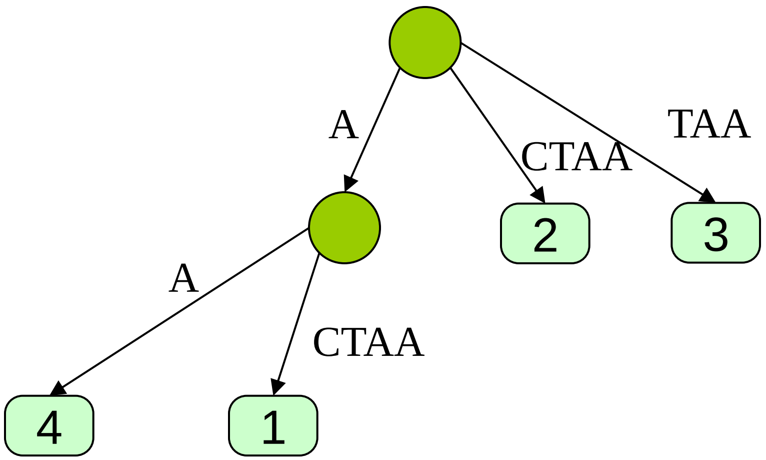
Phases & extensions

- Phase i is when we add character i

- In phase i we have i extensions of suffixes

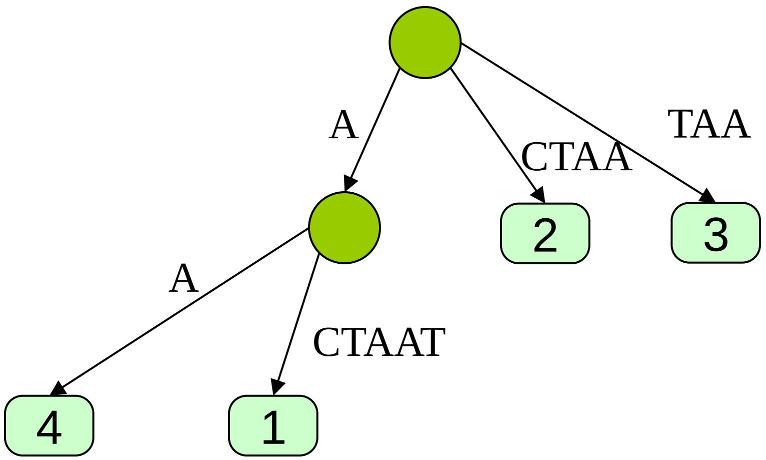
ACTAATC

ACTAAT



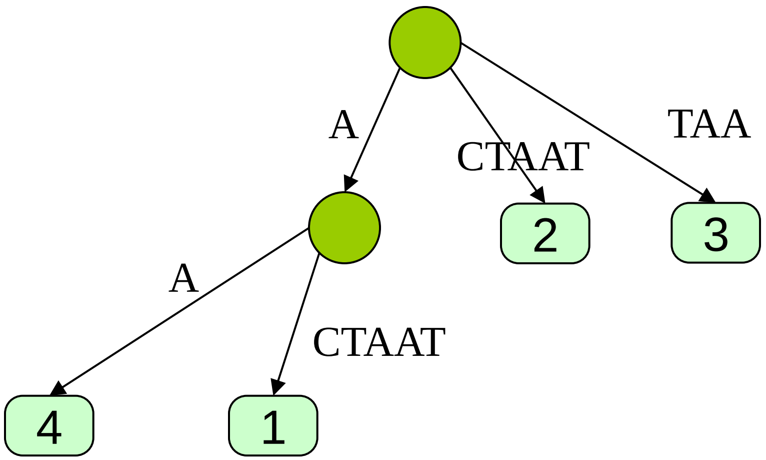
ACTAATC

ACTAAT



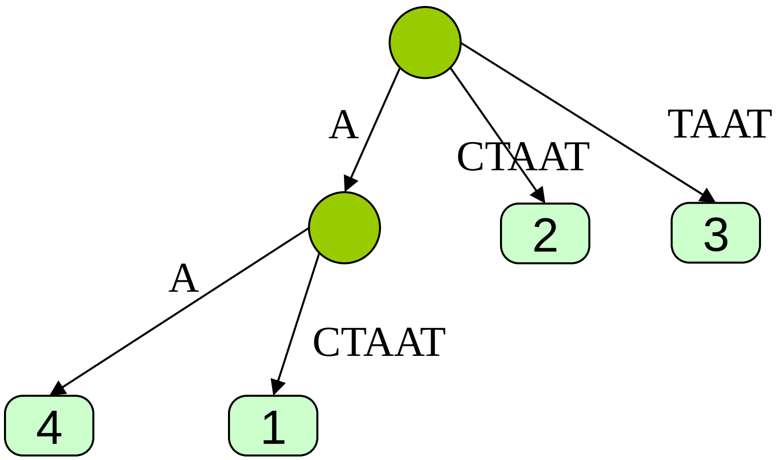
ACTAATC

ACTAAT



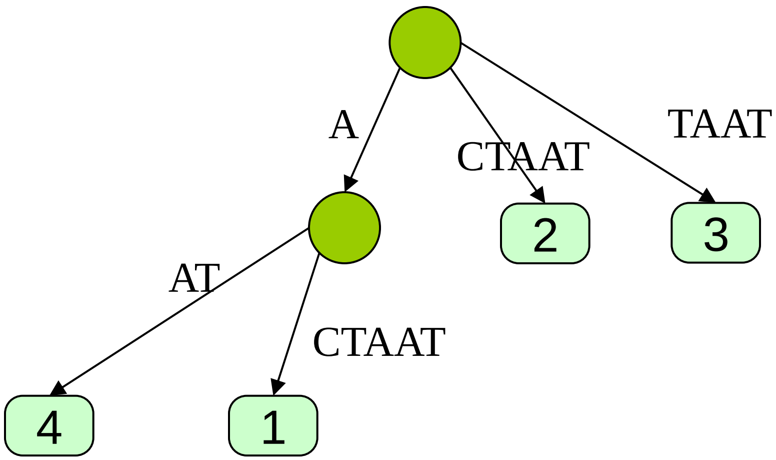
ACTAATC

ACTAAT



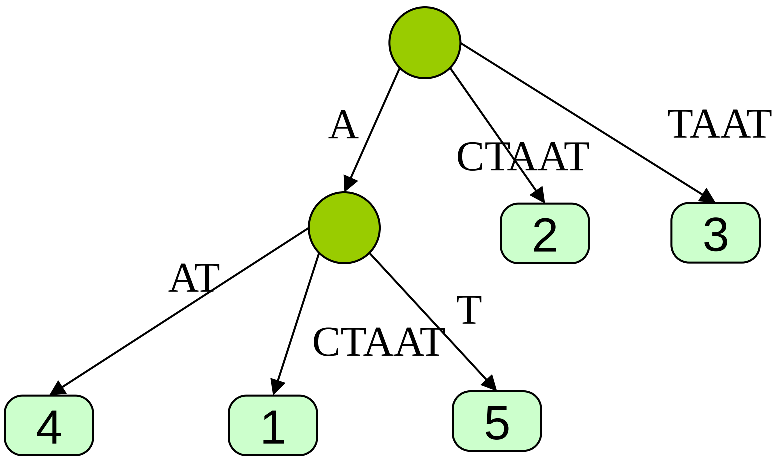
ACTAATC

ACTAAT



ACTAATC

ACTAAT

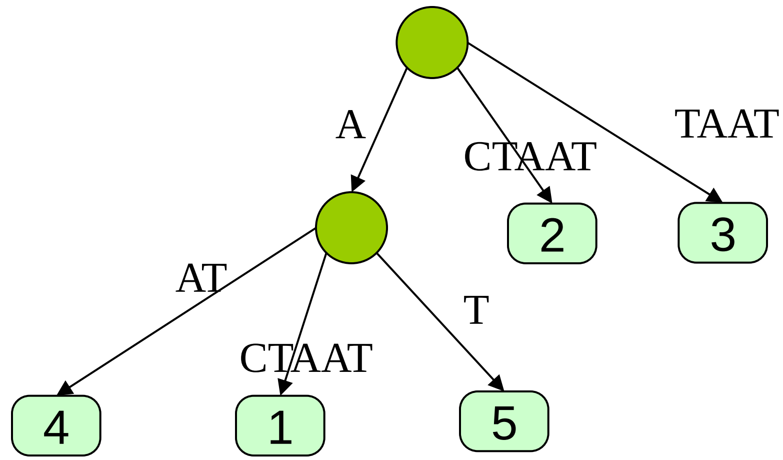


Extension rules

- Rule 1: The suffix ends at a leaf, you add a character on the edge entering the leaf
- Rule 2: The suffix ended internally and the extended suffix does not exist, you add a leaf and possibly an internal node
- Rule 3: The suffix exists and the extended suffix exists, you do nothing

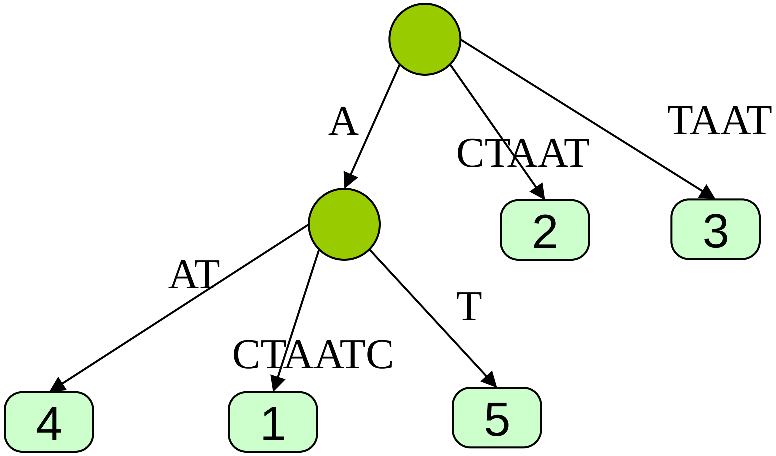
ACTAATC

ACTAATC



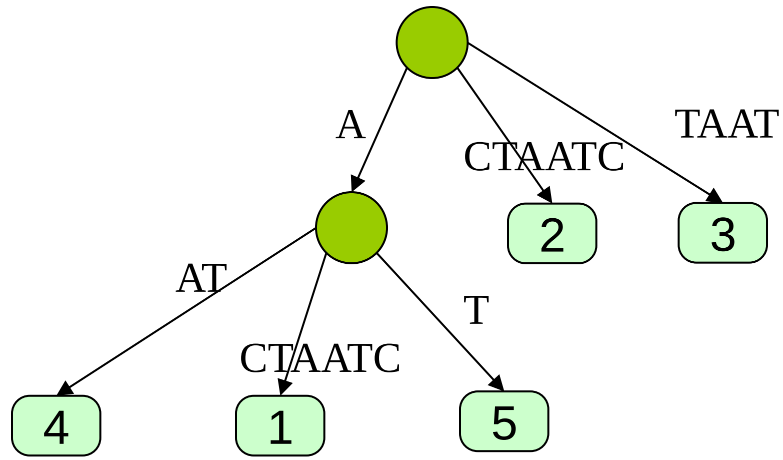
ACTAATC

ACTAATC



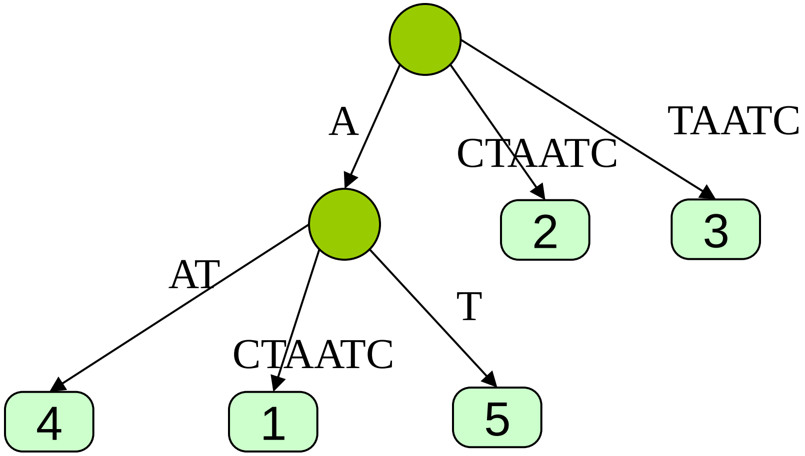
ACTAATC

ACTAATC



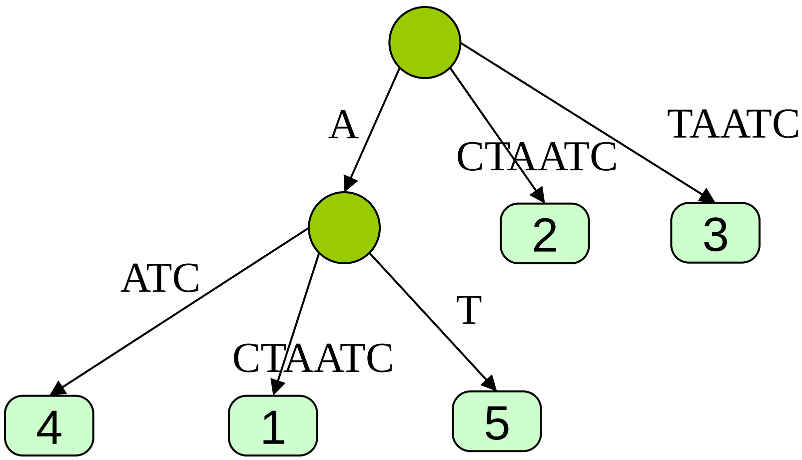
ACTAATC

ACTAATC



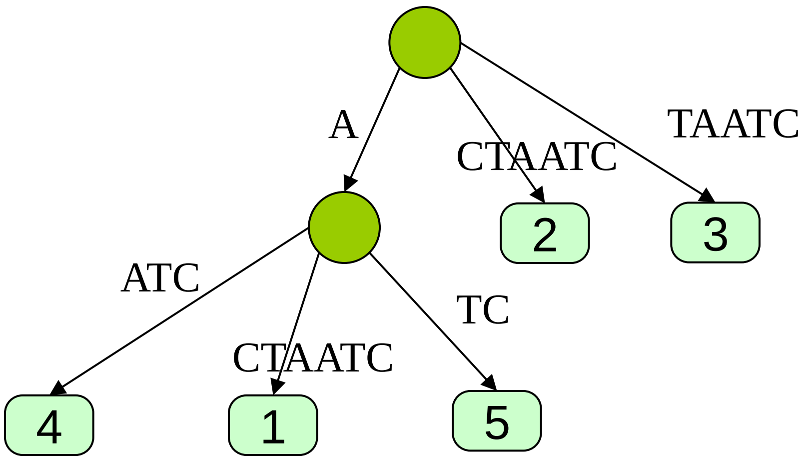
ACTAATC

ACTAATC



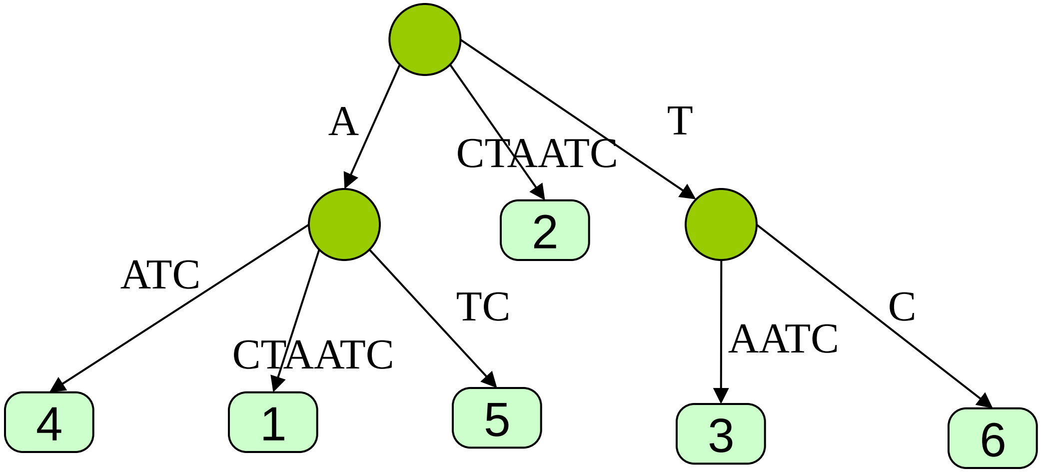
ACTAATC

ACTAATC



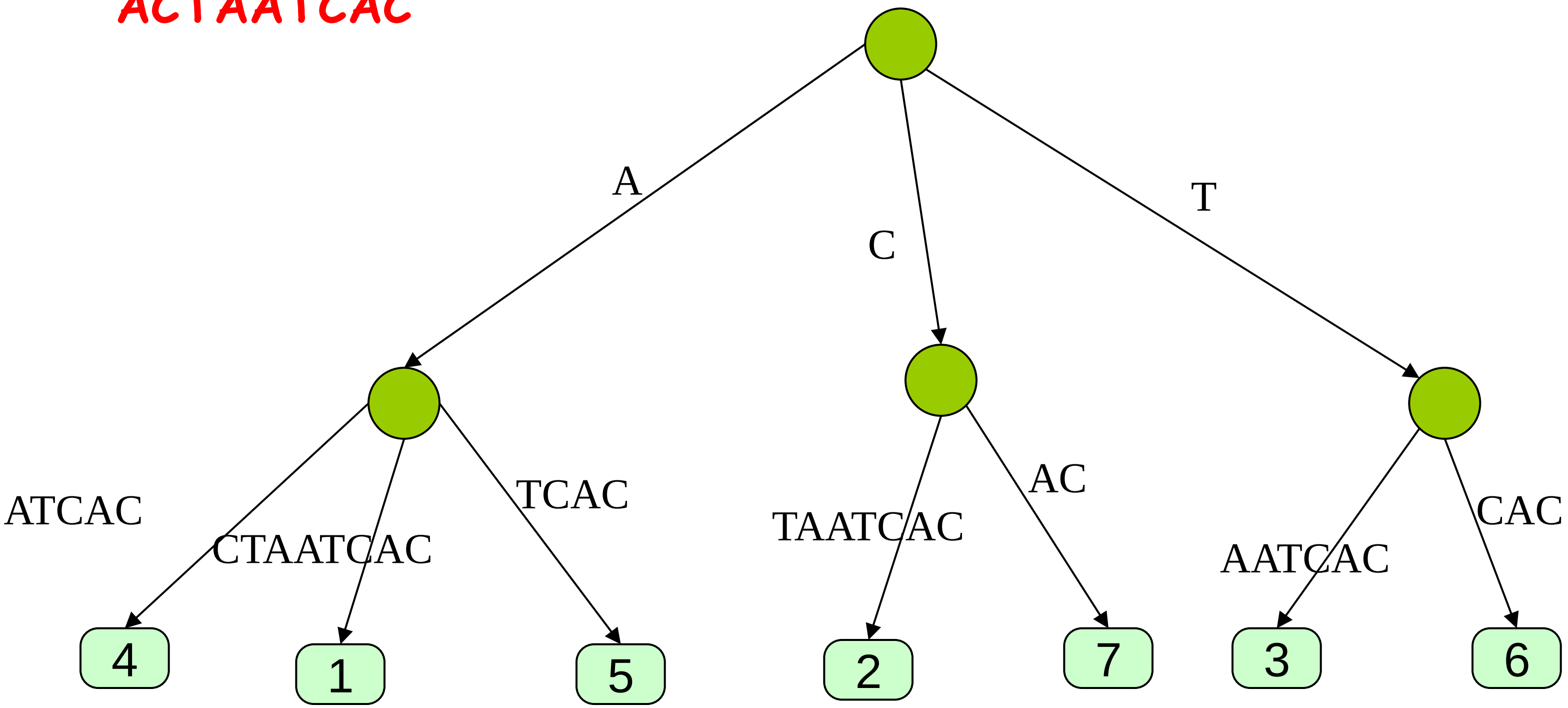
ACTAATC

ACTAATC

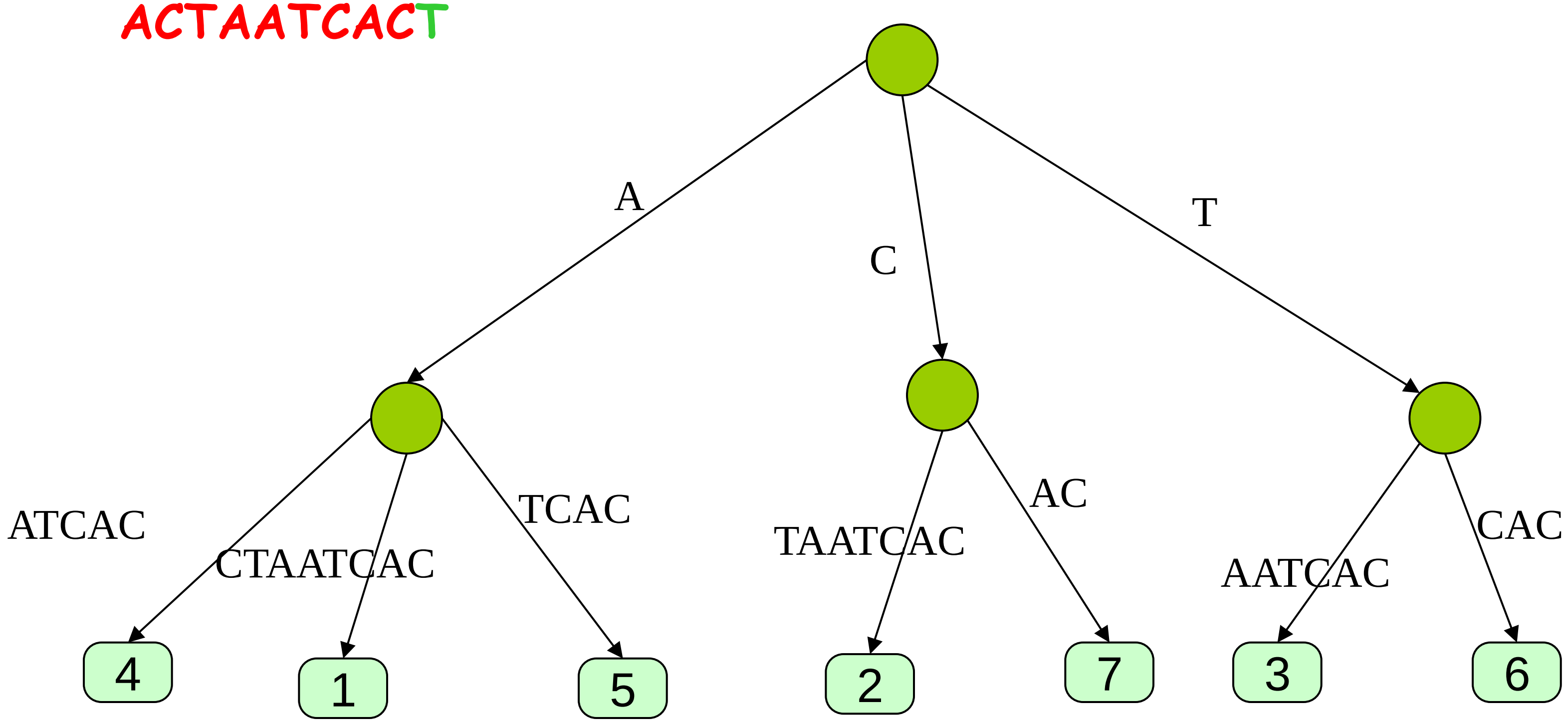


Skip forward..

