Suffix trees

Definition and applications

Recall tries

- We can search in a trie containing multiple strings.
    - As a function of the key-length what is the complexity?

How would you use it to search for all occurrences of a key in a single string?

What is the space complexity?

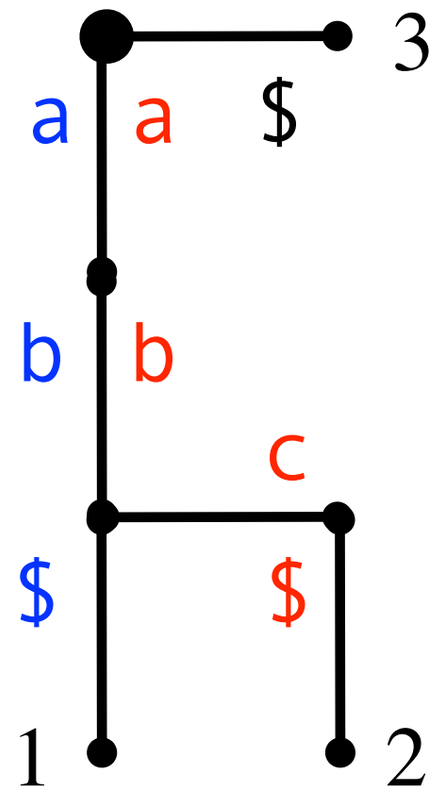
Compacted tries

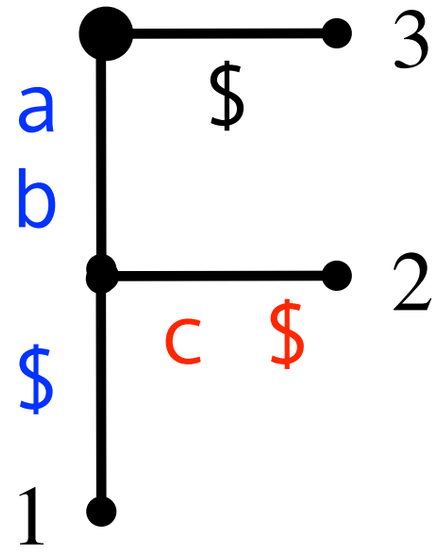
Saving space: Eliminate all internal nodes of degree 2 ...

x_1 = a b $

x_2 = a b c $

x_3 = $





If we have n input-strings, then the trie has n+1 leaves and at most n internal nodes, i.e space O(n) for the tree. What about the labels?

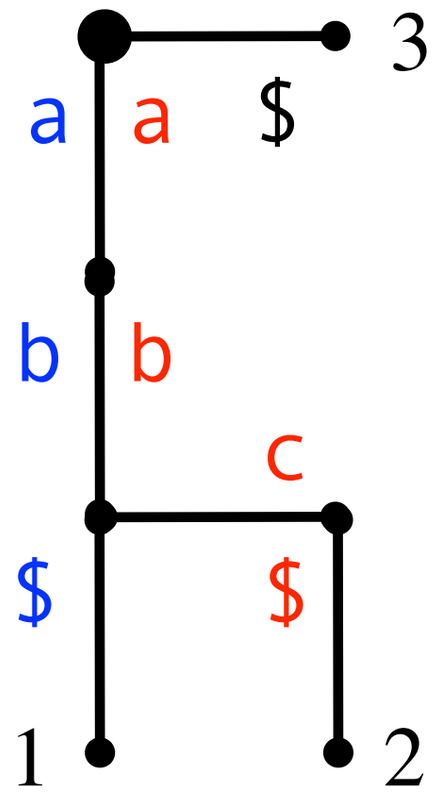
Compacted tries

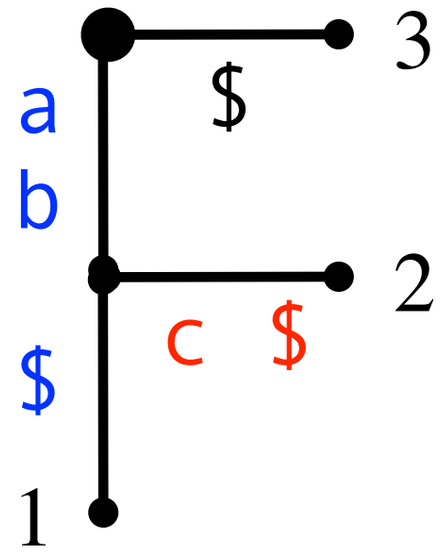
Saving space: Eliminate all internal nodes of degree 2 ...

x_1 = a b $

x_2 = a b c $

x_3 = $





If we have n input-strings, then the trie has n+1 leaves and at most n internal nodes, i.e space O(n) for the tree. What about the labels?