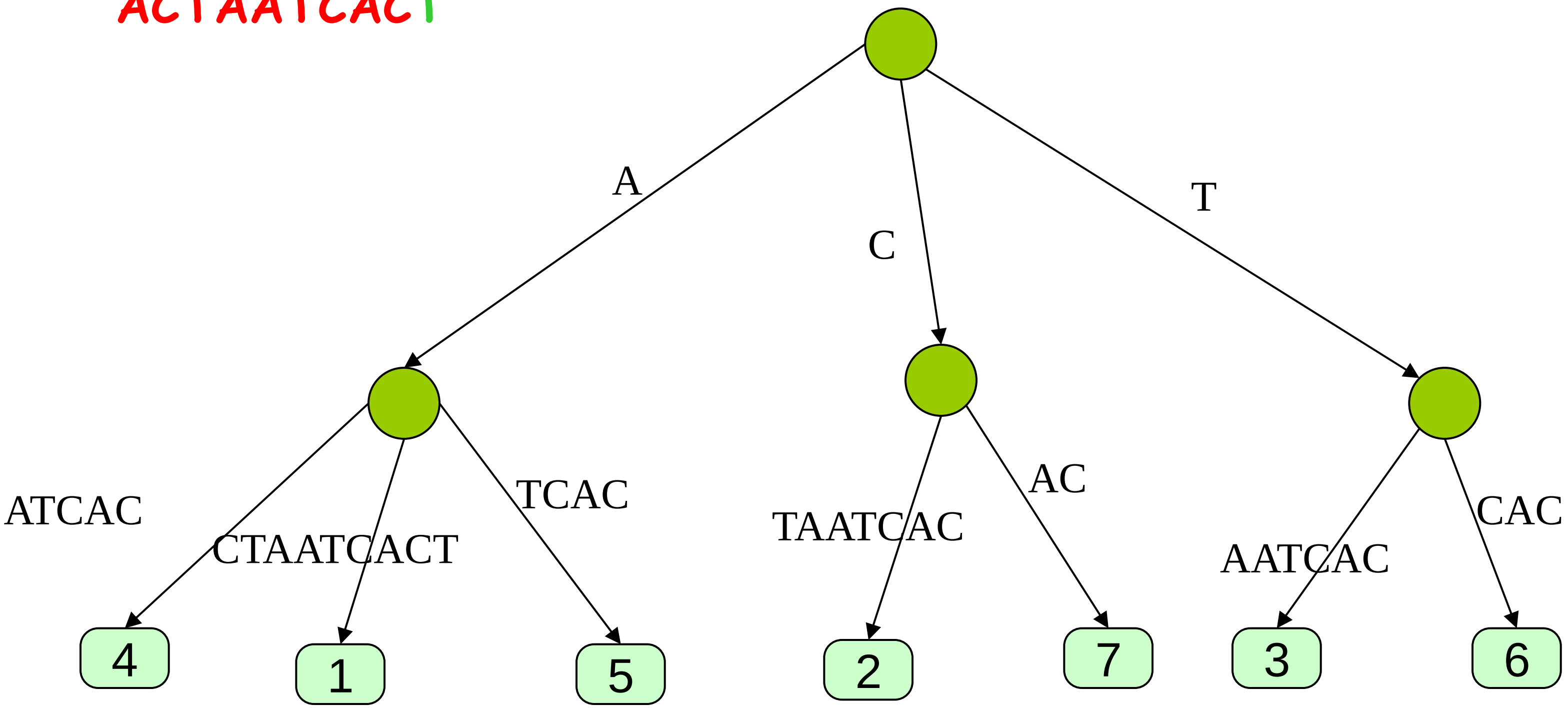
ACTAATCAC

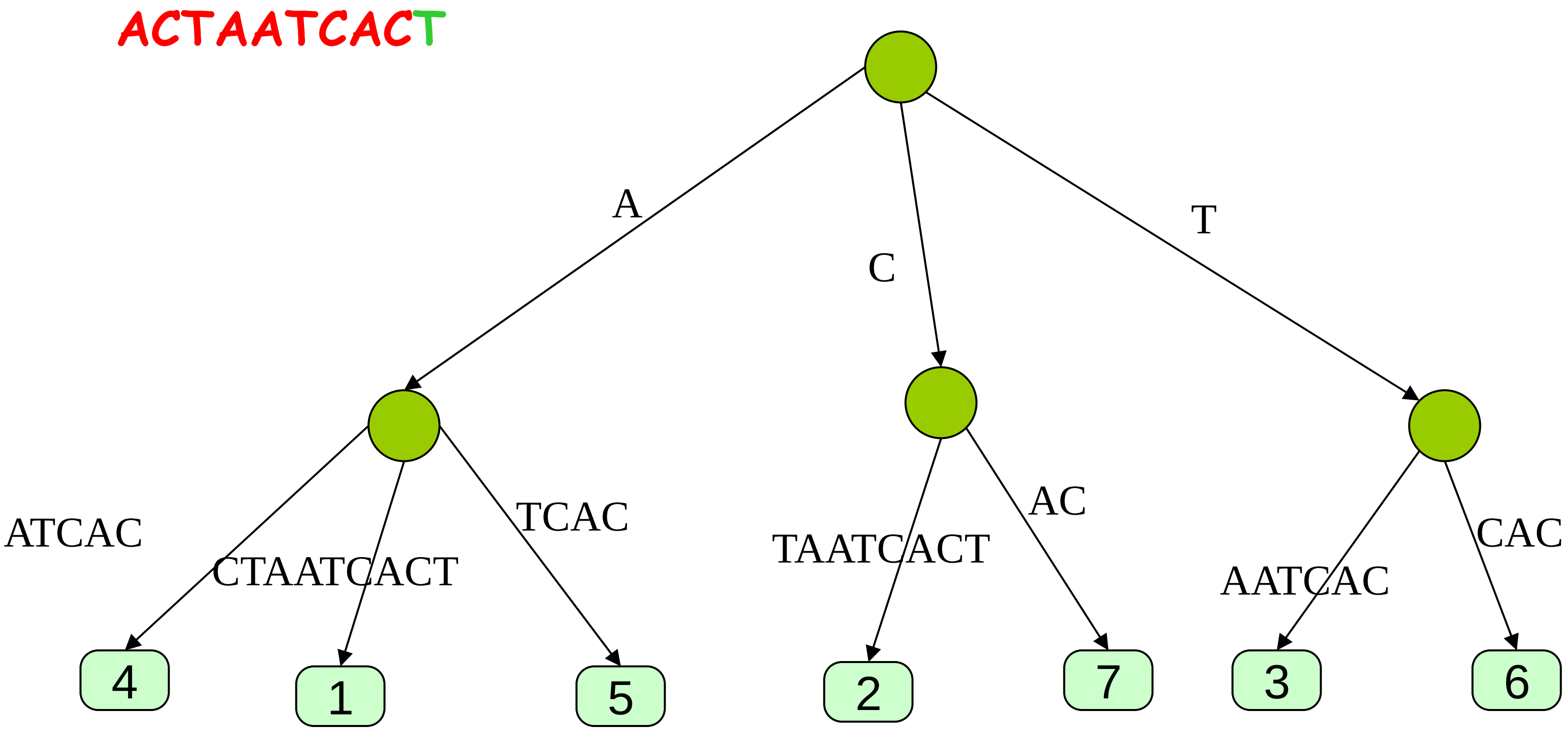


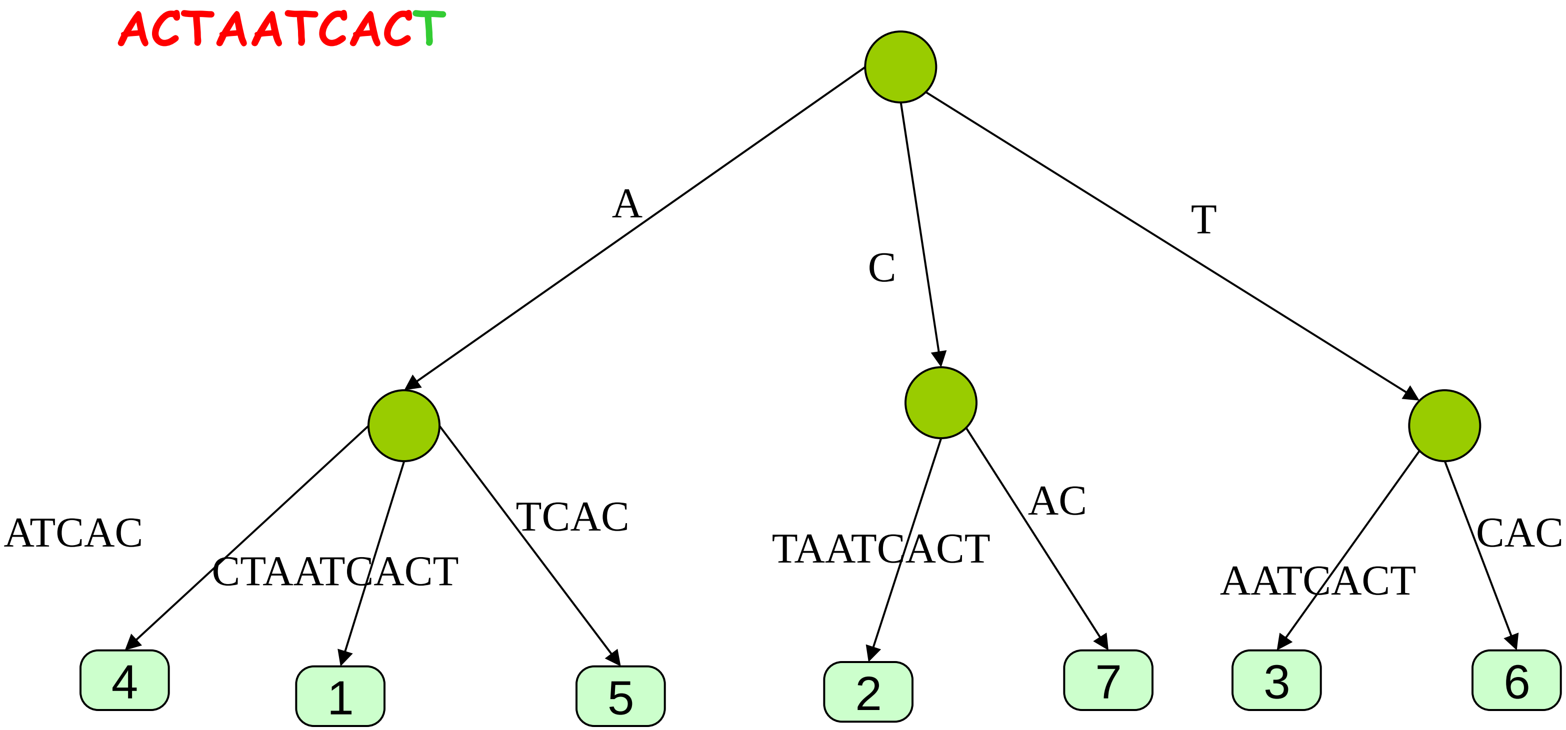
ACTAATCACT

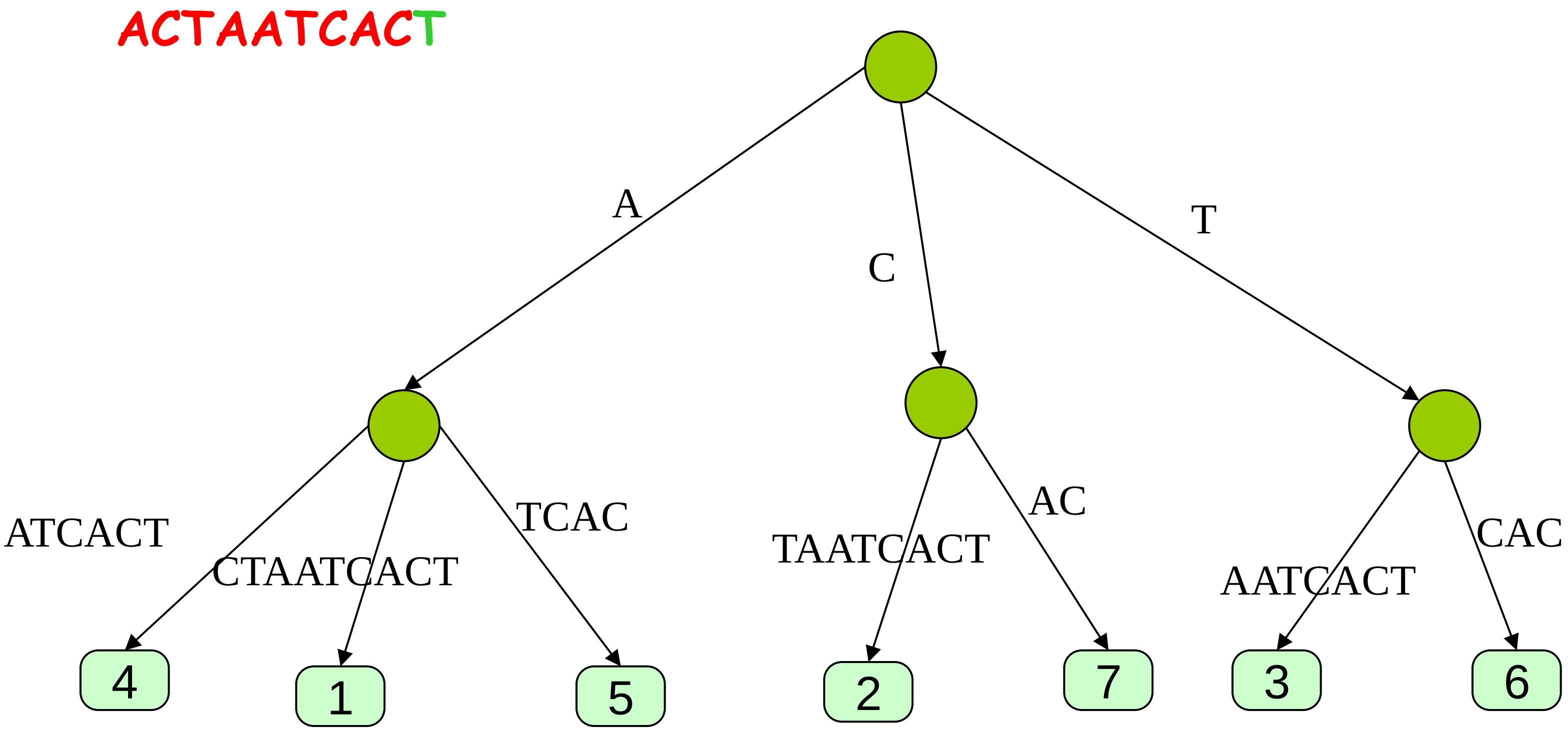


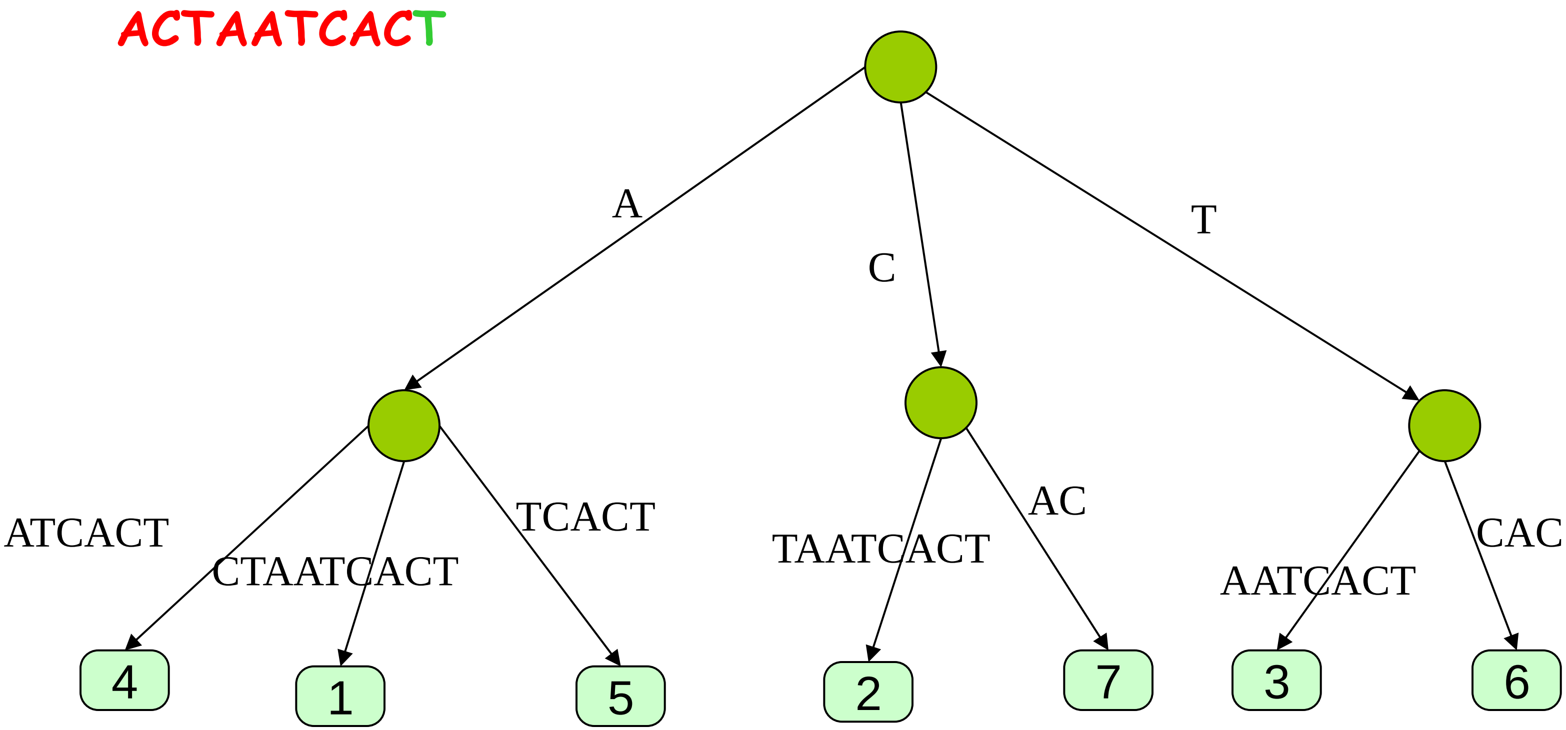
ACTAATCACT

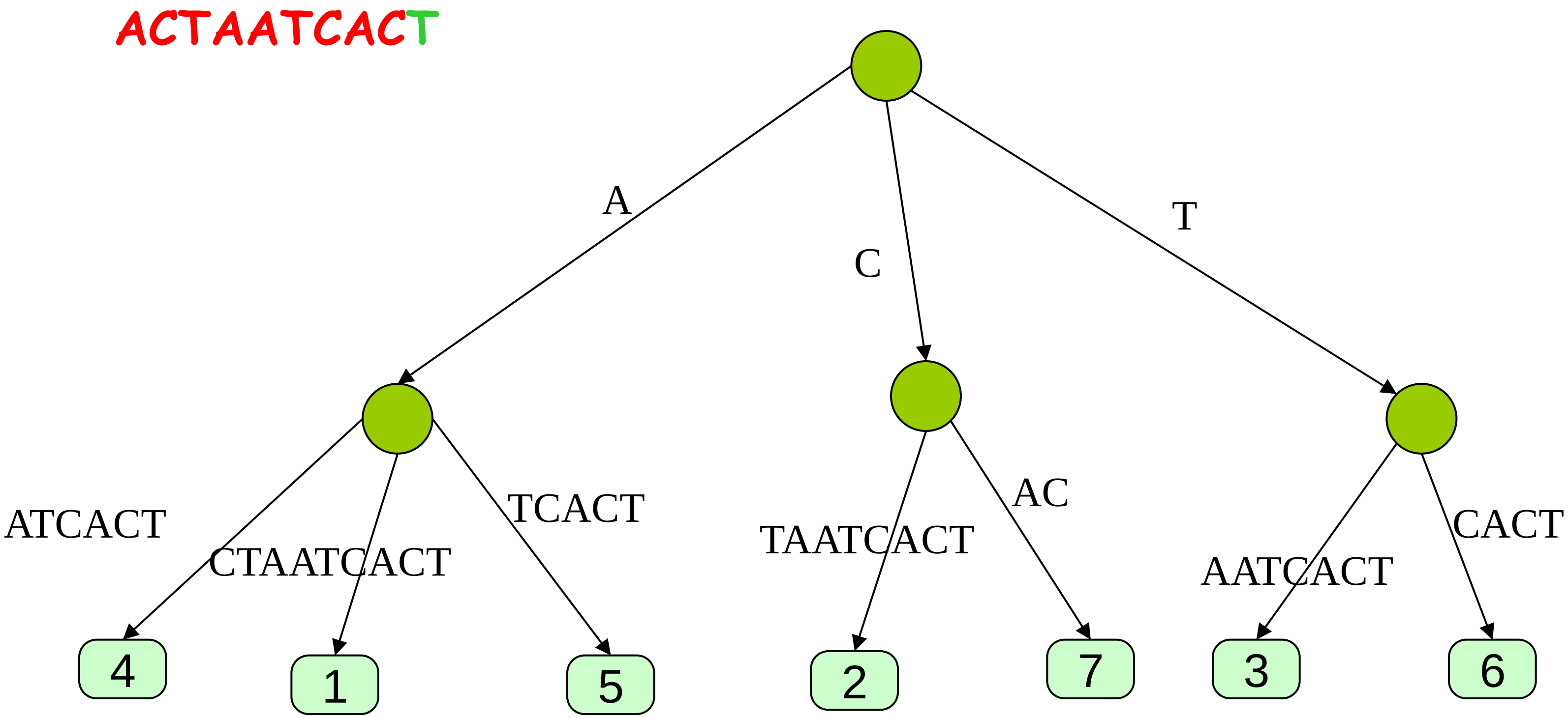



ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCAC
CAC
4
1
5
2
7
3
6


ACTAATCACT
A
C
T
ATCAC
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

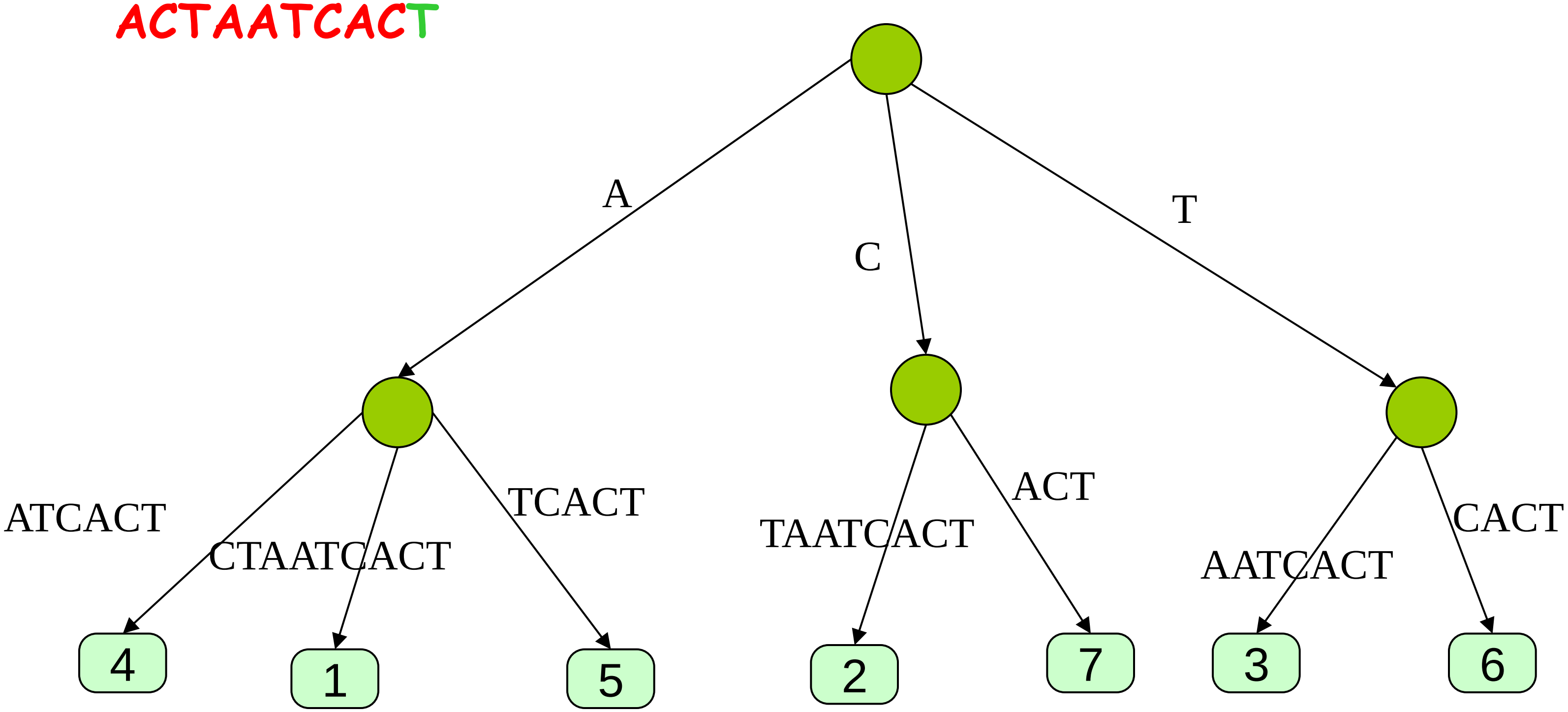

ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCAC
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6


ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CAC
4
1
5
2
7
3
6

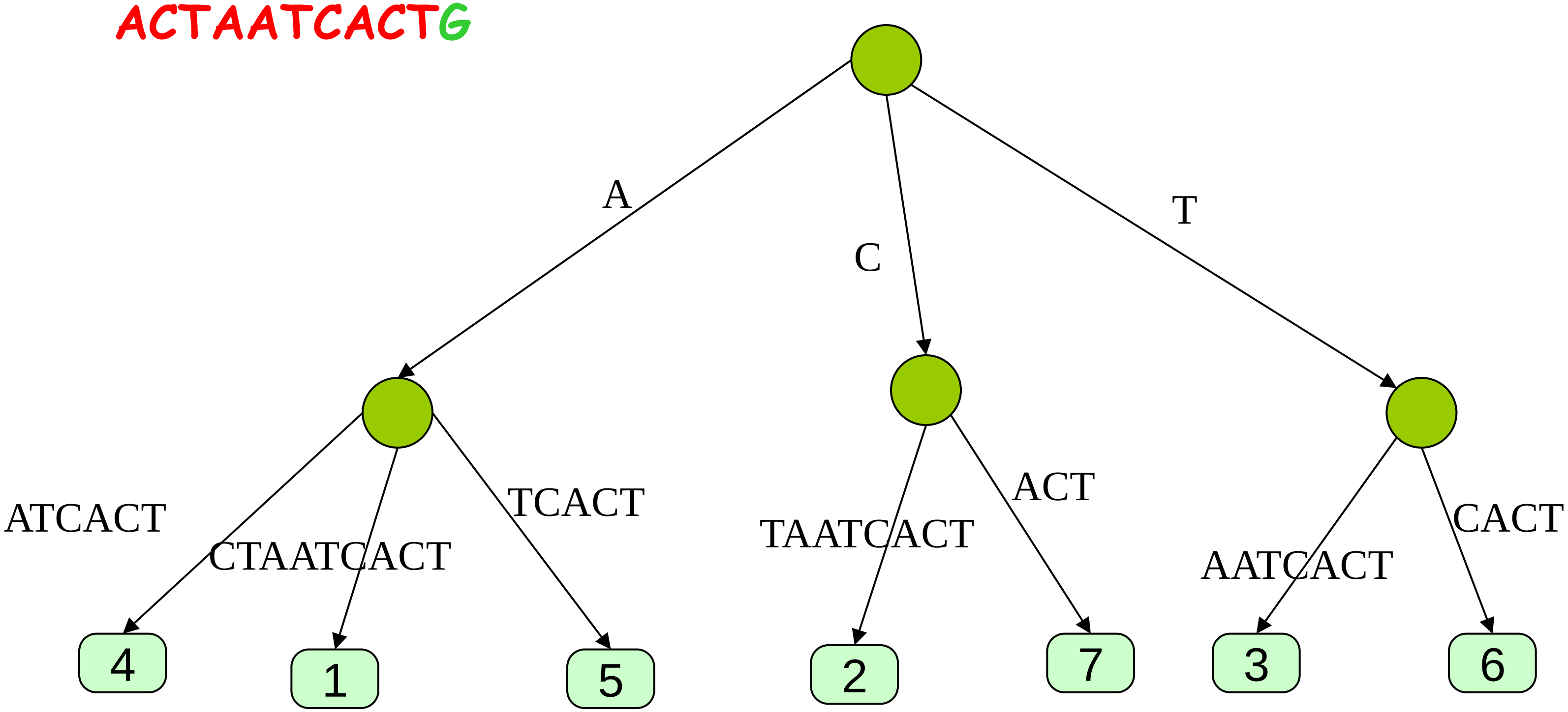


ACTAATCACT
A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
AC
AATCACT
CACT
4
1
5
2
7
3
6

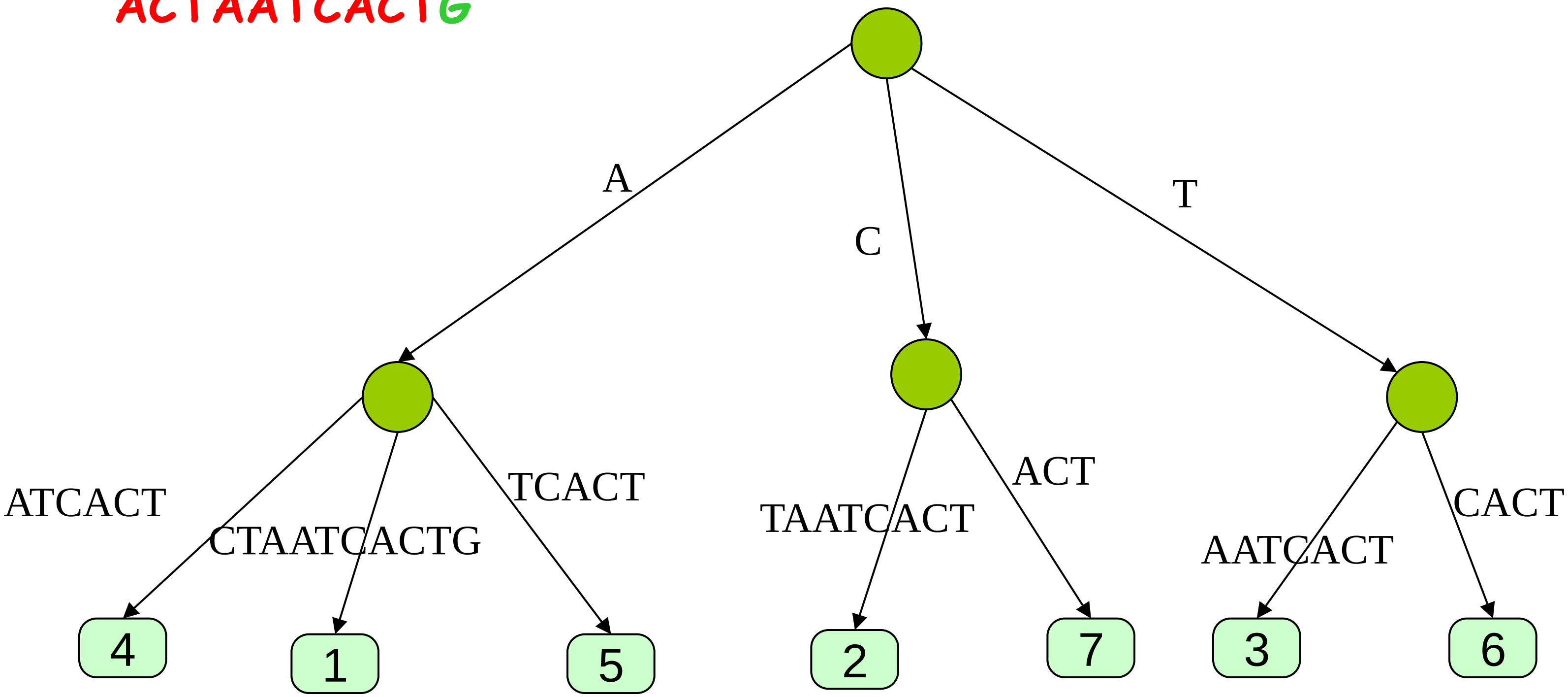
ACTAATCACT


A
C
T
ATCACT
CTAATCACT
TCACT
TAATCACT
ACT
AATCACT
CACT
4
1
5
2
7
3
6

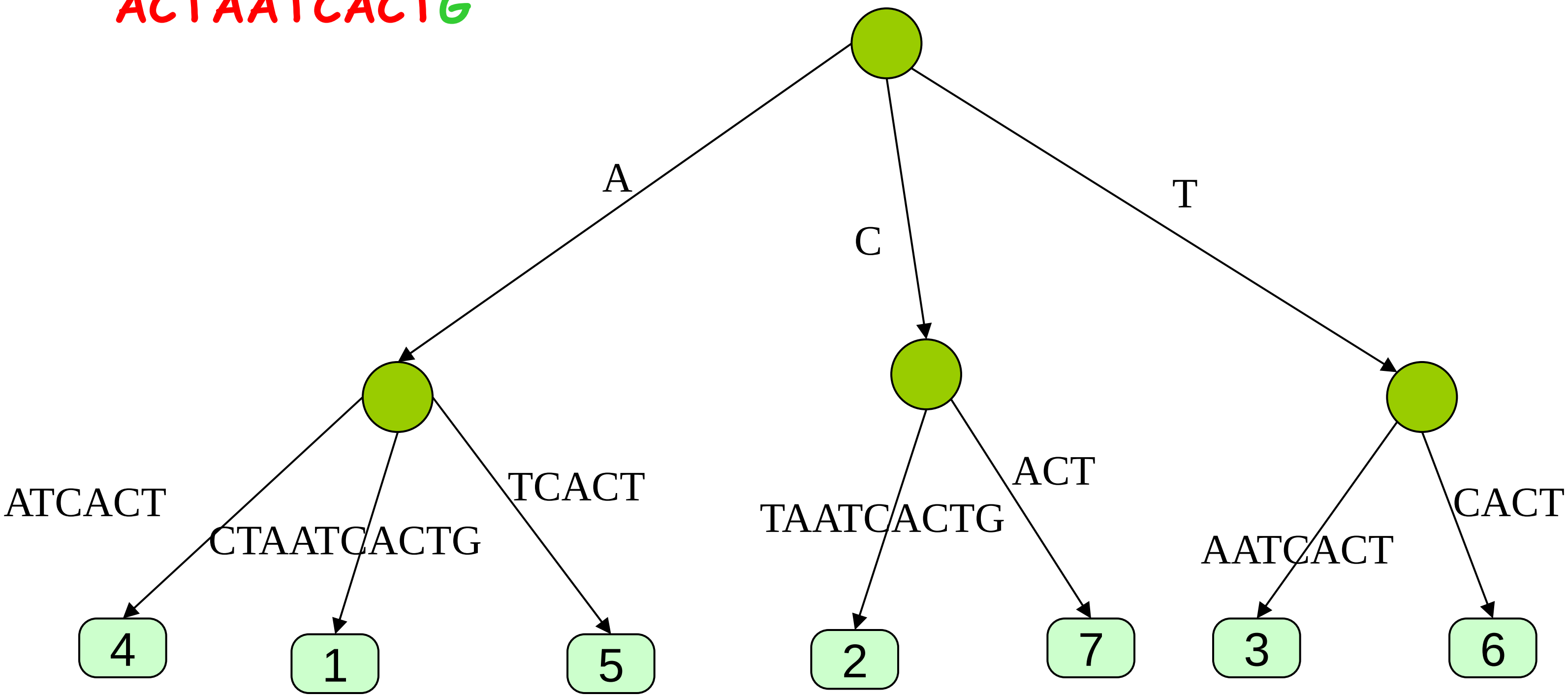
ACTAATCACTG



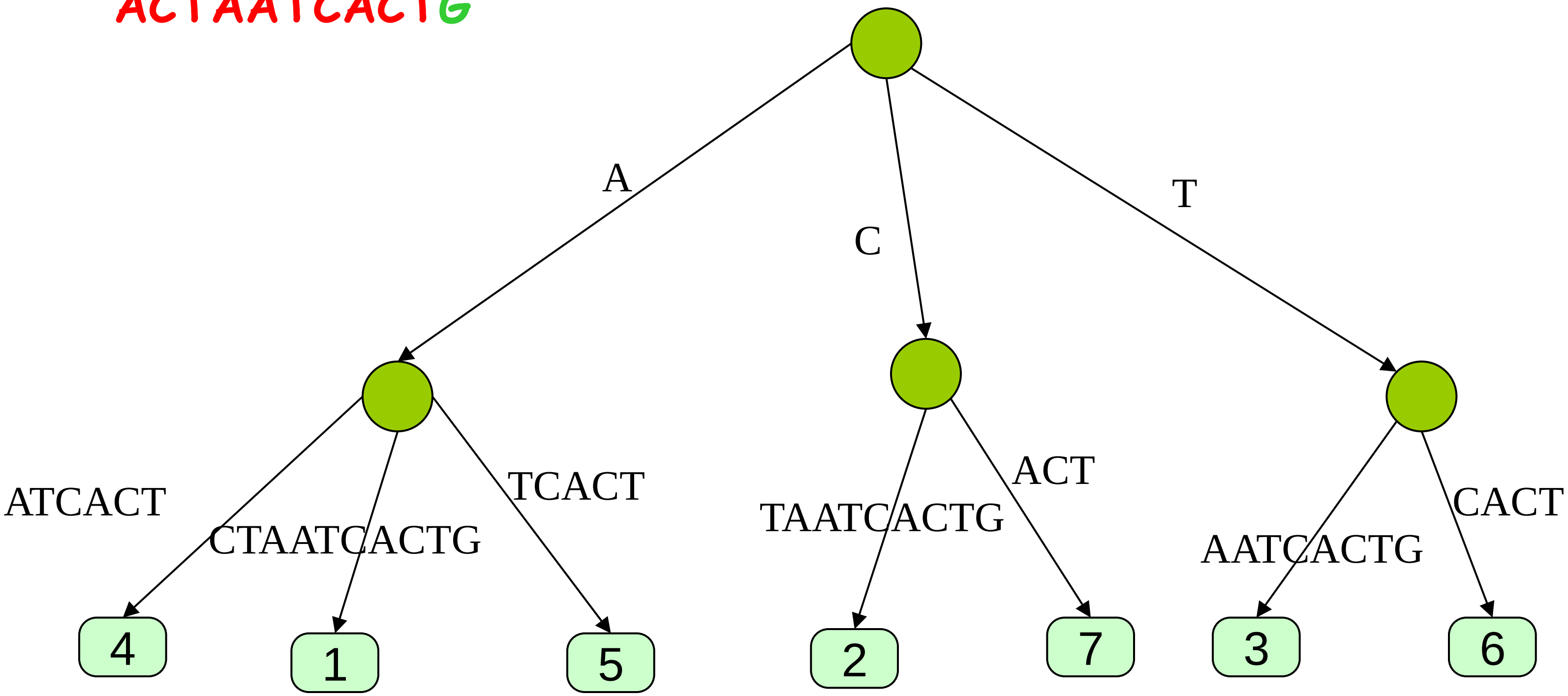
ACTAATCACTG



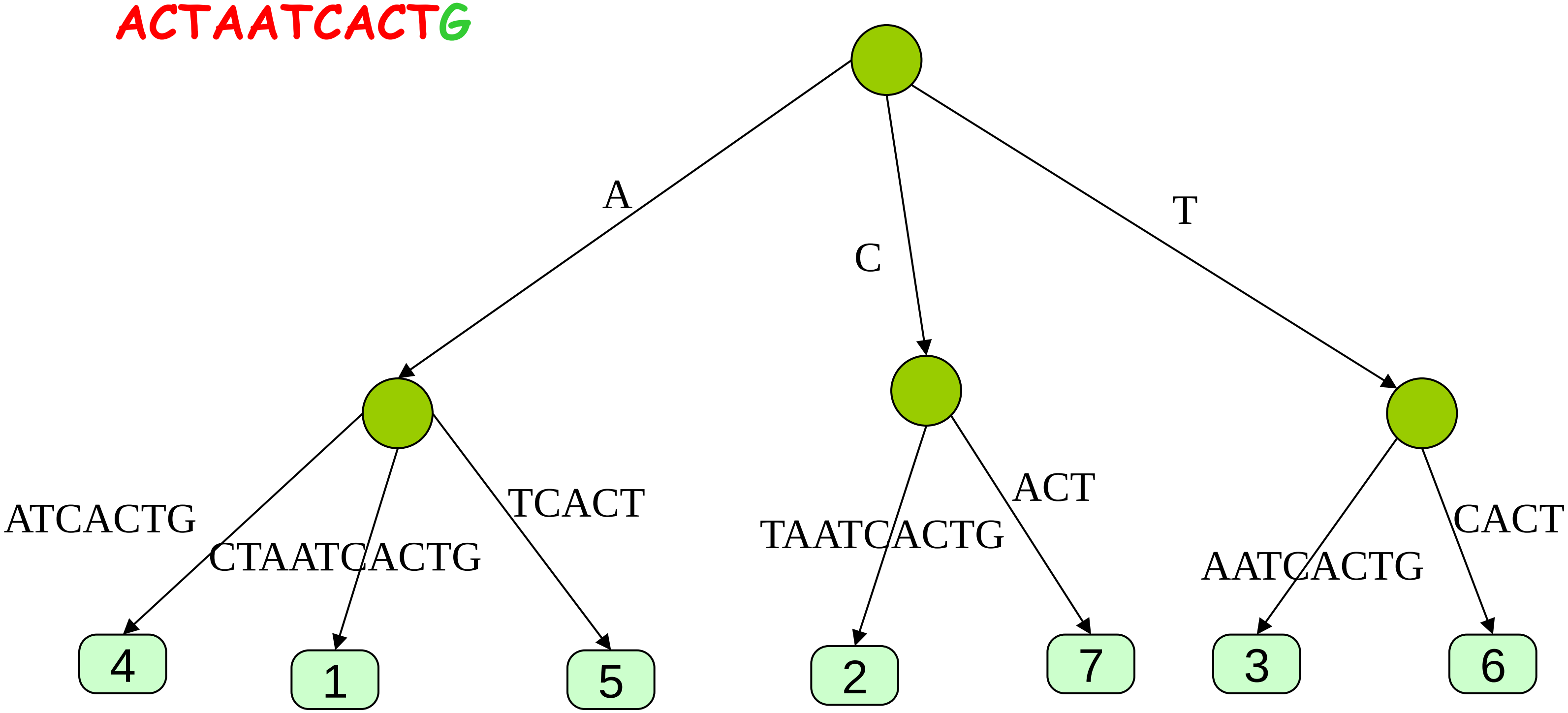
ACTAATCACTG



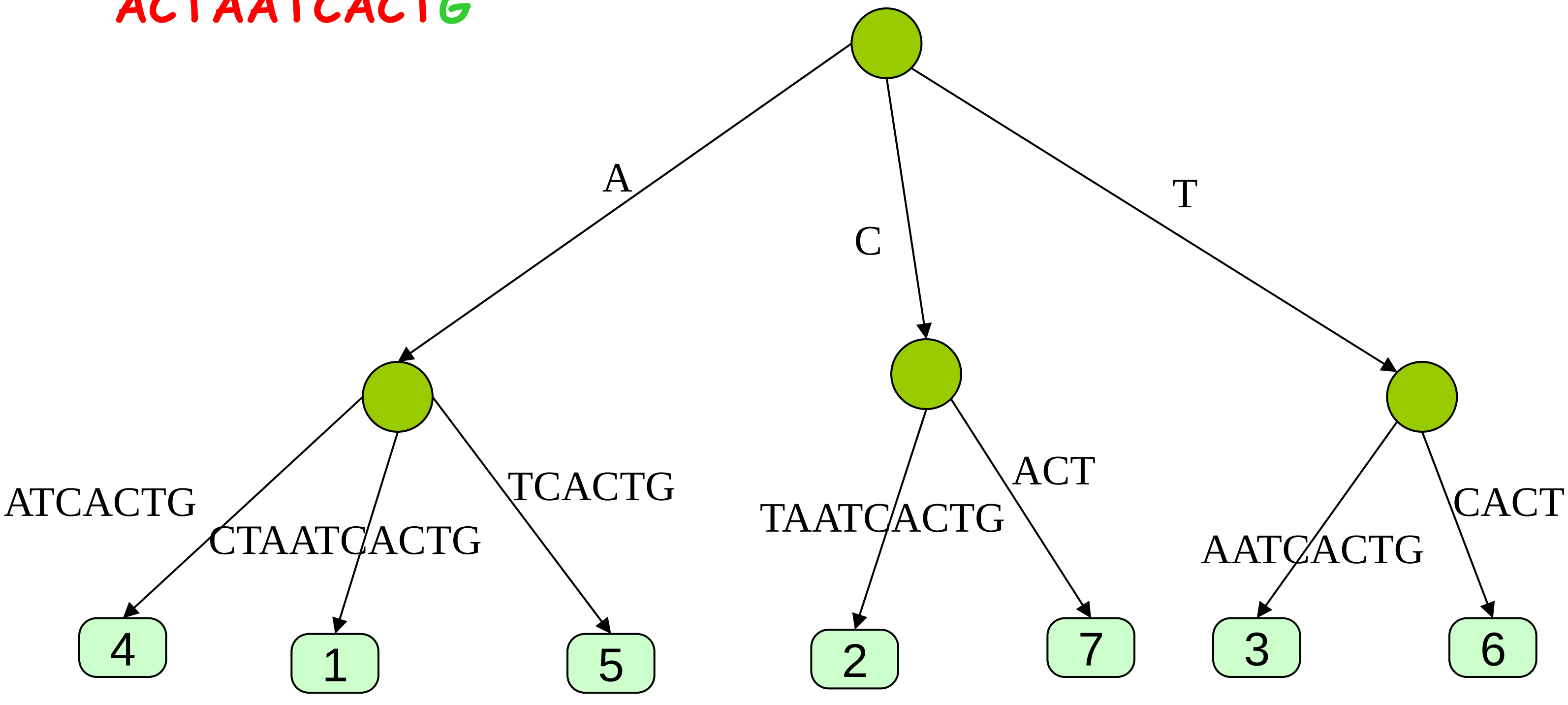
ACTAATCACTG



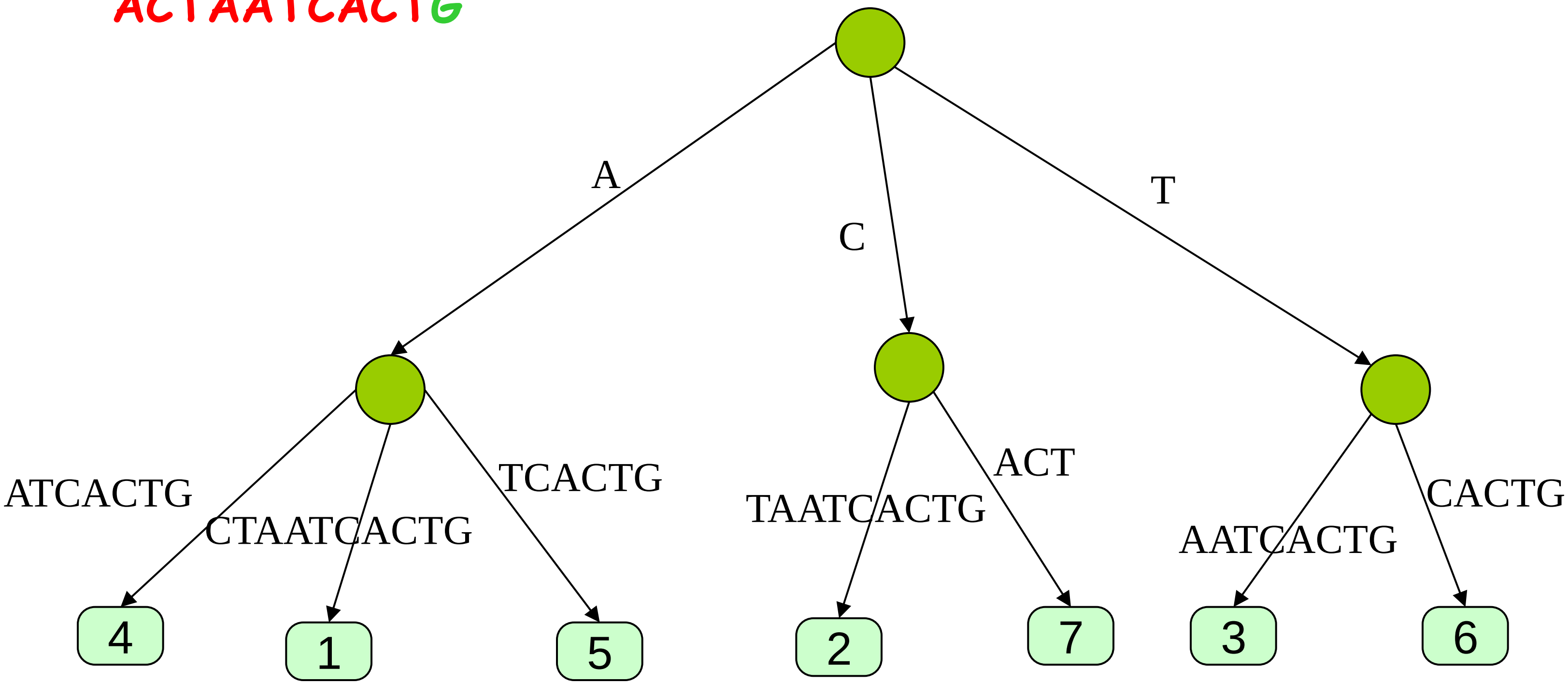
ACTAATCACTG



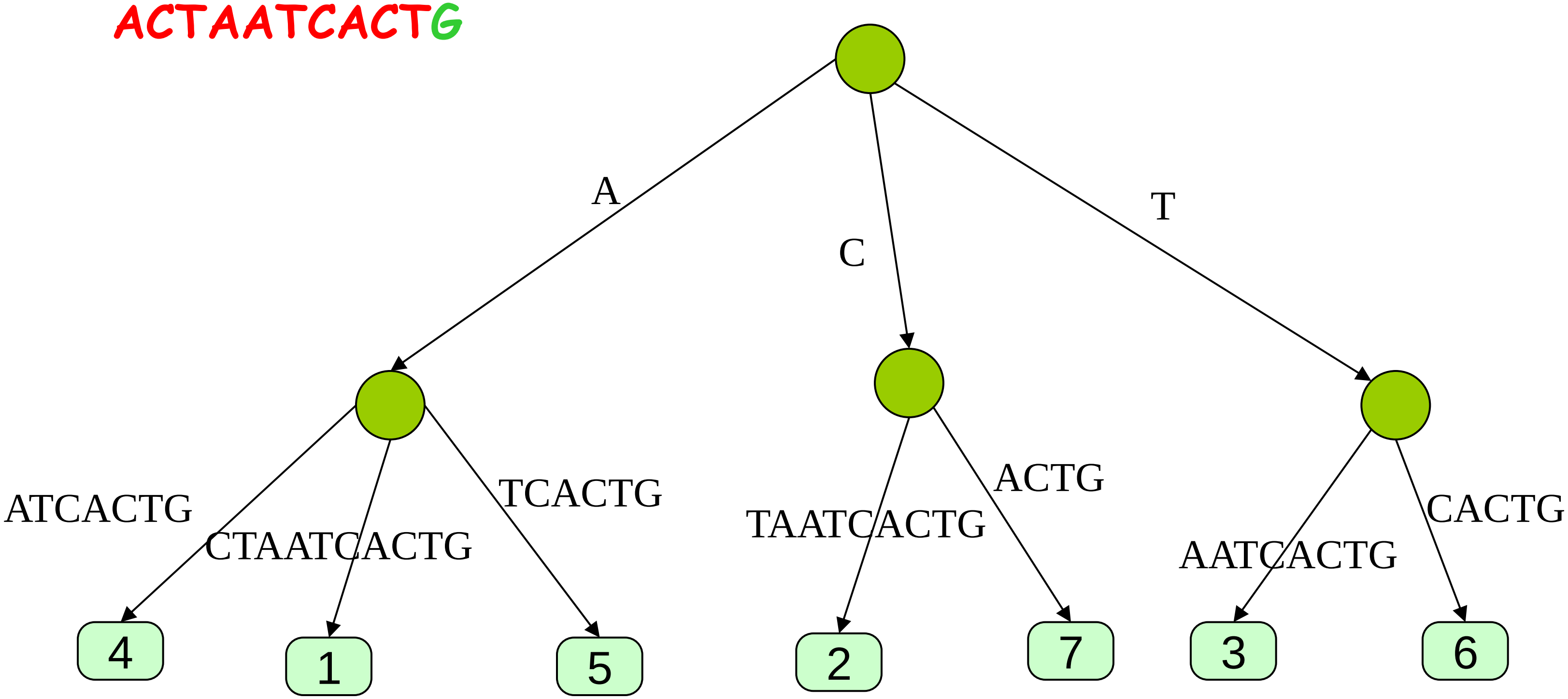
ACTAATCACTG



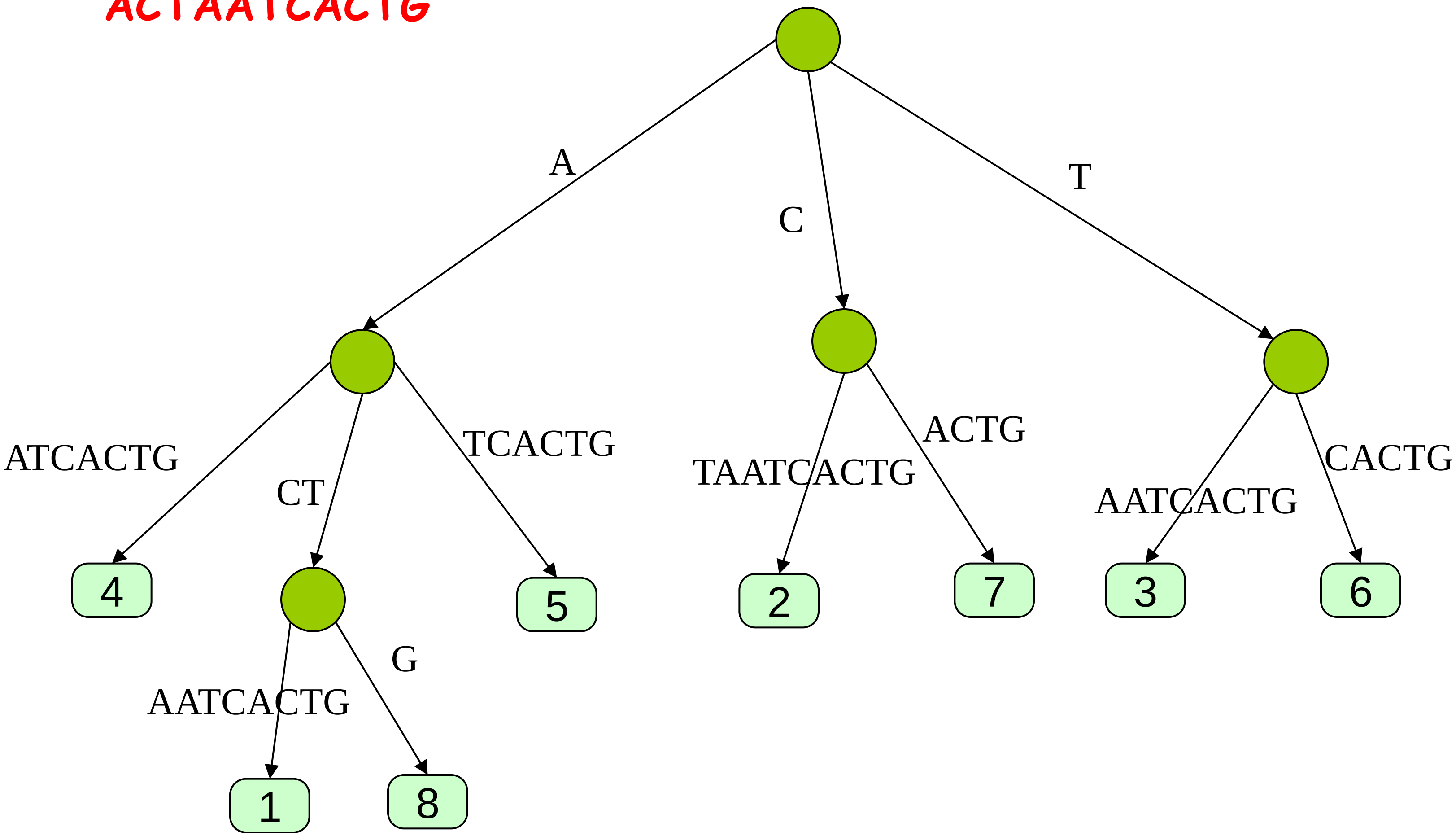
ACTAATCACTG



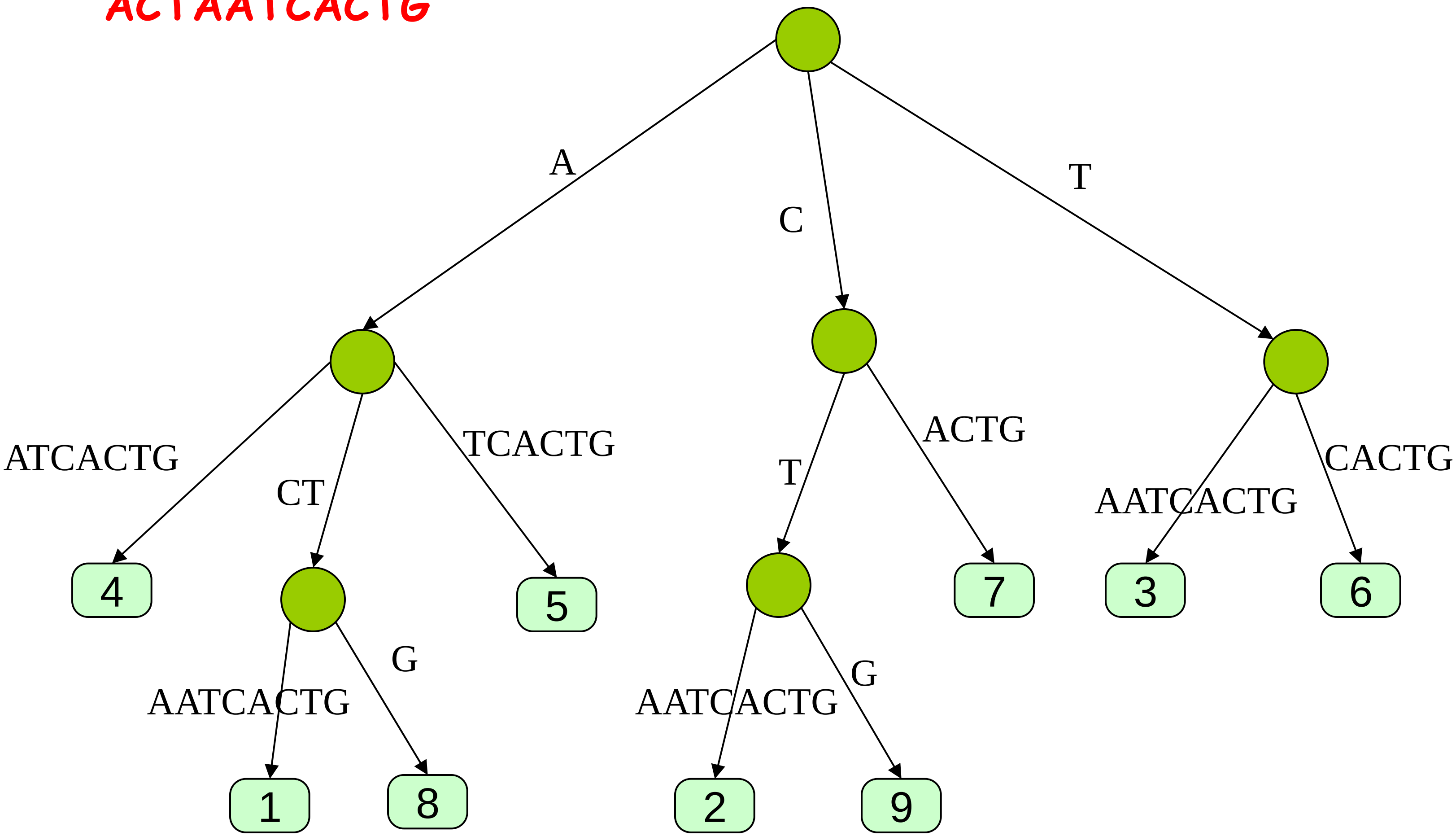
ACTAATCACTG



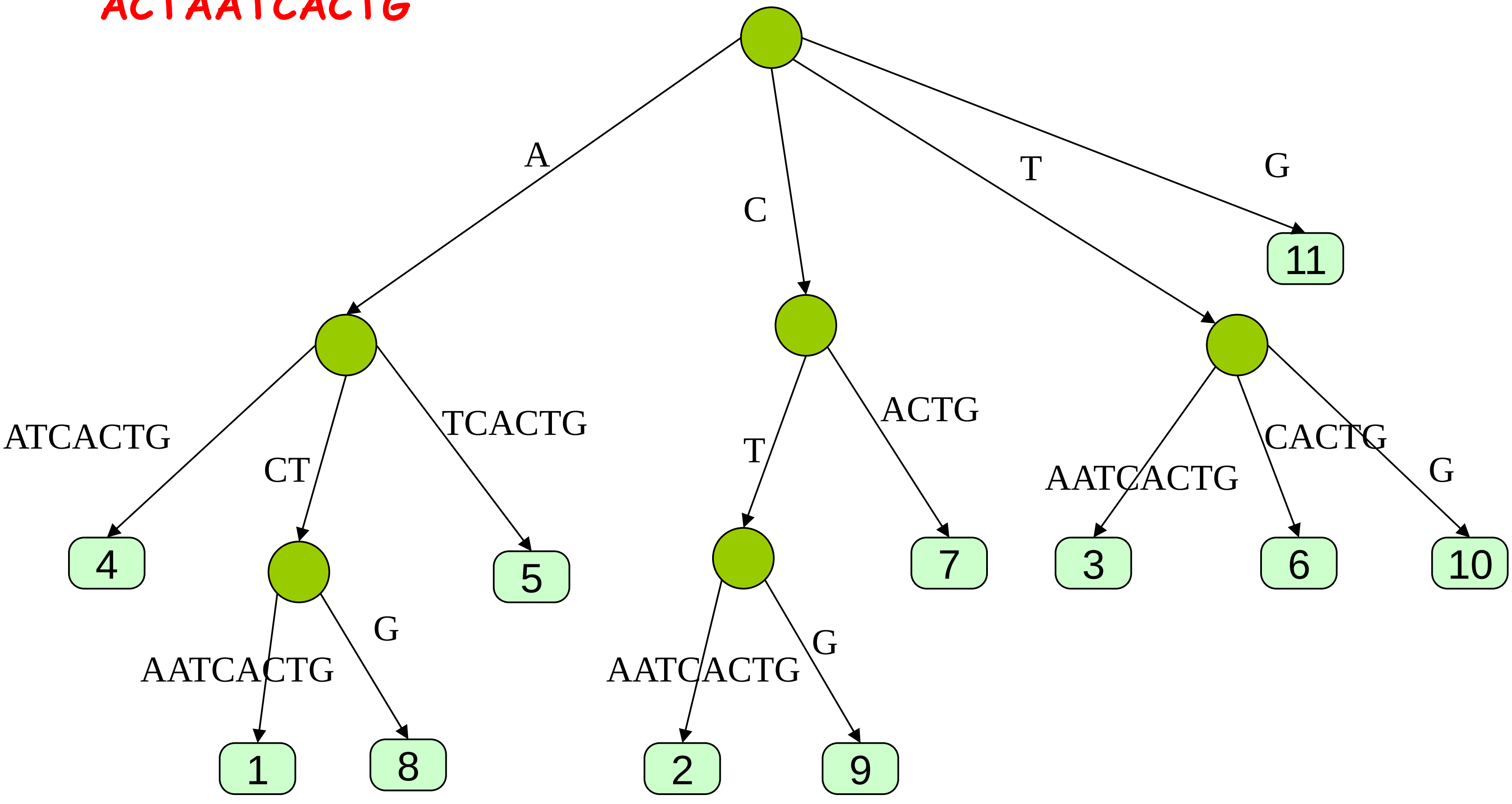
ACTAATCACTG



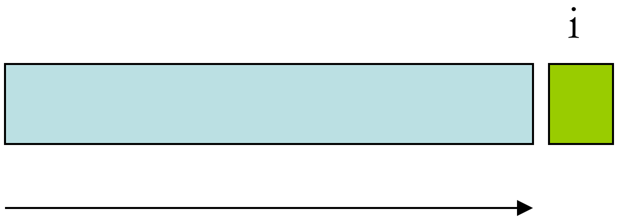
ACTAATCACTG



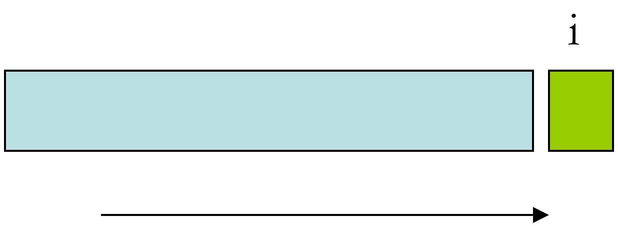
ACTAATCACTG



Observations

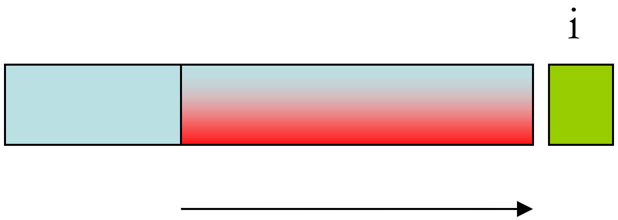


At the first extension we must end at a leaf because no longer suffix exists (rule 1)



At the second extension we still most likely to end at a leaf.

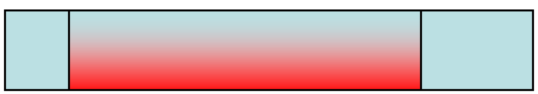
We will not end at a leaf only if the second suffix is a prefix of the first



Say at some extension we do not end at a leaf

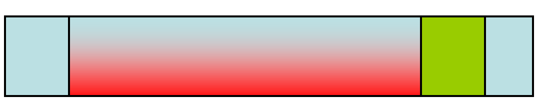
Then this suffix is a prefix of some other suffix (suffixes)

We will not end at a leaf in subsequent extensions

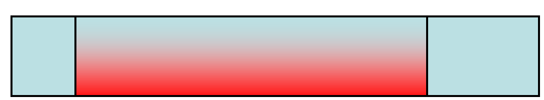


Is there a way to continue using i^{th} character ?

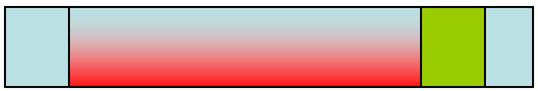
(Is it a prefix of a suffix where the next character is the i^{th} character ?)