Labels can be represented in space O(1), i.e. "ab" ⇒ (1,0,2)

Suffix tree

The ***suffix tree*** $T(\boldsymbol{x})$ of string $\boldsymbol{x}$ is the **compacted trie** of all suffixes $\boldsymbol{x}[i{:}n]$ for $i = 0,..., n$, i.e. including the empty suffix

Suffix tree

The ***suffix tree*** $T(\boldsymbol{x})$ of string $\boldsymbol{x}$ is the **compacted trie** of all suffixes $\boldsymbol{x}[i{:}n]$ for $i = 0,..., n$, i.e. including the empty suffix

Example for x = `tatat`

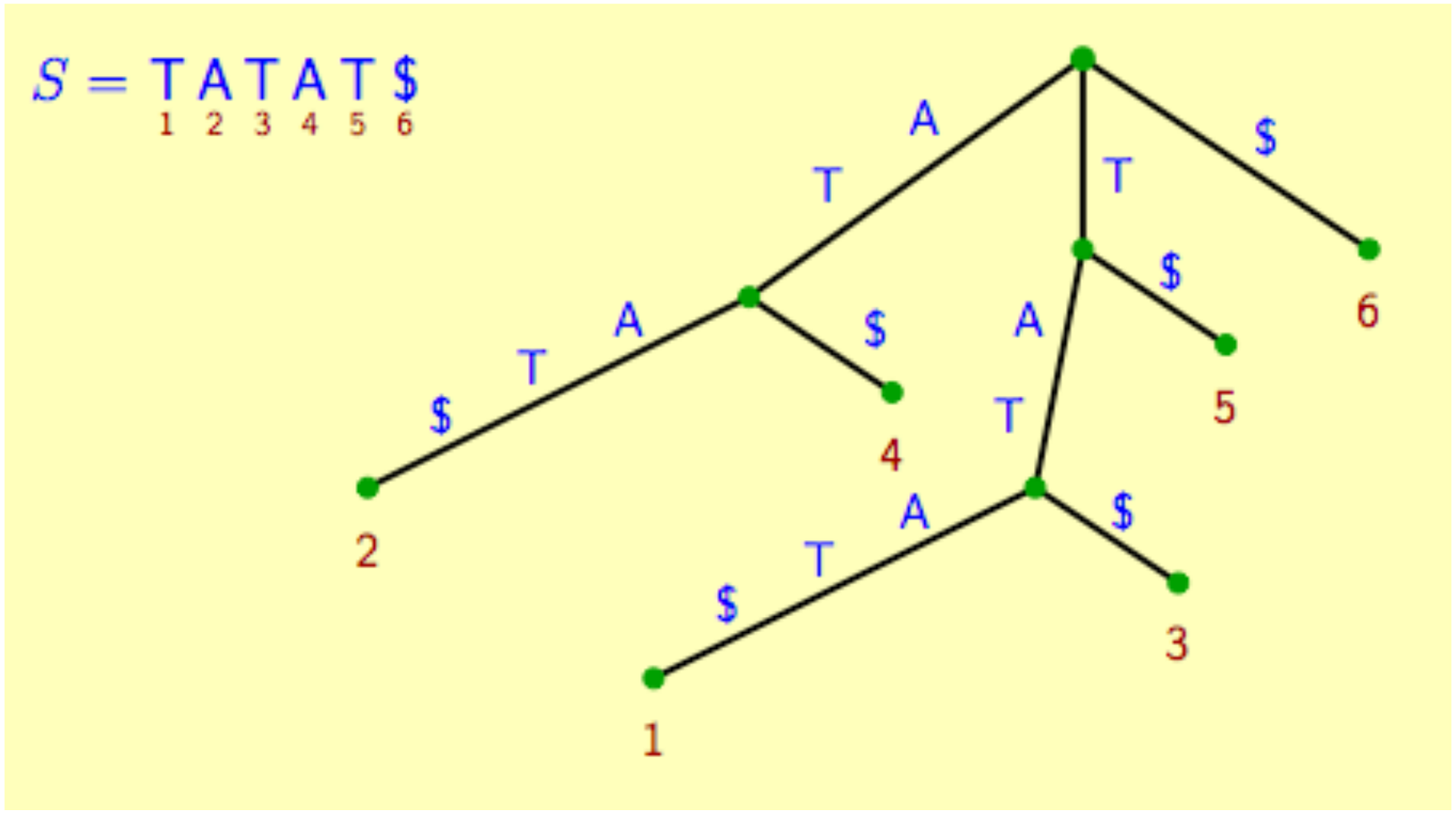
```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

Suffix tree

The ***suffix tree*** $T(\boldsymbol{x})$ of string $\boldsymbol{x}$ is the **compacted trie** of all suffixes $\boldsymbol{x}[i{:}n]$ for $i = 0,..., n$, i.e. including the empty suffix

Example for x = `tatat`

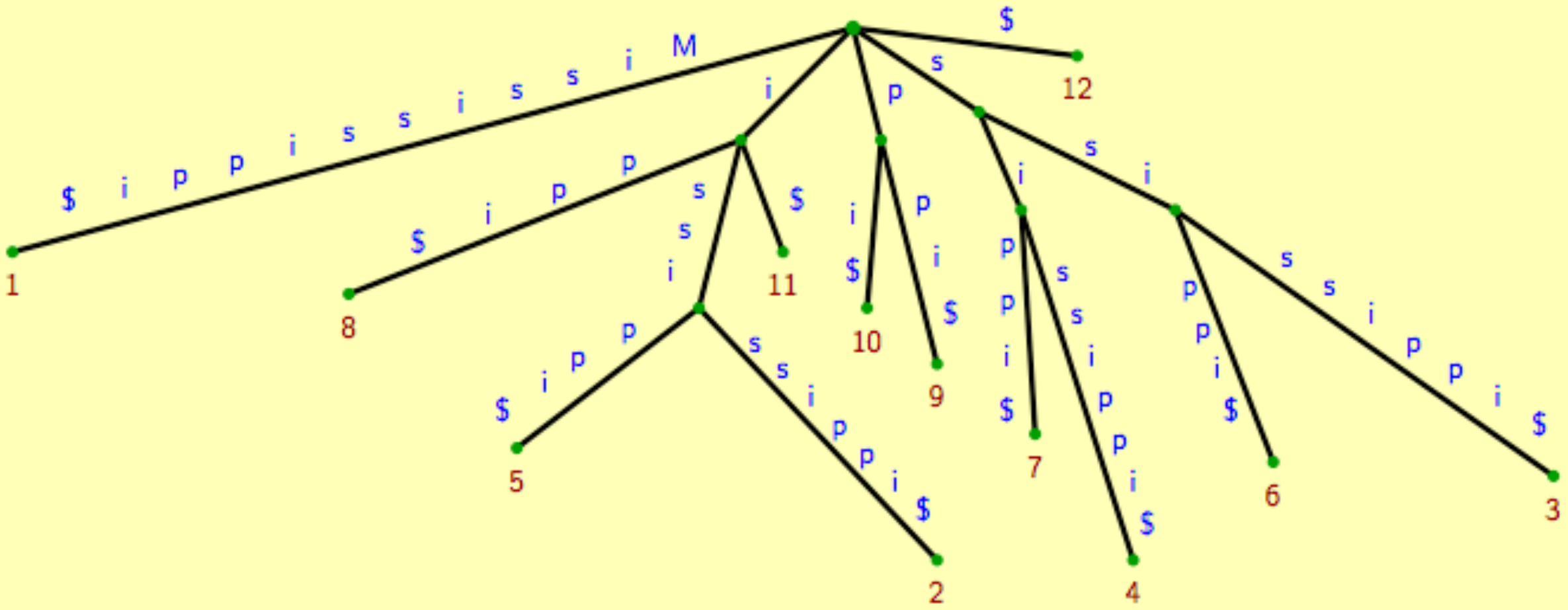
```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```