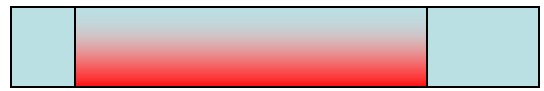
Rule 3



Rule 2



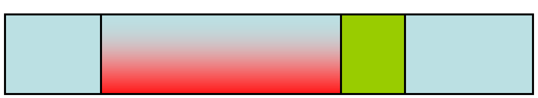
Rule 3



Rule 2

If we apply rule 3 then in all subsequent extensions we will apply rule 3

Otherwise we keep applying rule 2 until in some subsequent extensions we will apply rule 3



Rule 3

In terms of the rules that we apply a phase looks like:

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

We have nothing to do when applying rule 3, so once rule 3 happens we can stop

We don't really do anything significant when we apply rule 1 (the structure of the tree does not change)

Representation

- We do not really store a substring with each edge, but rather pointers into the starting position and ending position of the substring in the text
- With this representaion we do not really have to do anything when rule 1 applies

How do phases relate to each other

1 1 1 1 1 1 1 2 2 2 2 3 3 3 3

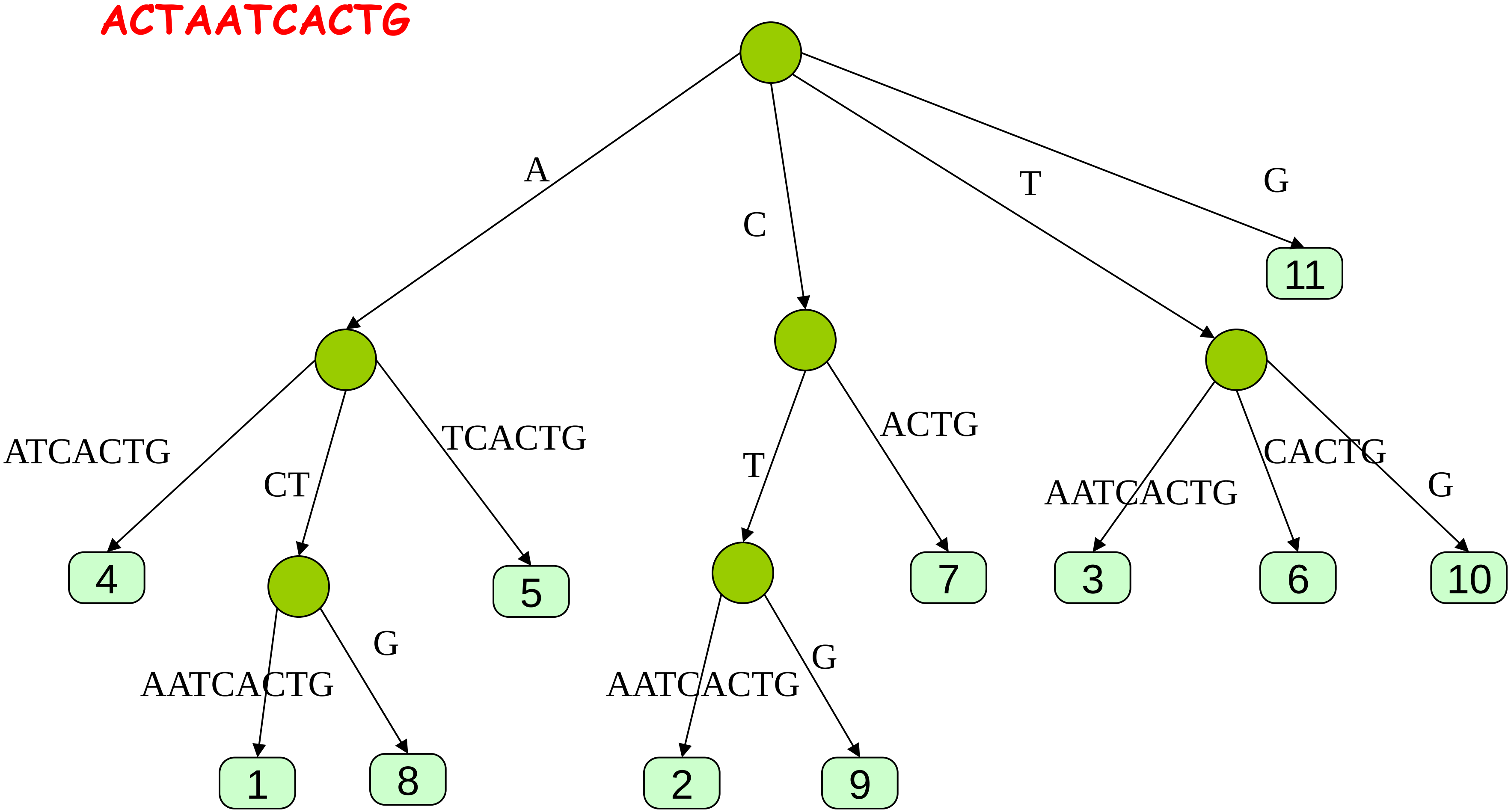
The next phase we must have:

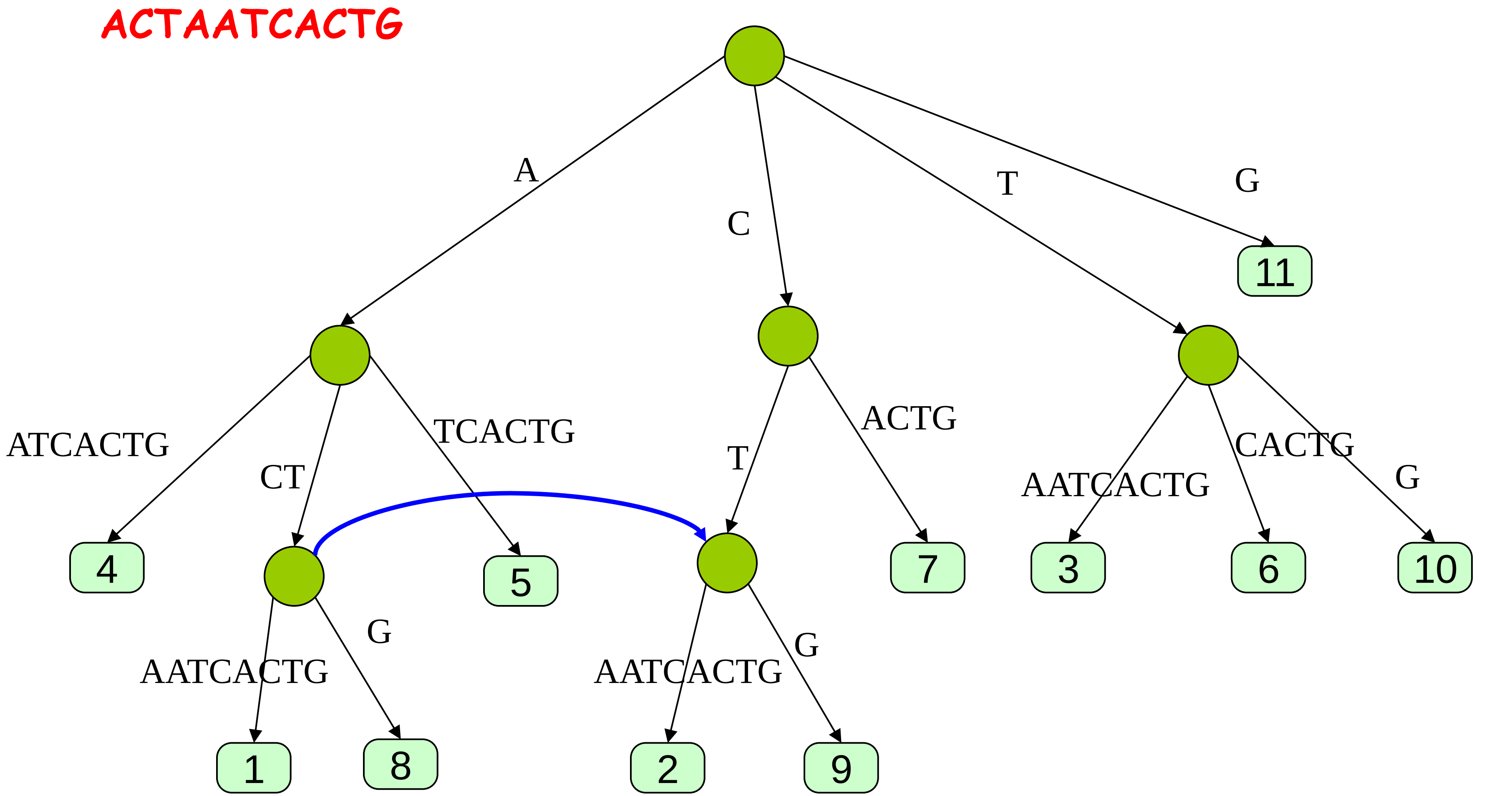
1 1 1 1 1 1 1 1 1 1 1 2/3

So we start the phase with the extension that was the first where we applied rule 3 in the previous phase

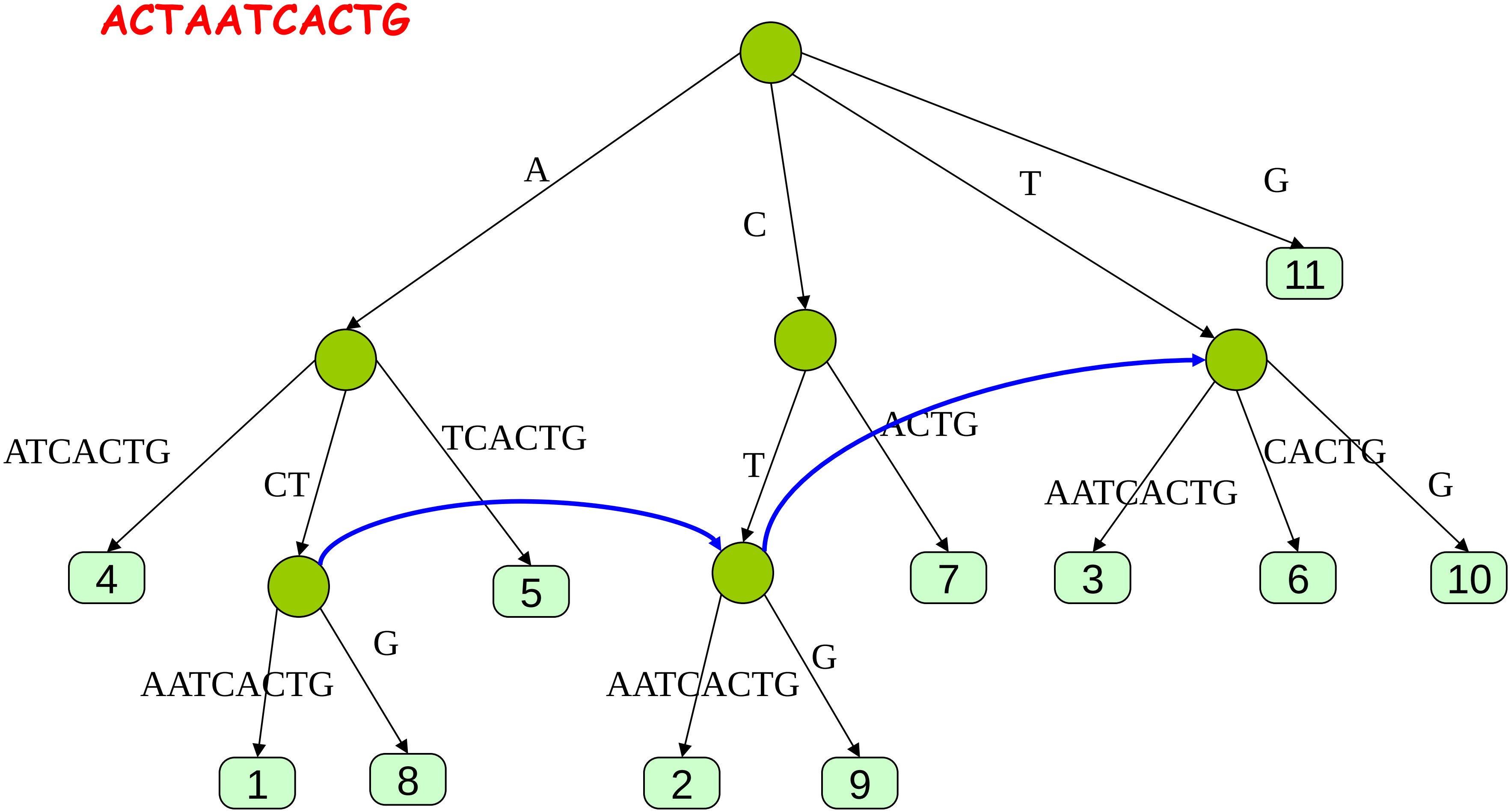
Suffix Links

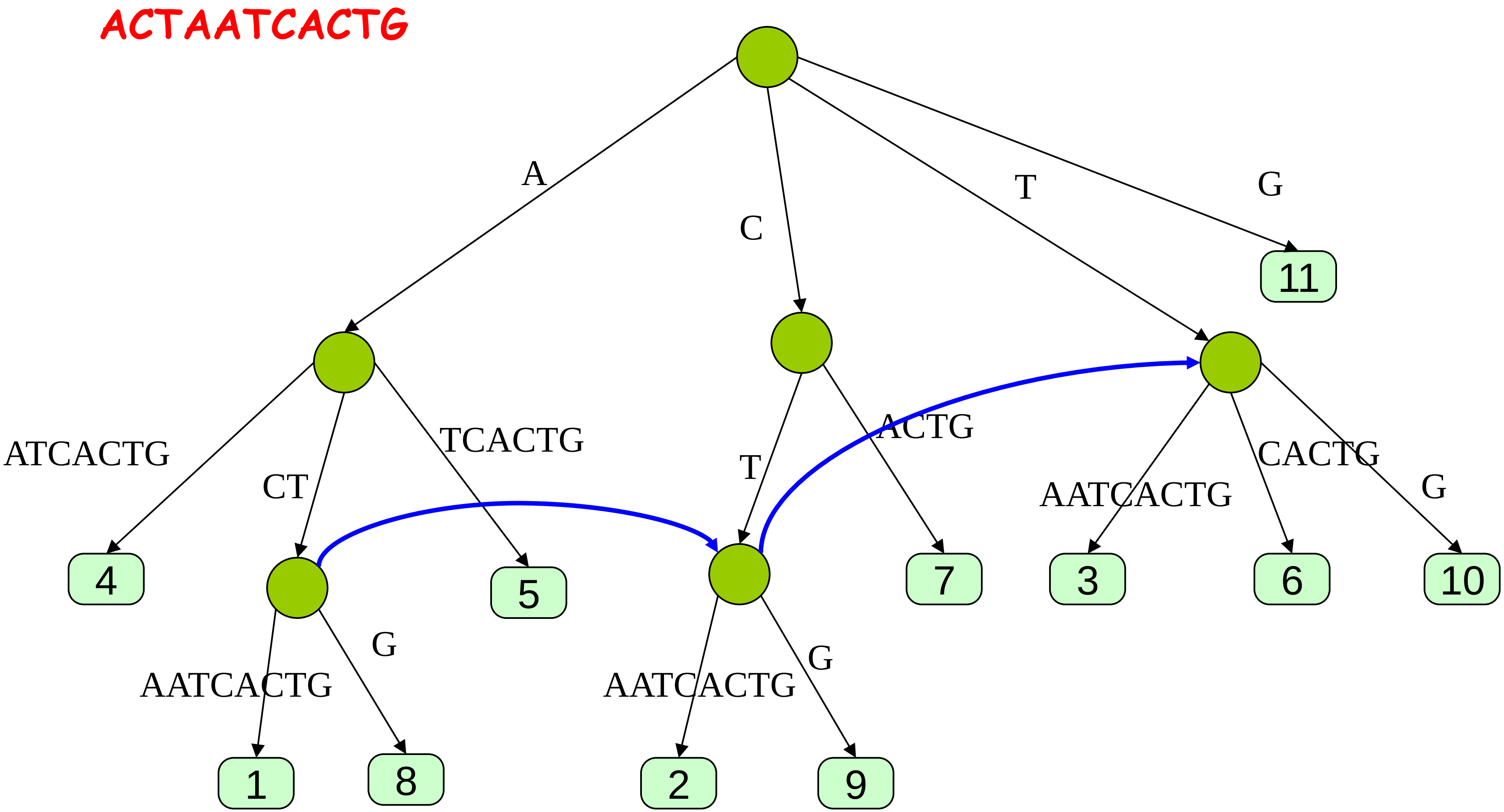
ACTAATCACTG

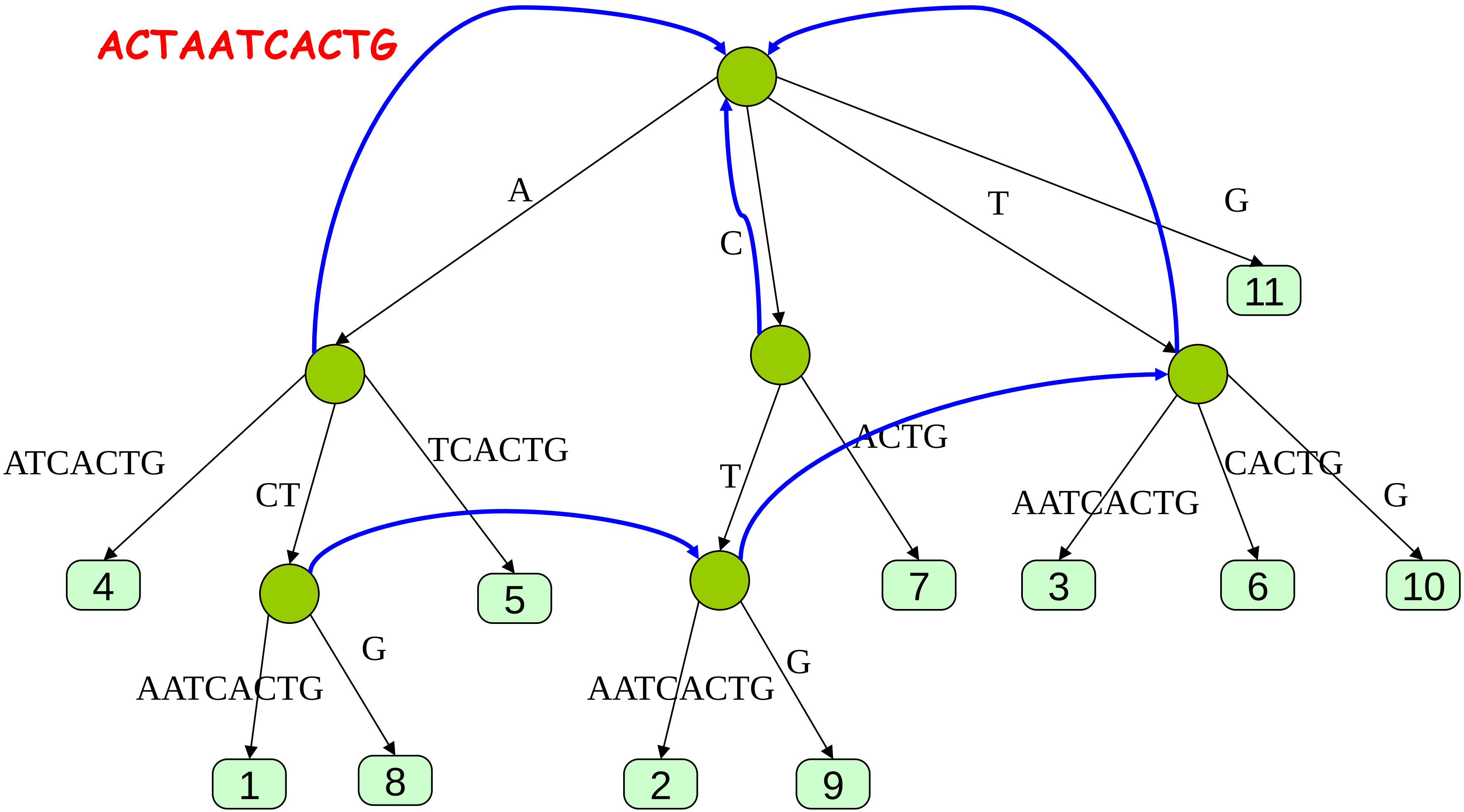



ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
CACTG
AATCACTG
G
4
5
7
3
6
10
G
AATCACTG
G
AATCACTG
1
8
2
9

ACTAATCACTG

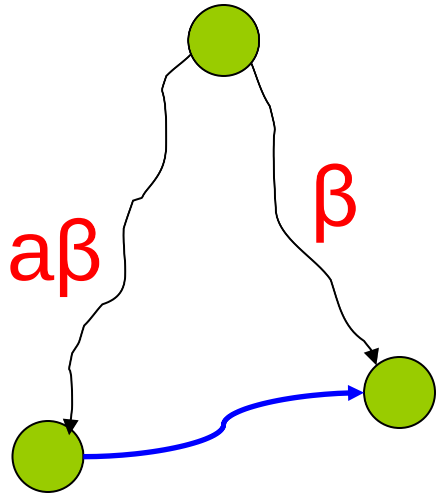



ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
AATCACTG
CACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9


ACTAATCACTG
A
C
T
G
11
ATCACTG
CT
TCACTG
T
ACTG
CACTG
AATCACTG
G
4
5
7
3
6
10
G
AATCACTG
AATCACTG
G
1
8
2
9

Suffix Links

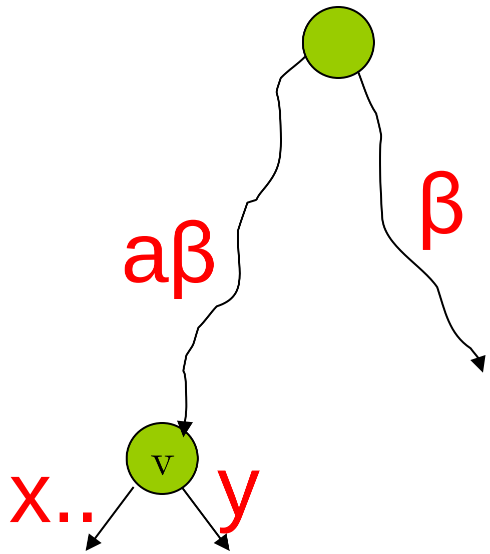
- From an internal node that corresponds to the string aβ to the internal node that corresponds to β (if there is such node)



- Is there such a node ?