A larger example

$S =$ M i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12

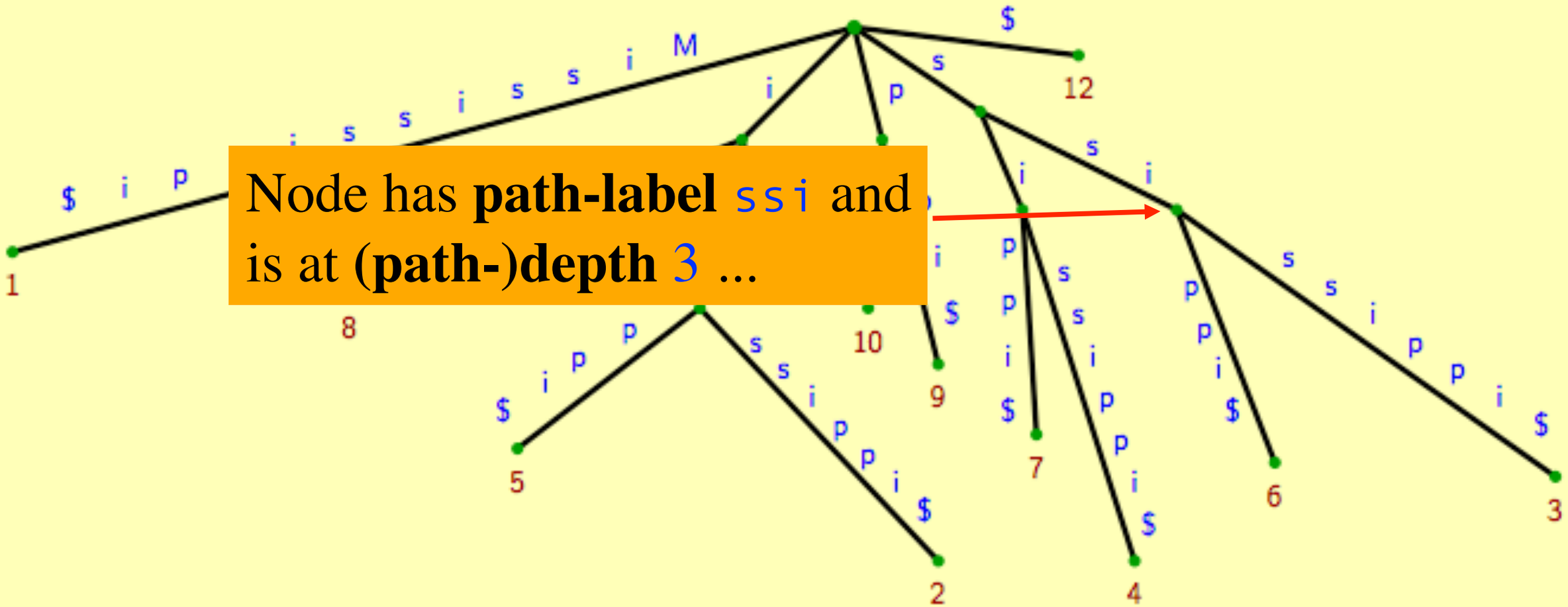


```
> gsa show suffixtree x | idot
```

A larger example

$S =$ M i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12

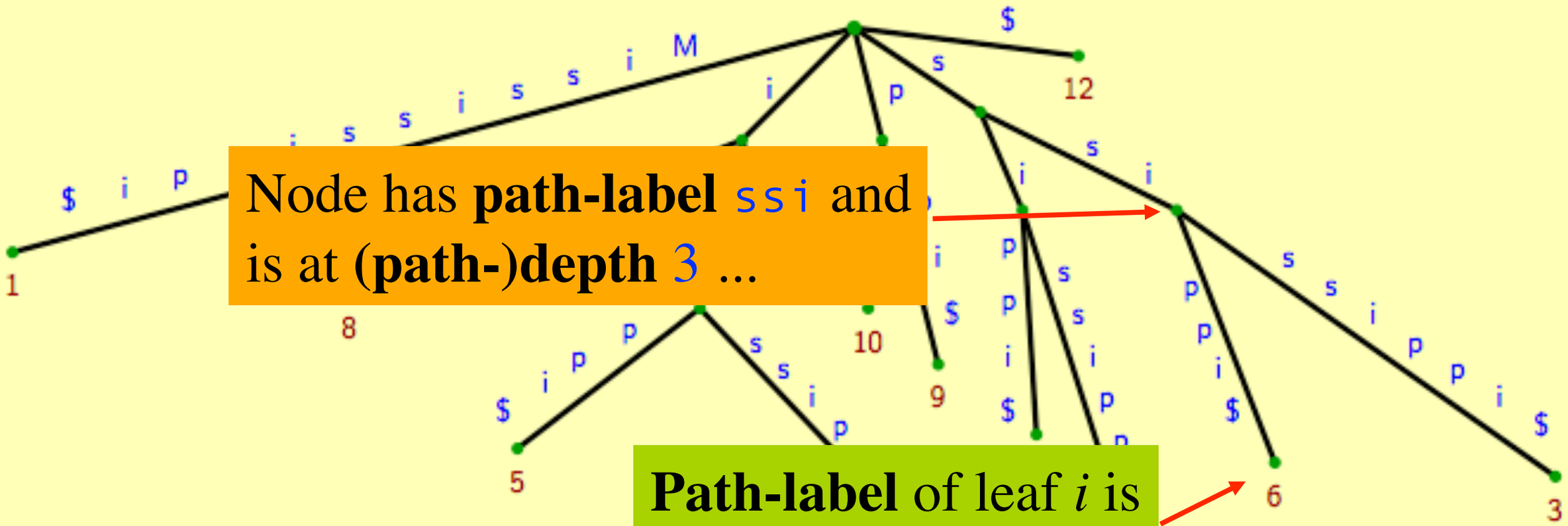


Node has **path-label** `ssi` and is at **(path-)depth** 3 ...

A larger example

$S =$ M i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12



Node has **path-label** `ssi` and is at **(path-)depth** 3 ...