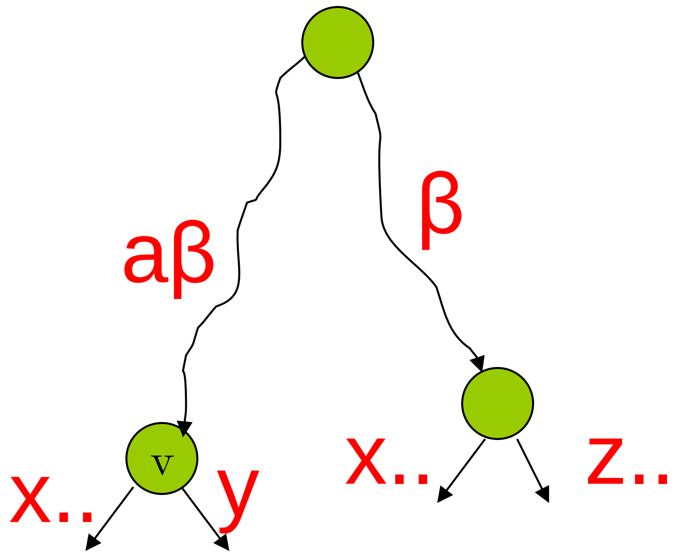
Suppose we create v applying rule 2. Then there was a suffix αβx… and now we add αβy

So there was a suffix βx…



- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy



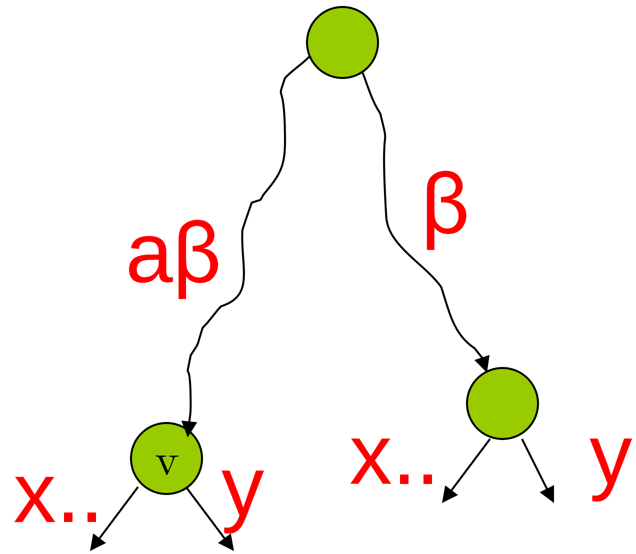
So there was a suffix βx…

If there was also a suffix βz…

Then a node corresponding to β is there

- Is there such a node ?

Suppose we create v applying rule 2. Then there was a suffix aβx… and now we add aβy



So there was a suffix βx…

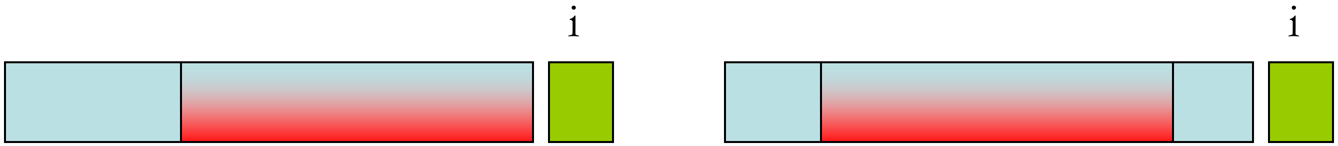
If there was also a suffix βz…

Then a node corresponding to β is there

Otherwise it will be created in the next extension when we add βy

Inv: All suffix links are there except (possibly) of the last internal node added

You are at the (internal) node corresponding to the last extension



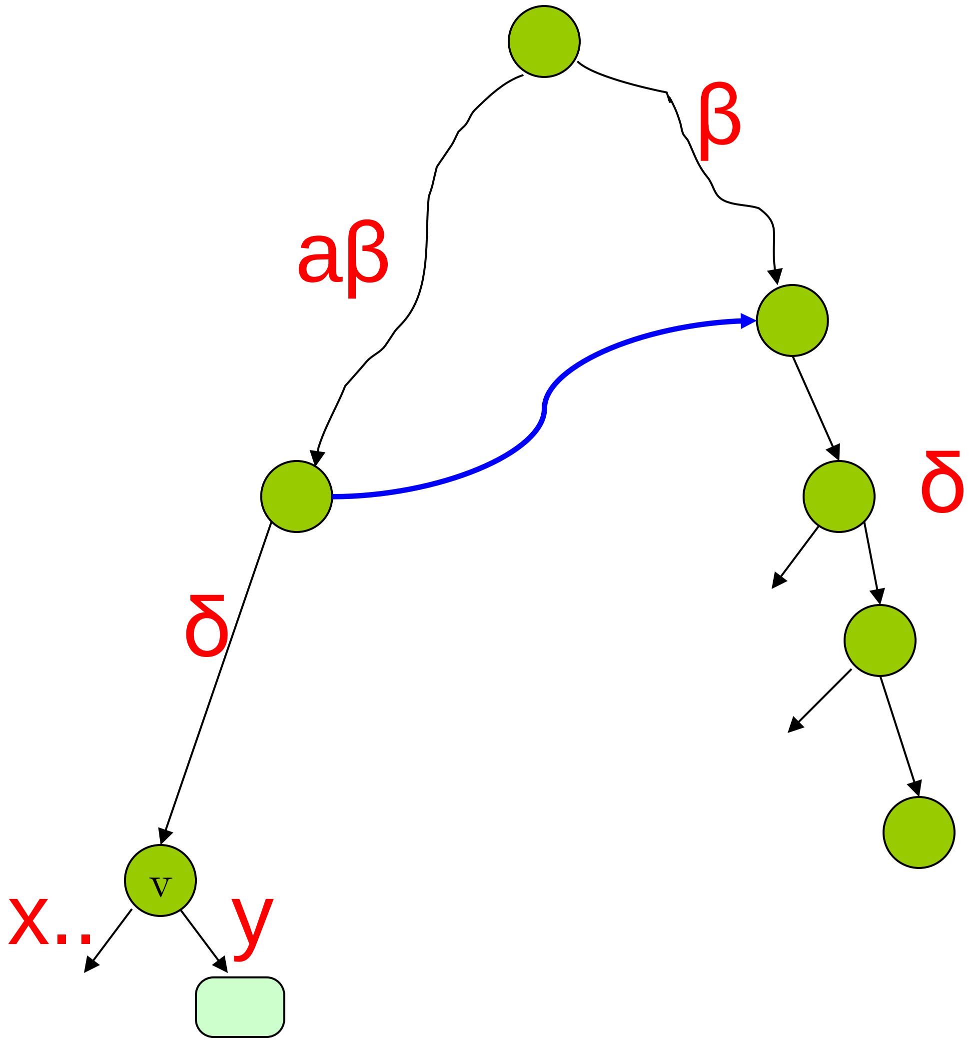
Remember: we apply rule 2

You start a phase at the last internal node of the first extension in which you applied rule 3 in the previous iteration

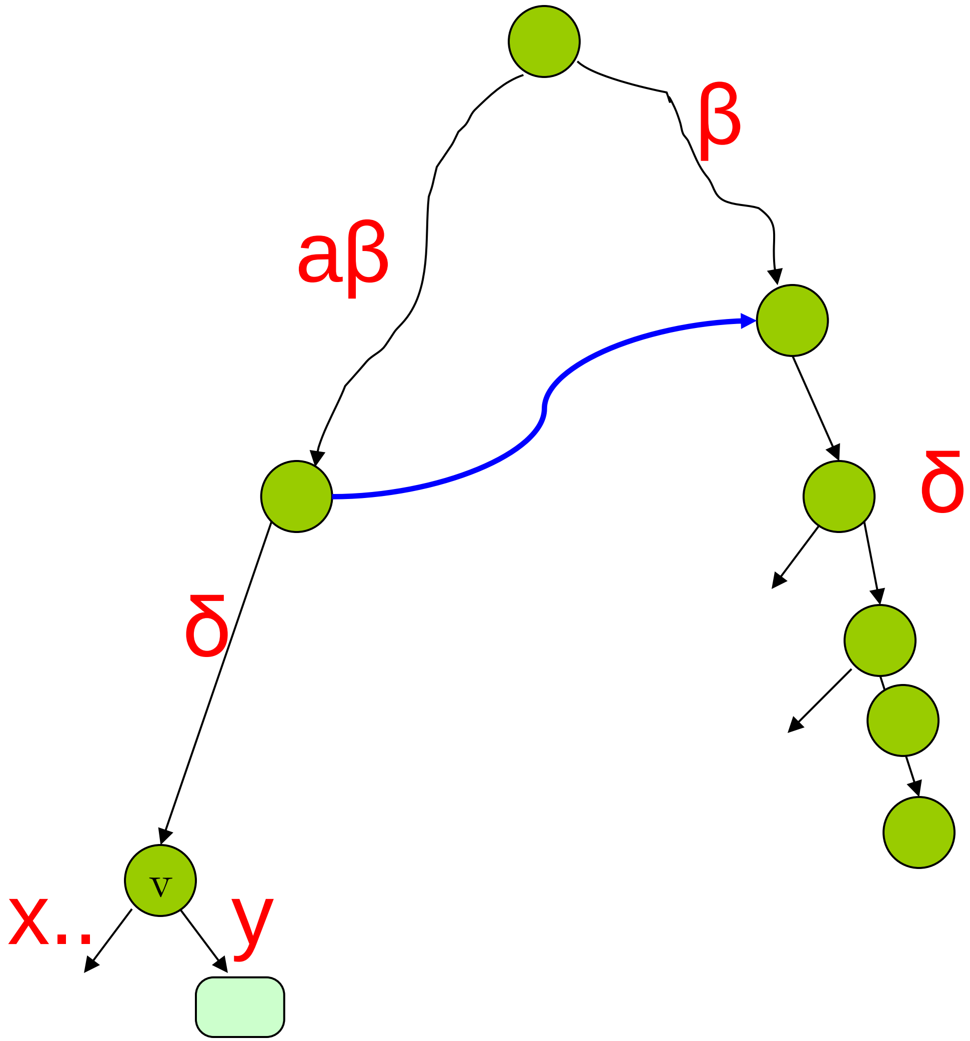
1) Go up one node (if needed) to find a suffix link

2) Traverse the suffix link

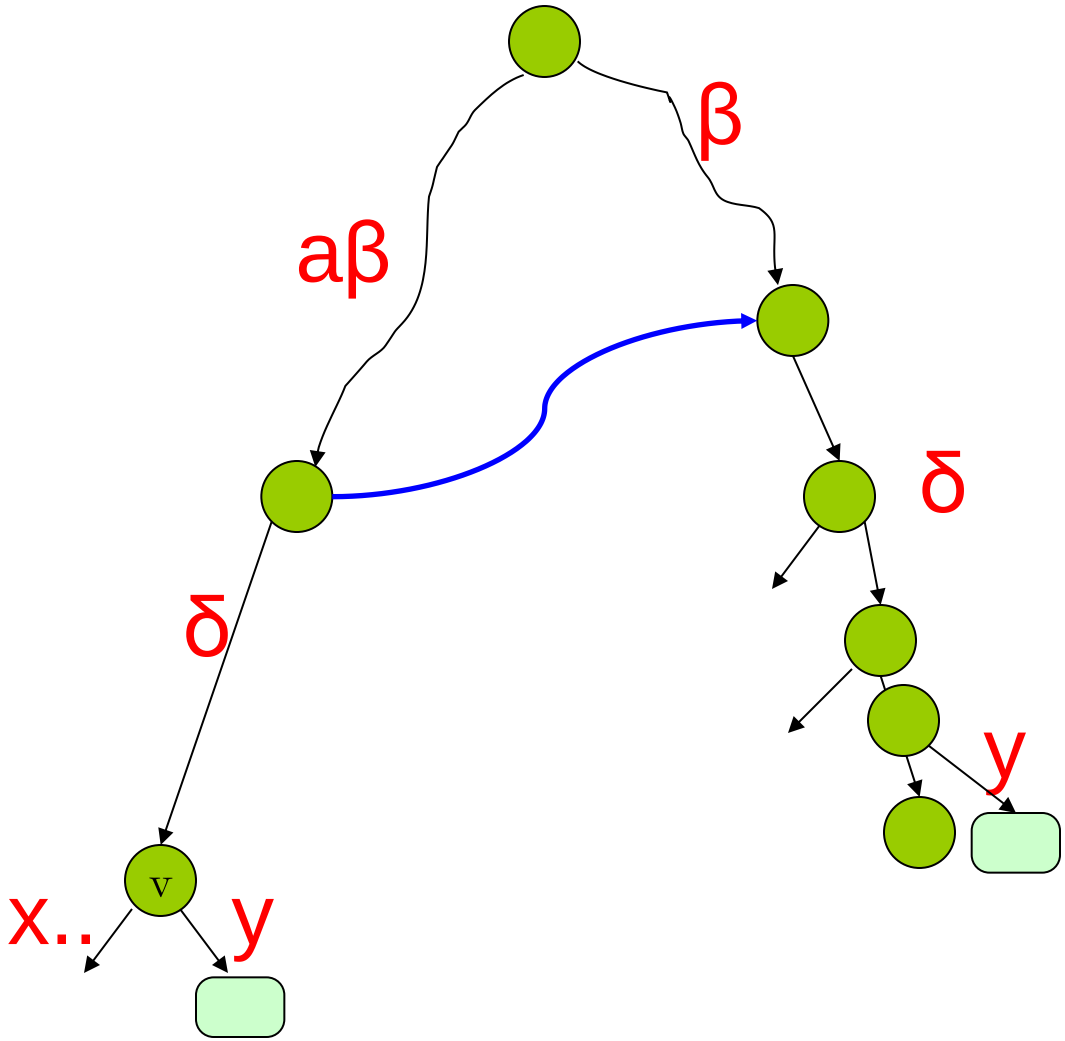
3) If you went up in step 1 along an edge that was labeled δ then go down consuming a string δ



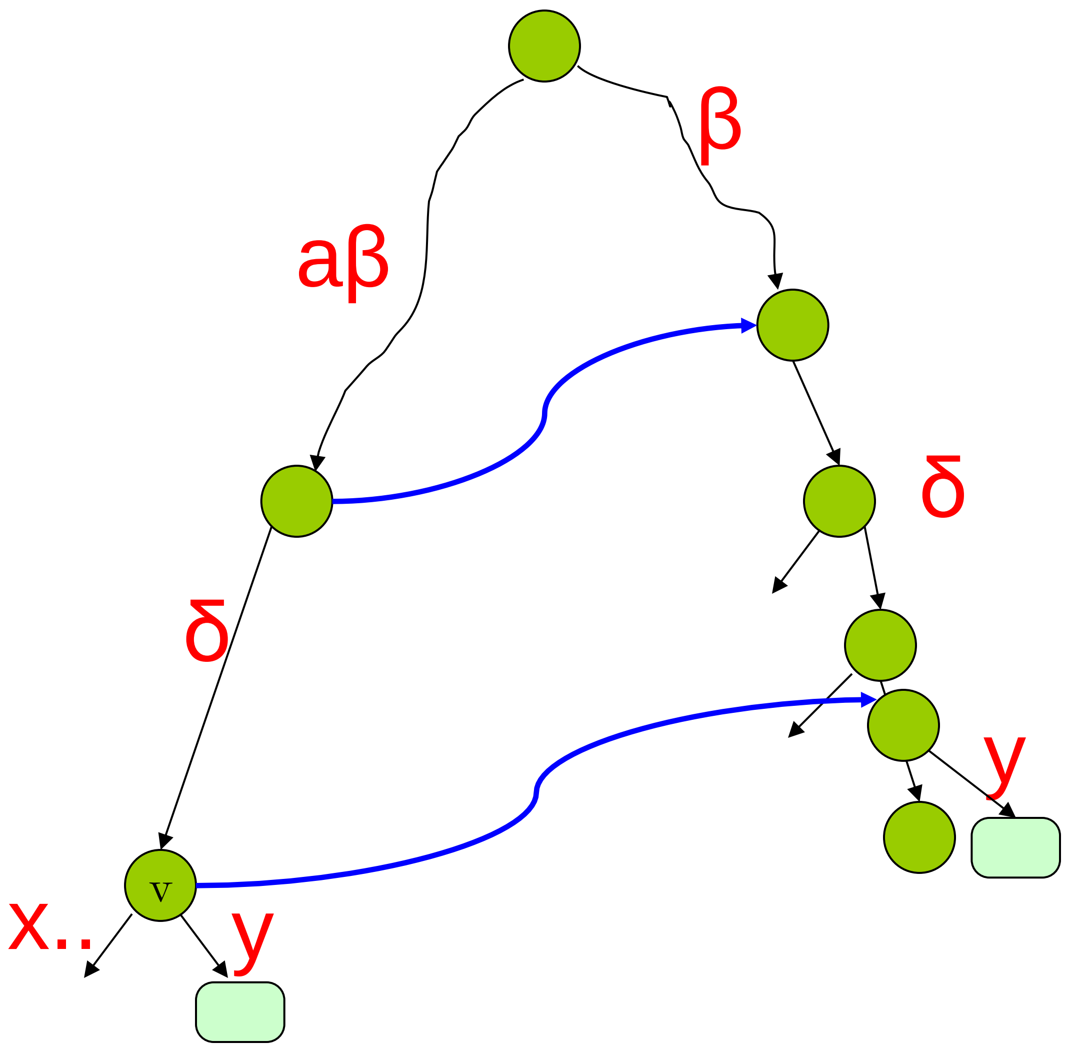
Create the new internal node if necessary



Create the new internal node if necessary



Create the new internal node if necessary, add the suffix



Create the new internal node if necessary, add the suffix and install a suffix link if necessary

Analysis

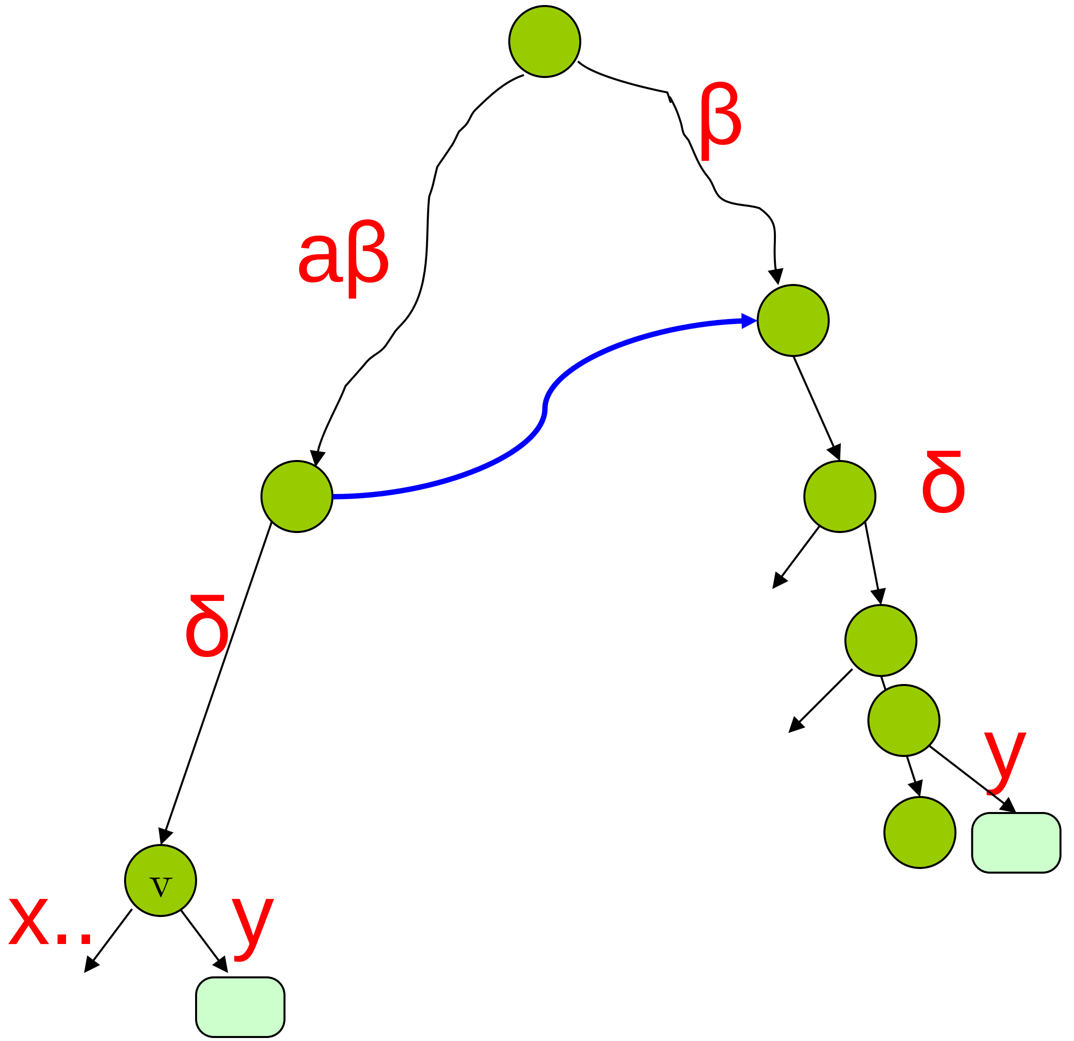
Handling all extensions of rule 1 and all extensions of rule 3 per phase take O(1) time ➔ O(n) total

How many times do we carry out rule 2 in all phases ?

O(n)

Does each application of rule 2 takes constant time ?

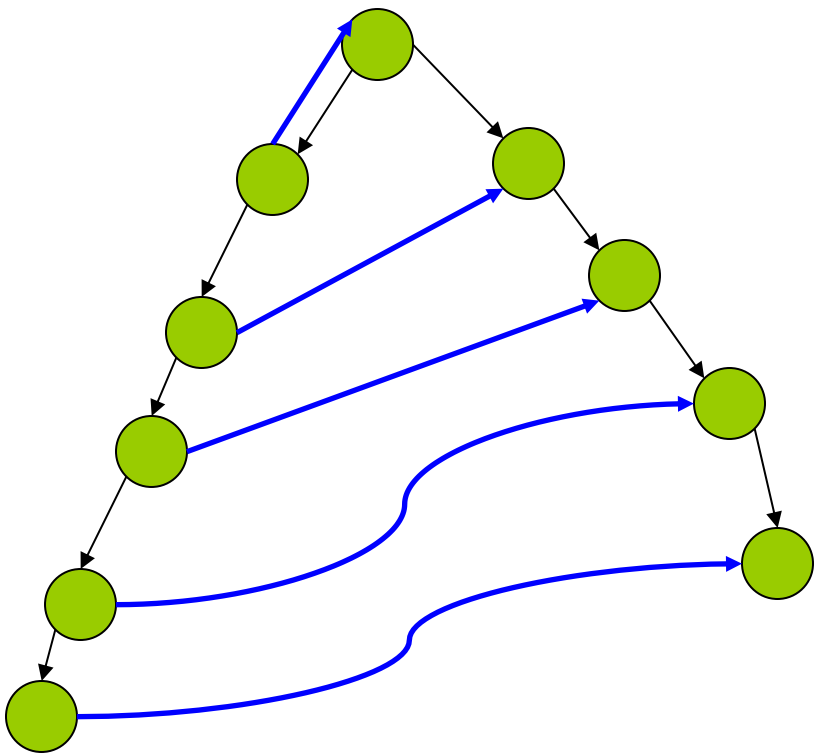
No ! (going up and traversing the suffix link takes constant time, but then we go down possibly on many edges..)



β
aβ
δ
δ
y
v
x..
y

So why is it a linear time algorithm ?

How much can the depth change when we traverse a suffix link ?



It can decrease by at most 1

Punch line

Each time we go up or traverse a suffix link the depth decreases by at most 1

When starting the depth is 0, final depth is at most n

So during all applications of rule 2 together we cannot go down more than 3n times

THM: The running time of Ukkonen's algorithm is O(n)

Drawbacks of suffix trees

- Suffix trees consume a lot of space
- It is $O(n)$ but the constant is quite big
- Notice that if we indeed want to traverse an edge in $O(1)$ time then we need an array of ptrs. of size $|\Sigma|$ in each node

Suffix arrays

Suffix array

- We loose some of the functionality but we save space.

Let s = abab

Sort the suffixes lexicographically:
ab, abab, b, bab

The suffix array gives the indices of the suffixes in sorted order

2	0	3	1

How do we build it ?

- Build a suffix tree
- Traverse the tree in DFS, lexicographically picking edges outgoing from each node and fill the suffix array.
- O(n) time

How do we search for a pattern ?

- If P occurs in T then all its occurrences are consecutive in the suffix array.
- Do a binary search on the suffix array
- Takes O(mlogn) time

Example

Let S = mississippi

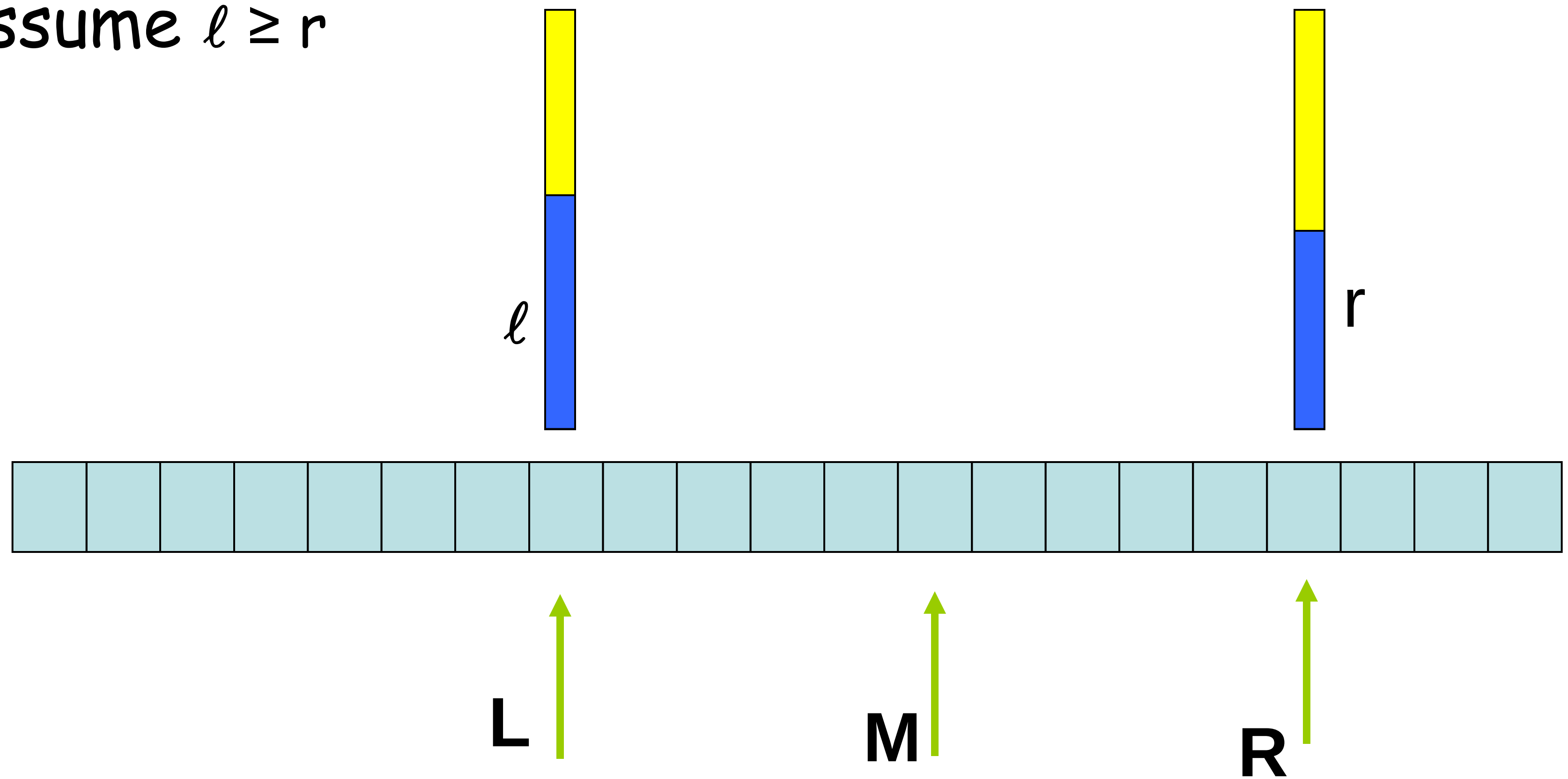
Let P = issa

L →	10	i
	7	ippi
	4	issippi
	1	ississippi
	0	mississippi
M →	9	pi
	8	ppi
	6	sippi
	3	sisippi
	5	ssippi
R →	2	ssissippi

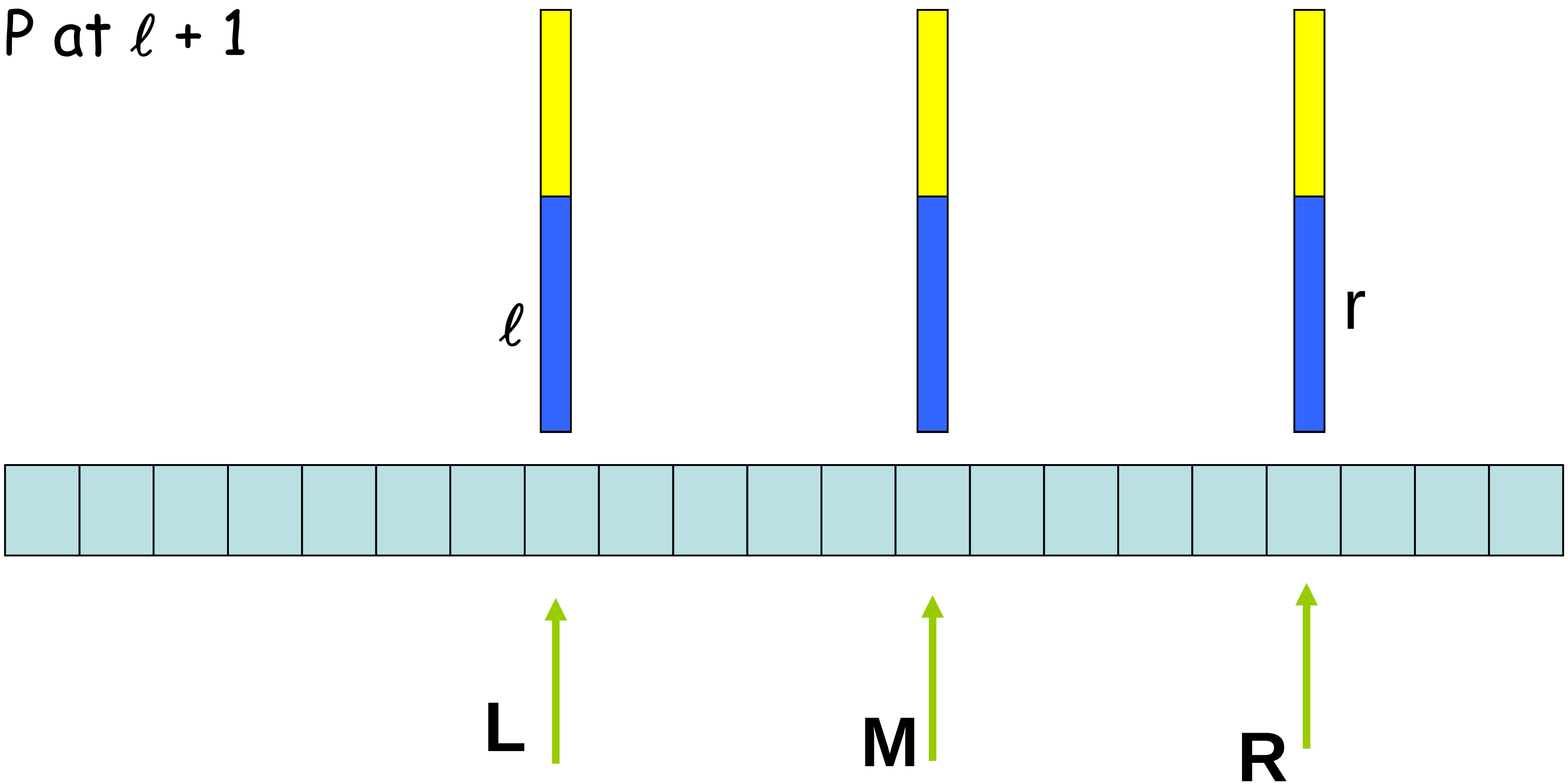
How do we accelerate the search ?

Maintain ℓ = LCP(P,L)
Maintain r = LCP(P,R)

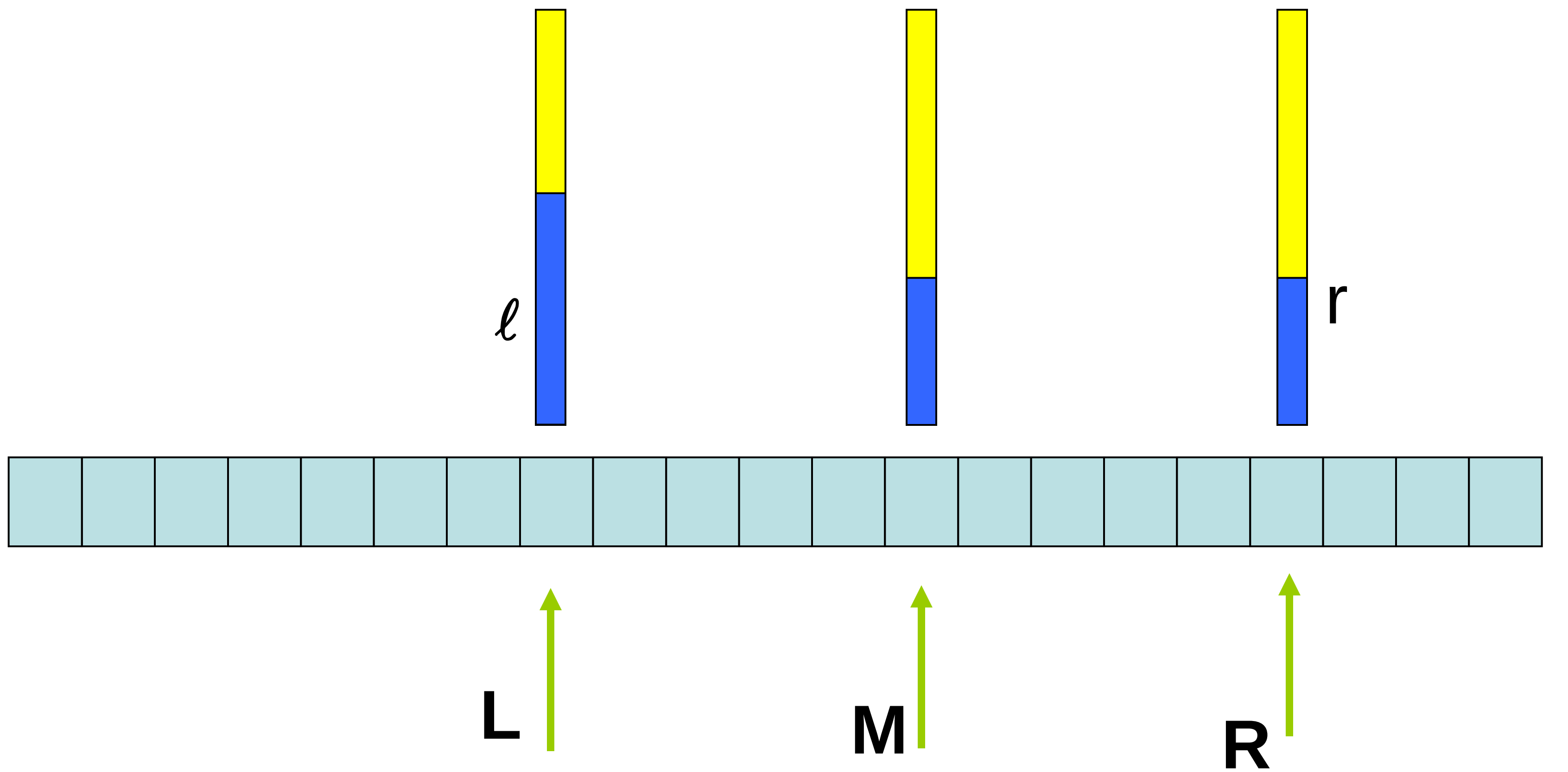
Assume $\ell \geq r$



If $\ell = r$ then
start comparing M
to P at $\ell + 1$

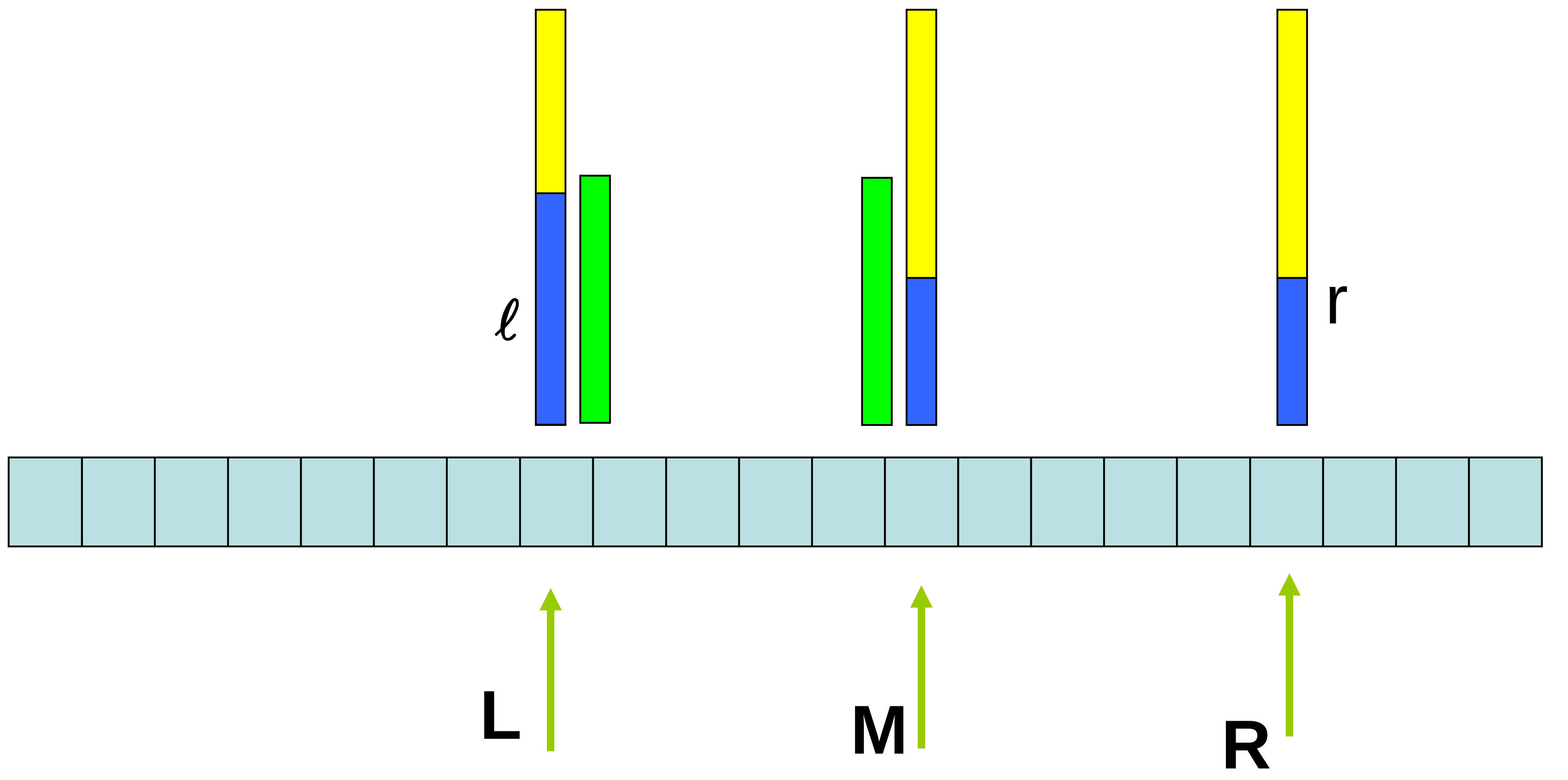


$\ell > r$



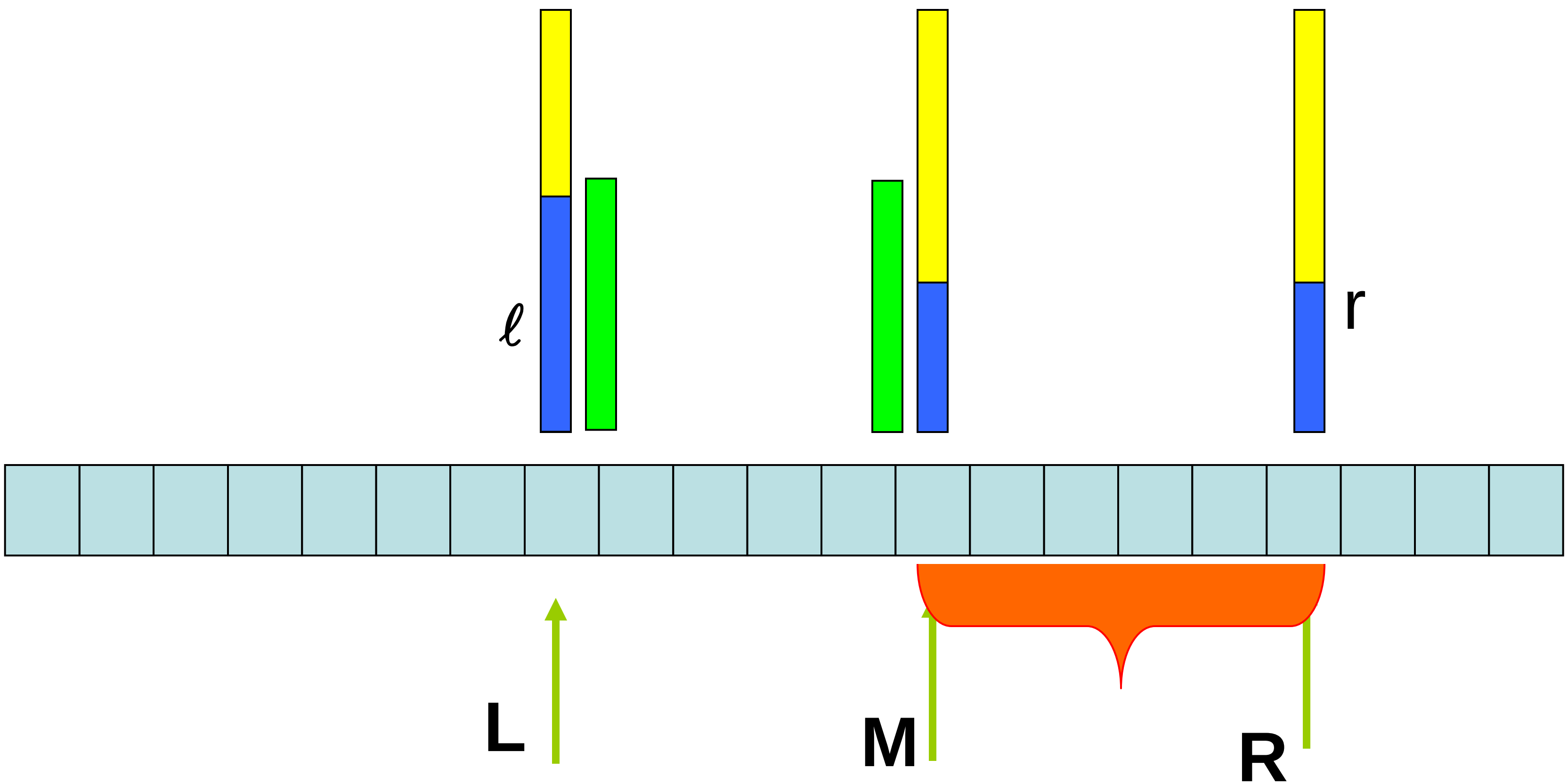
Someone whispers LCP(L,M)