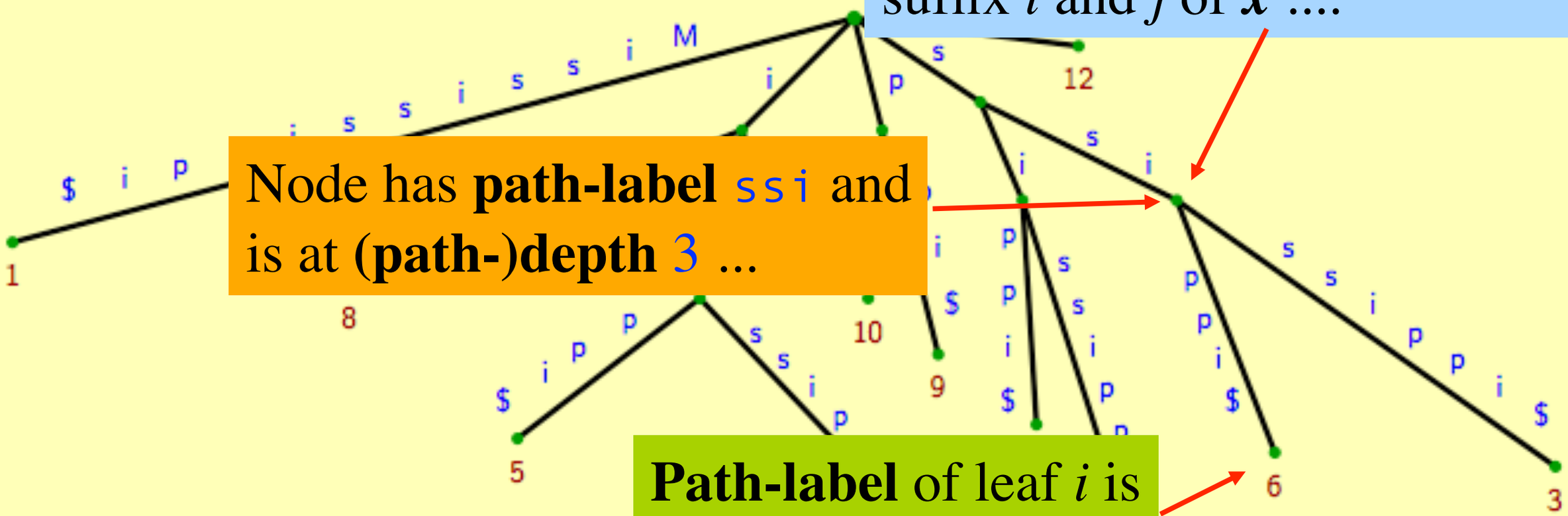
Path-label of leaf *i* is suffix *i*, i.e. ***x***[*i*:*n*]$ ...

A larger example

Path-label of lowest common ancestor of leaf i and j, is longest common prefix of suffix i and j of $\boldsymbol{x}$ ....

$S =$ M i s s i s s i p p i $

1 2 3 4 5 6 7 8 9 10 11 12



Node has **path-label** `ssi` and is at **(path-)depth** 3 ...

Path-label of leaf i is suffix i, i.e. $\boldsymbol{x}[i{:}n]$\$ ...

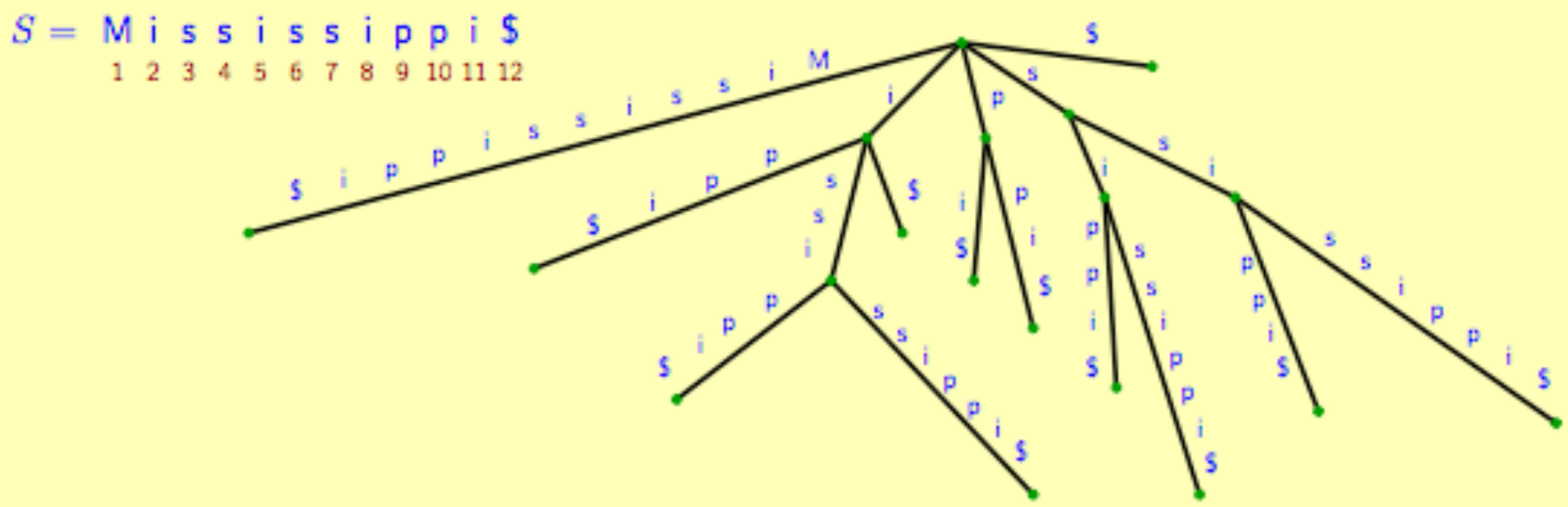
What is the space complexity?

Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers (i, j) into S.

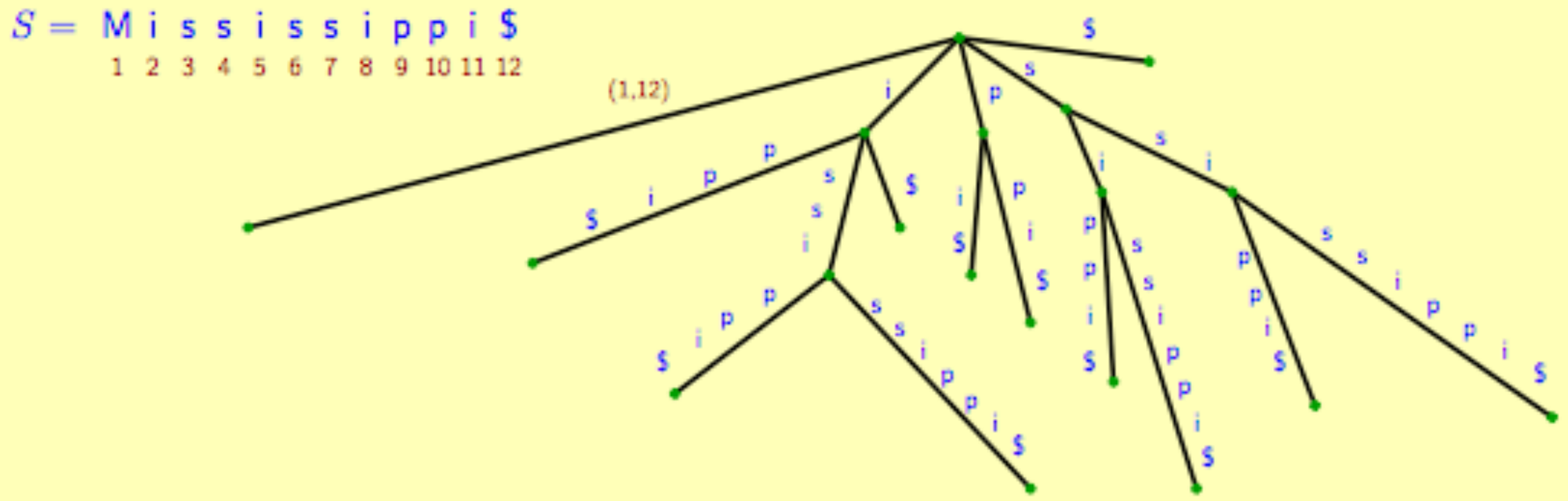


Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers (i, j) into S.

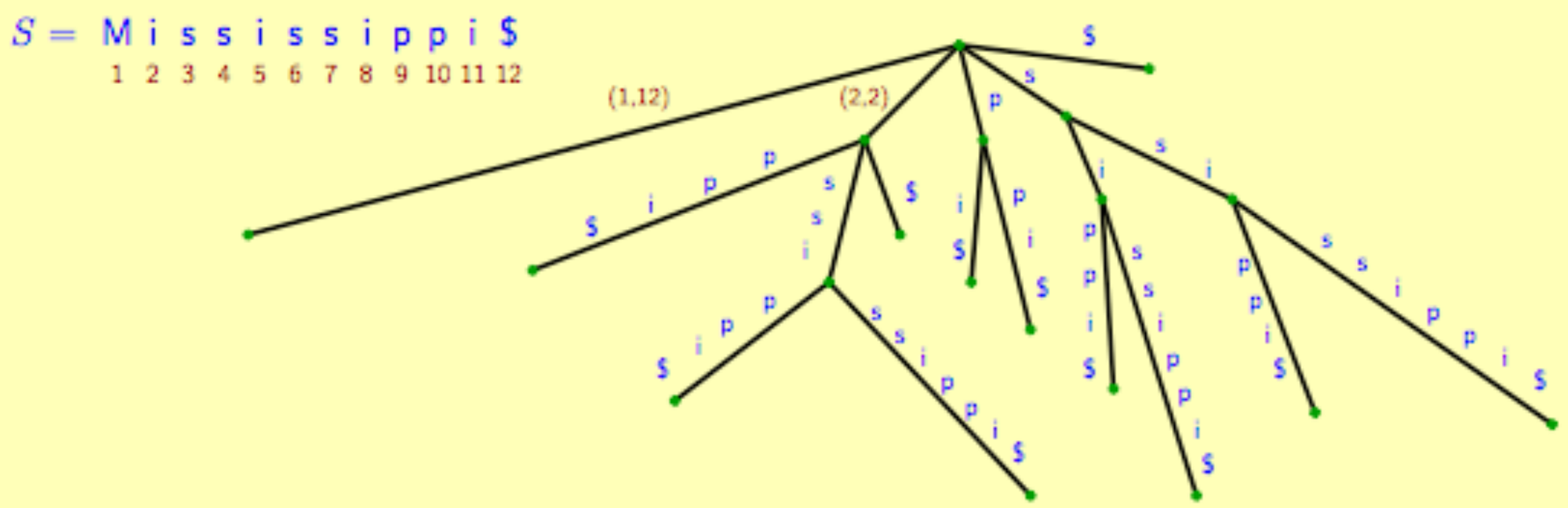


Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers (i, j) into S.