LCP(L,M) > ℓ

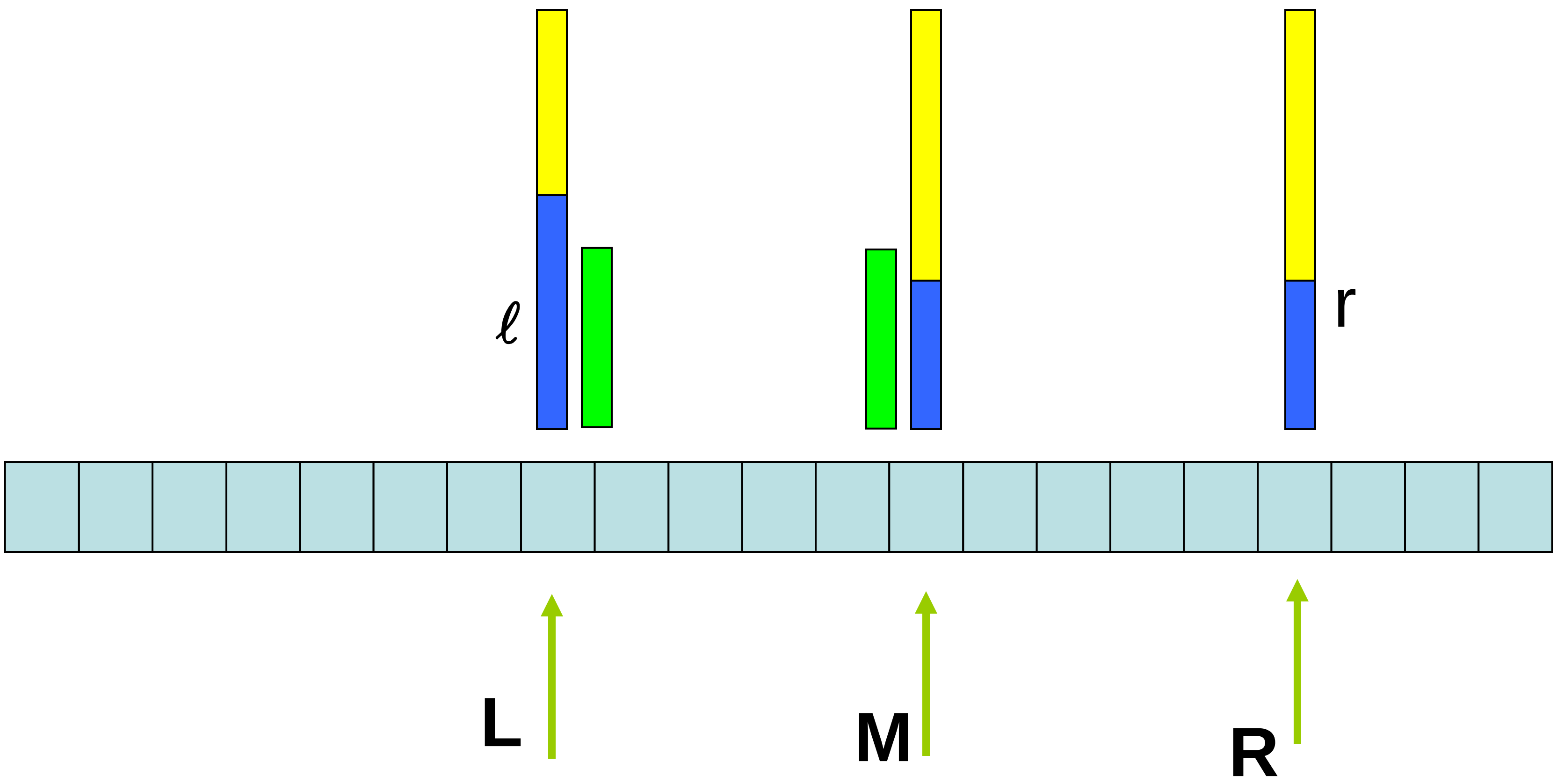


Continue in the right half

LCP(L,M) > ℓ

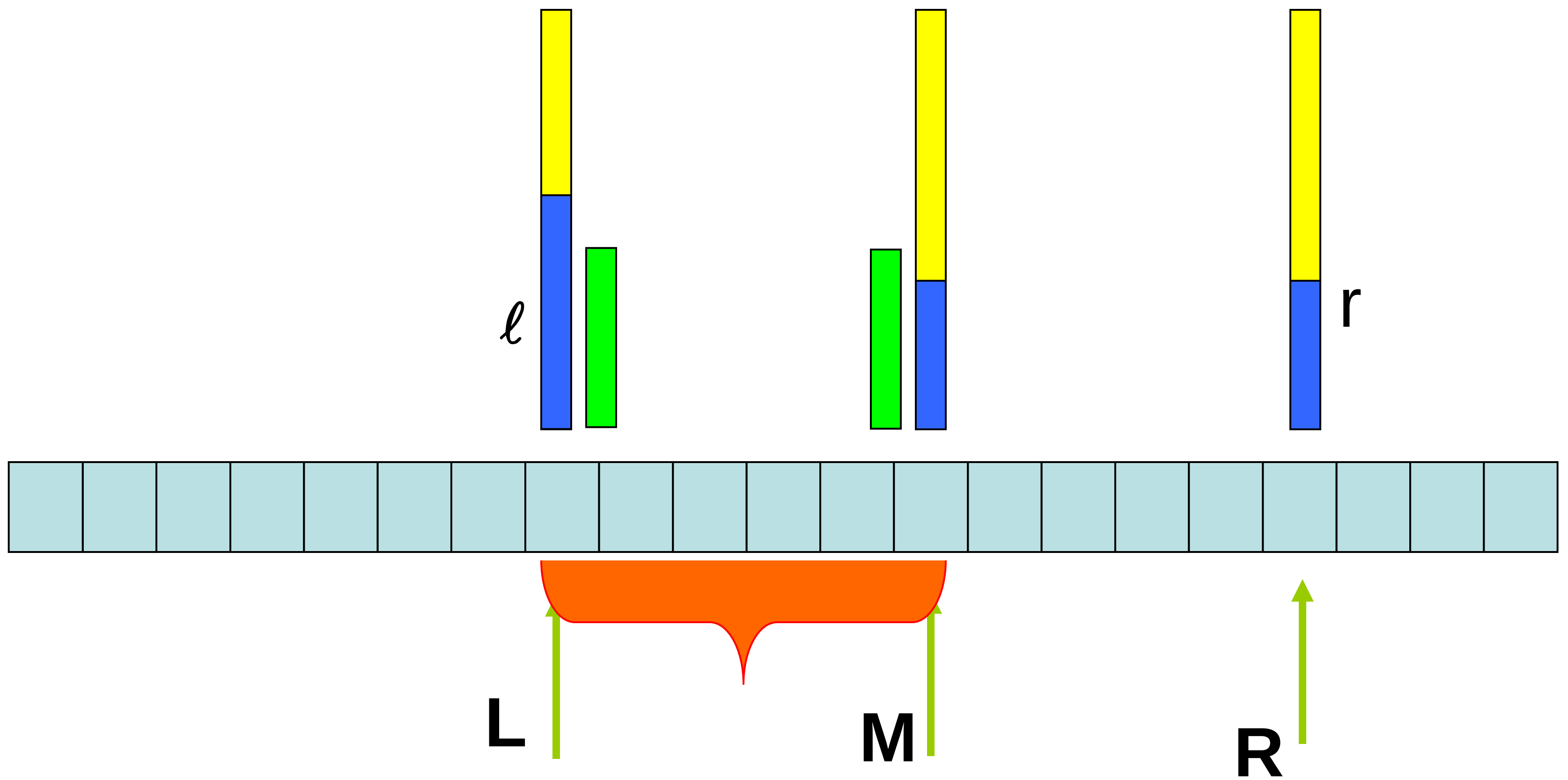


$LCP(L,M) < \ell$



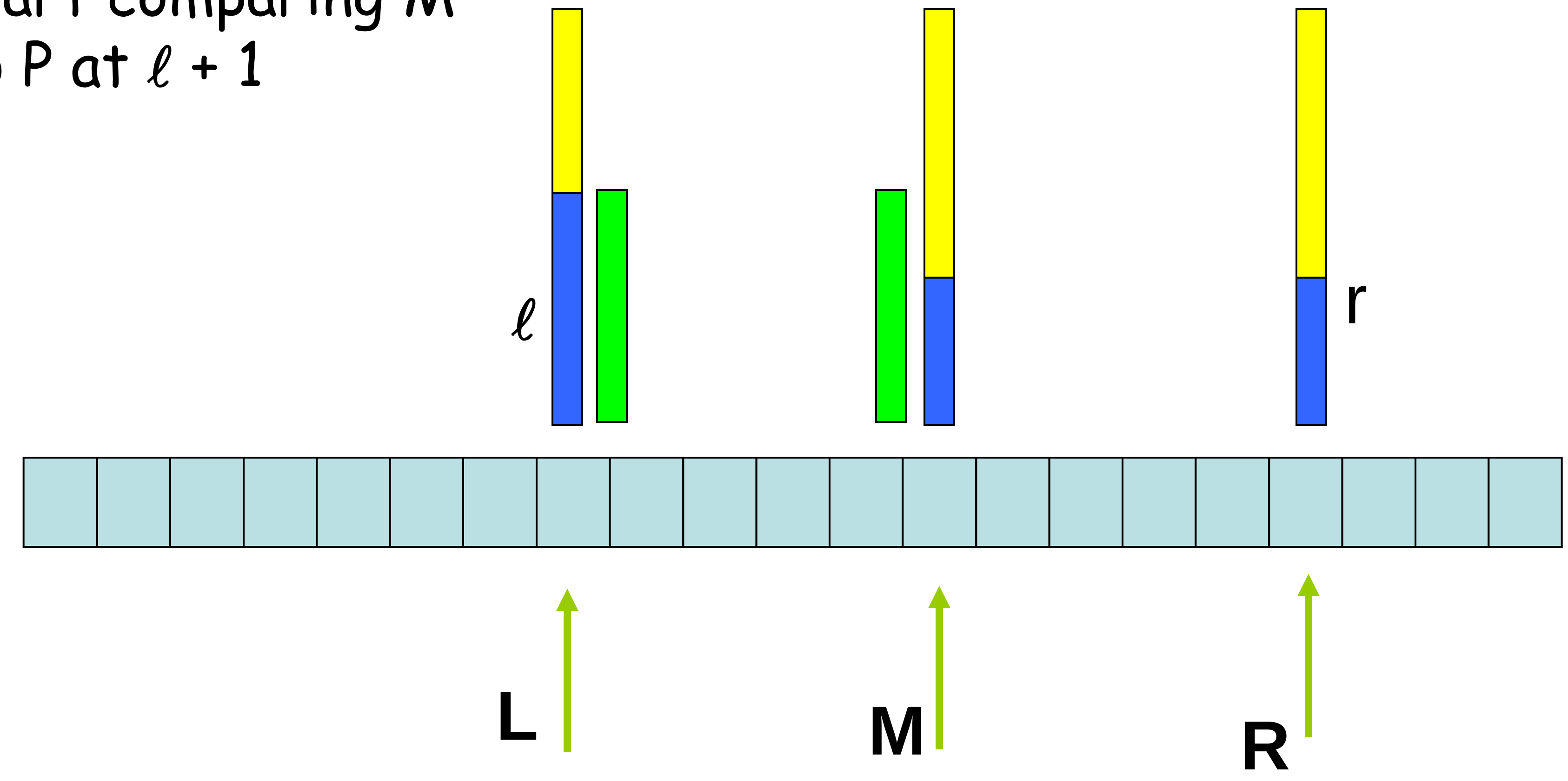
Continue in the left half

LCP(L,M) < ℓ



LCP(L,M) = ℓ

start comparing M to P at ℓ + 1



Analysis

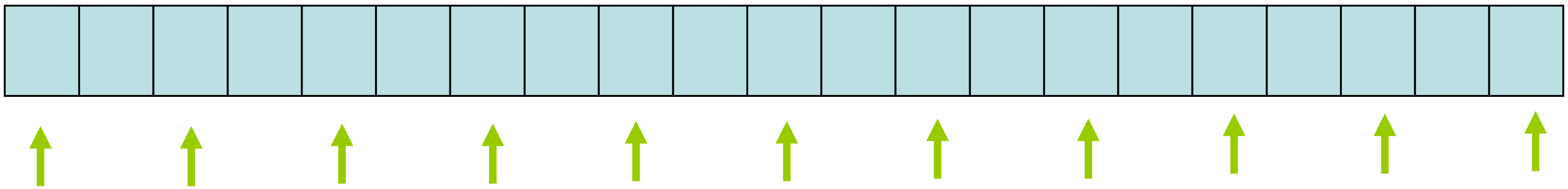
If we do more than a single comparison in an iteration then $\max(\ell, r)$ grows by 1 for each comparison ➔ $O(m + \log n)$ time

Construct the suffix array without the suffix tree

Linear time construction

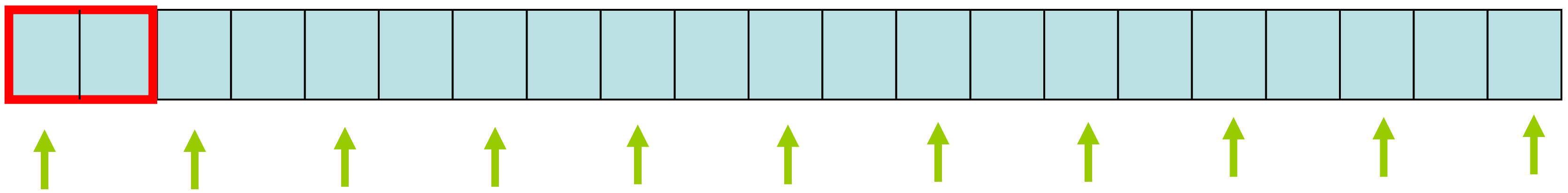
Recursively ?

Say we want to sort only suffixes that start at even positions ?



Change the alphabet

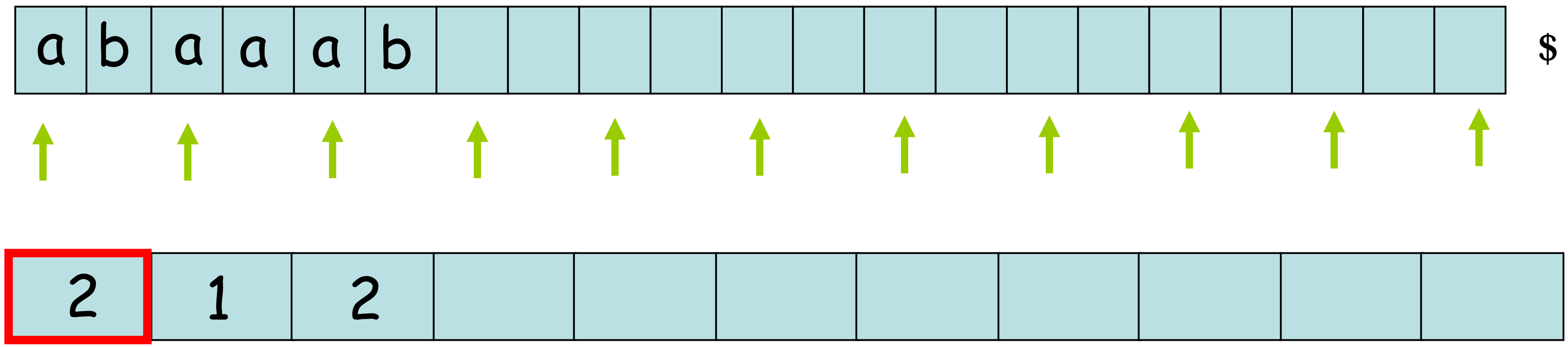
Every pair of characters is now a character



You in fact sort suffixes of a string shorter by a factor of 2 !

Change the alphabet

a$	0
aa	1
ab	2
b$	3
ba	4
bb	5



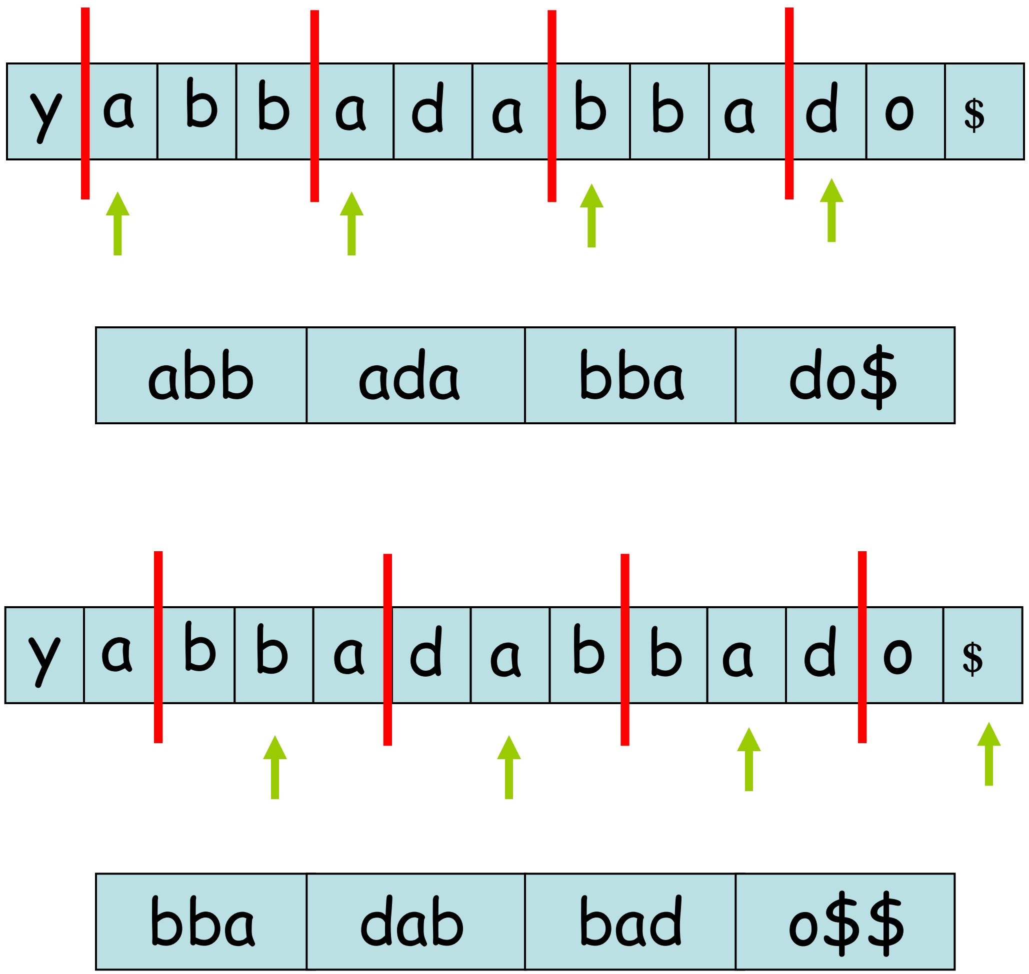
But we do not gain anything…

Divide into triples

y	a	b	b	a	d	a	b	b	a	d	o	$

abb	ada	bba	do$

Divide into triples



Sort recursively 2/3 of the suffixes

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

0	1	2	3	4	5	6	7
abb	ada	bba	do$	bba	dab	bad	o$$
1	2	4	6	4	5	3	7
0	1	6	4	2	5	3	7
1	4	8	2	7	5	10	11

y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

Sort the remaining third

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

(y, 1) (b, 2) (a, 5) (a, 7)

$\rightarrow$

(a, 5) (a, 7) (b, 2) (y, 1)

6 9 3 0

1 4 8 2 7 5 10 11

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

6 9 3 0

1 4 8 2 7 5 10 11

1

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

6 9 3 0

4 8 2 7 5 10 11

1 6

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

9 3 0

4 8 2 7 5 10 11

1 6 4

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

9 3 0

8 2 7 5 10 11

1 6 4 9

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

3 0

8 2 7 5 10 11

1 6 4 9 3

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

0

8 2 7 5 10 11

1 6 4 9 3 8

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

0

2 7 5 10 11

1 6 4 9 3 8 2

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

0

7 5 10 11

1 6 4 9 3 8 2 7

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

0

5 10 11

1 6 4 9 3 8 2 7 5

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

0

10 11

1 6 4 9 3 8 2 7 5

Merge

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

1 6 4 9 3 8 2 7 5 10 11 0

summary

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$
	1	4		2	6		5	3		7	8	

1 6 4 9 3 8 2 7 5 10 11 0

When comparing to a suffix with index 1 (mod 3) we compare the char and break ties by the ranks of the following suffixes

When comparing to a suffix with index 2 (mod 3) we compare the char, the next char if there is a tie, and finally the ranks of the following suffixes

Compute LCP's

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

Crucial observation

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

$$LCP(i,j) = \min\{LCP(i,i+1), LCP(i+1,i+2), \ldots, LCP(j-1,j)\}$$

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

Find LCP's of consecutive suffixes

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

0

LCP(11,0)

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
						1					0

LCP(8,2)

0	yabbadabbado$
11	o$
10	do$
5	dabbado$
7	bbado$
2	bbadabbado$
8	bado$
3	badabbado$
9	ado$
4	adabbado$
6	abbado$
1	abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
				0		1					0

LCP(9,3)

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
		1		0		1					0

LCP(6,4)

0	yabbadabbado$
11	o$
10	do$
5	dabbado$
7	bbado$
2	bbadabbado$
8	bado$
3	badabbado$
9	ado$
4	adabbado$
6	abbado$
1	abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
		1		0		1		0			0

LCP(7,5)

0	yabbadabbado$
11	o$
10	do$
5	dabbado$
7	bbado$
2	bbadabbado$
8	bado$
3	badabbado$
9	ado$
4	adabbado$
6	abbado$
1	abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
	5	1		0		1		0			0

LCP(1,6)

0	yabbadabbado$
11	o$
10	do$
5	dabbado$
7	bbado$
2	bbadabbado$
8	bado$
3	badabbado$
9	ado$
4	adabbado$
6	abbado$
1	abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
	5	1		0		1	4	0			0

LCP(2,7)

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1	6	4	9	3	8	2	7	5	10	11	0
	5	1		0	3	1	4	0			0

LCP(3,8)

0	yabbadabbado$
11	o$
10	do$
5	dabbado$
7	bbado$
2	bbadabbado$
8	bado$
3	badabbado$
9	ado$
4	adabbado$
6	abbado$
1	abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

5 1 2 0 3 1 4 0 0

LCP(4,9)

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

5 1 2 0 3 1 4 0 1 0

LCP(5,10)

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

5 1 2 0 3 1 4 0 1 0 0

LCP(10,11)

because of the fact that the array is sorted, if for a pair LCP is not zero then the consecutive next pair is predictable!

Probably

0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

Analysis

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

5 1 2 0 3 1 4 0 1 0 0

The starting position deceases by 1 in every iteration. So it cannot increase more than O(n) times

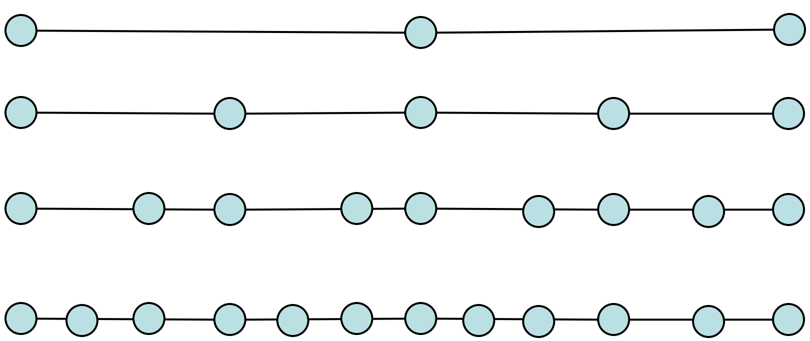
0 yabbadabbado$
11 o$
10 do$
5 dabbado$
7 bbado$
2 bbadabbado$
8 bado$
3 badabbado$
9 ado$
4 adabbado$
6 abbado$
1 abbadabbado$

We need more LCPs for search

0	1	2	3	4	5	6	7	8	9	10	11	12
y	a	b	b	a	d	a	b	b	a	d	o	$

1 6 4 9 3 8 2 7 5 10 11 0

5 1 2 0 3 1 4 0 1 0 0



Linearly many, calculate the all bottom up

Another example

1	2	3	4	5	6	7	8	9
a	b	c	a	b	b	c	a	$

4 1 8 5 2 6 3 7 9

2 1 0 1 3 0 2 0

4 abbca$
1 abcabbca$
8 a$
5 bbca$
2 bcabbca$
6 bca$
3 cabbca$
7 ca$
9 $

Analysis

Think about the LCP which we know at any point in the algorithm

A successful comparison increases it by one

It decreases by one when iteration starts

So the number of successful comparisons is O(n)