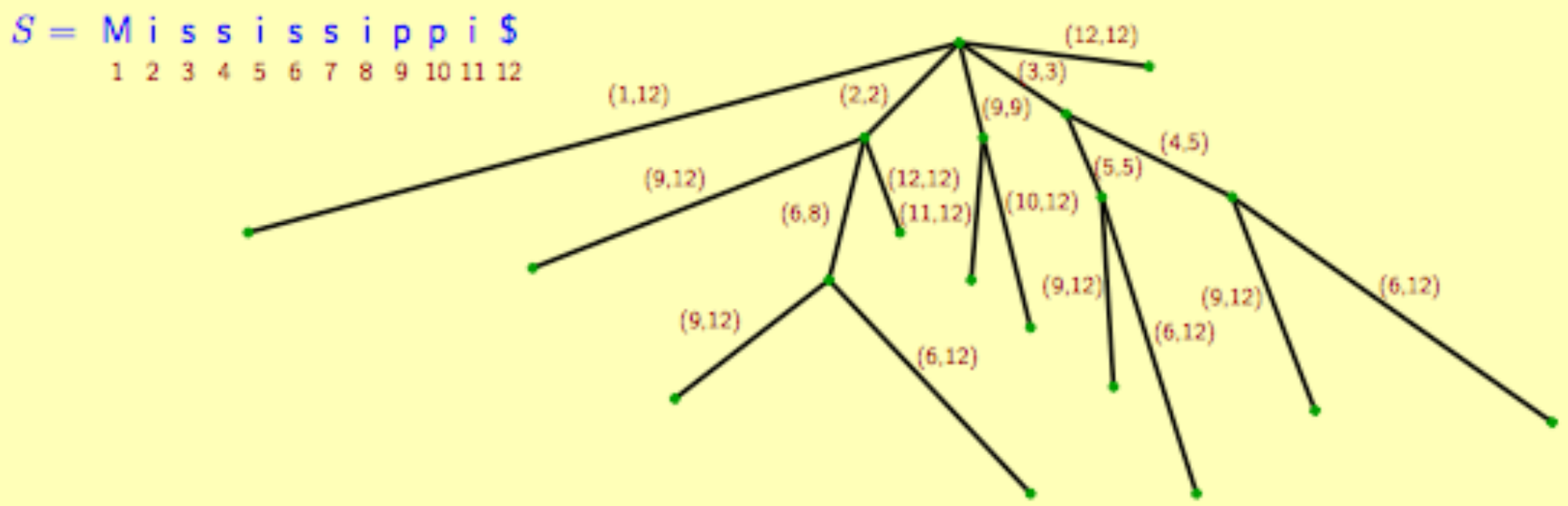


Space consumption

Observation: $T(S)$ requires $\mathcal{O}(n)$ space.

Proof sketch:

1. $T(S)$ has at most n leaves.
2. Each internal node is branching $\Rightarrow$ at most $n-1$ internal nodes.
3. A tree with at most $2n-1$ nodes has at most $2n-2$ edges.
4. Each node requires constant space.
5. Each edge label is a substring of $S \Rightarrow$ pair of pointers (i, j) into S.



Constructing suffix trees

Constructing $T(\boldsymbol{x})$ by inserting each suffix one by one takes time $O(n^2)$

Can we do better?

Constructing suffix trees

Constructing $T(\boldsymbol{x})$ by inserting each suffix one by one takes time $O(n^2)$

Can we do better?

We will see a linear time construction algorithm next week.

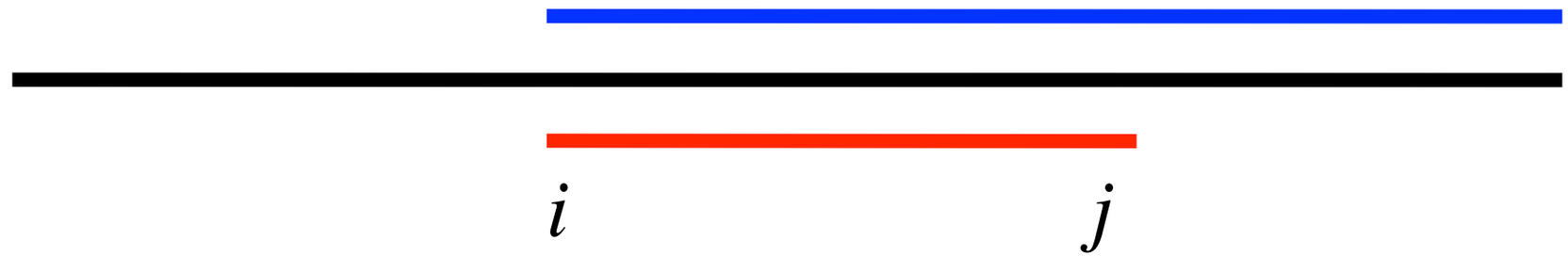
What about applications?

... exact matching, finding repeats, longest common substring ...

Exact matching

Given string x and pattern y, report where y occurs in x

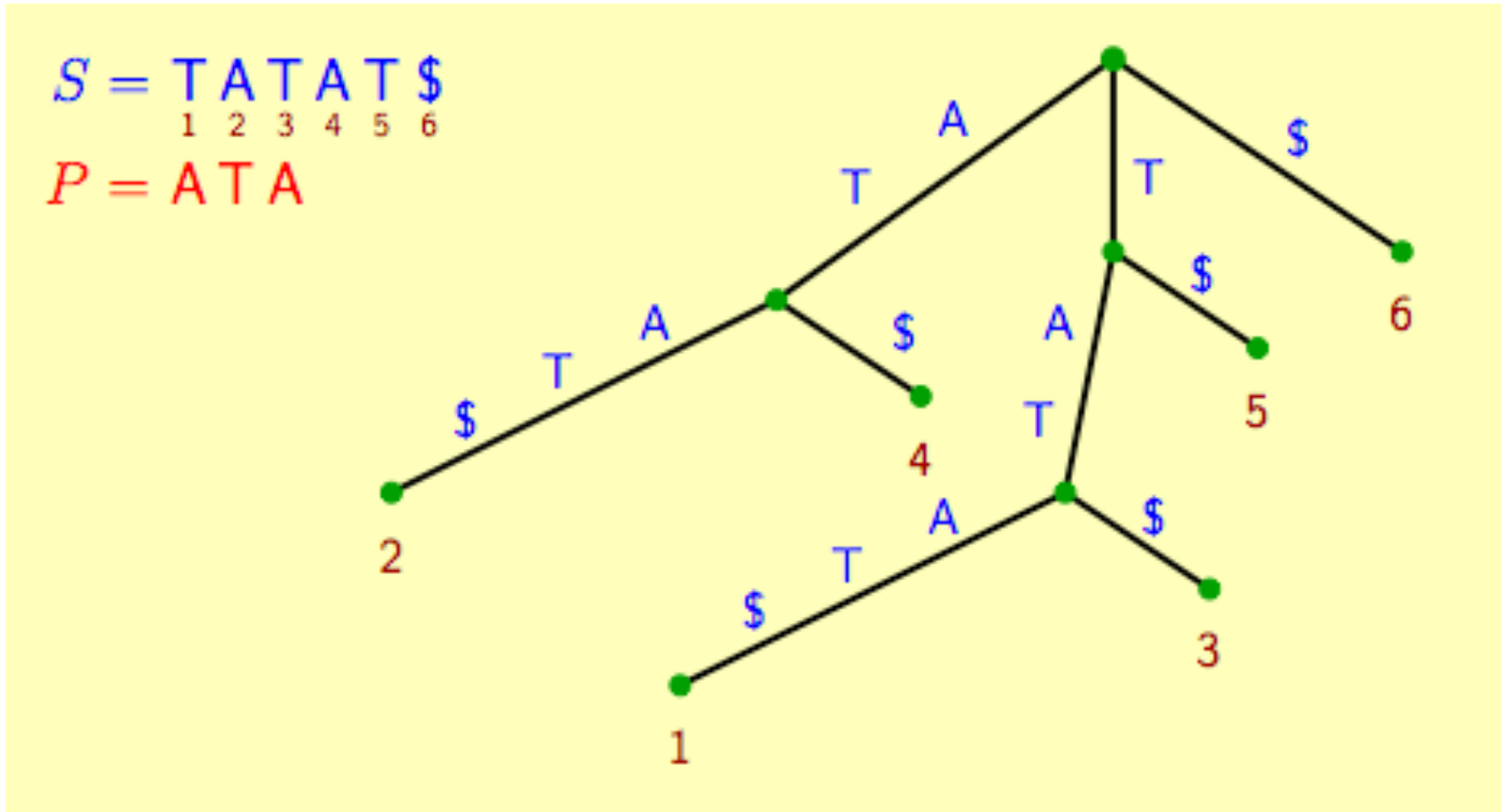
If y occurs in x at position i, then y is a prefix of suffix i of x



y is spelled by an initial part of the path from the root to leaf i in $T(x)$

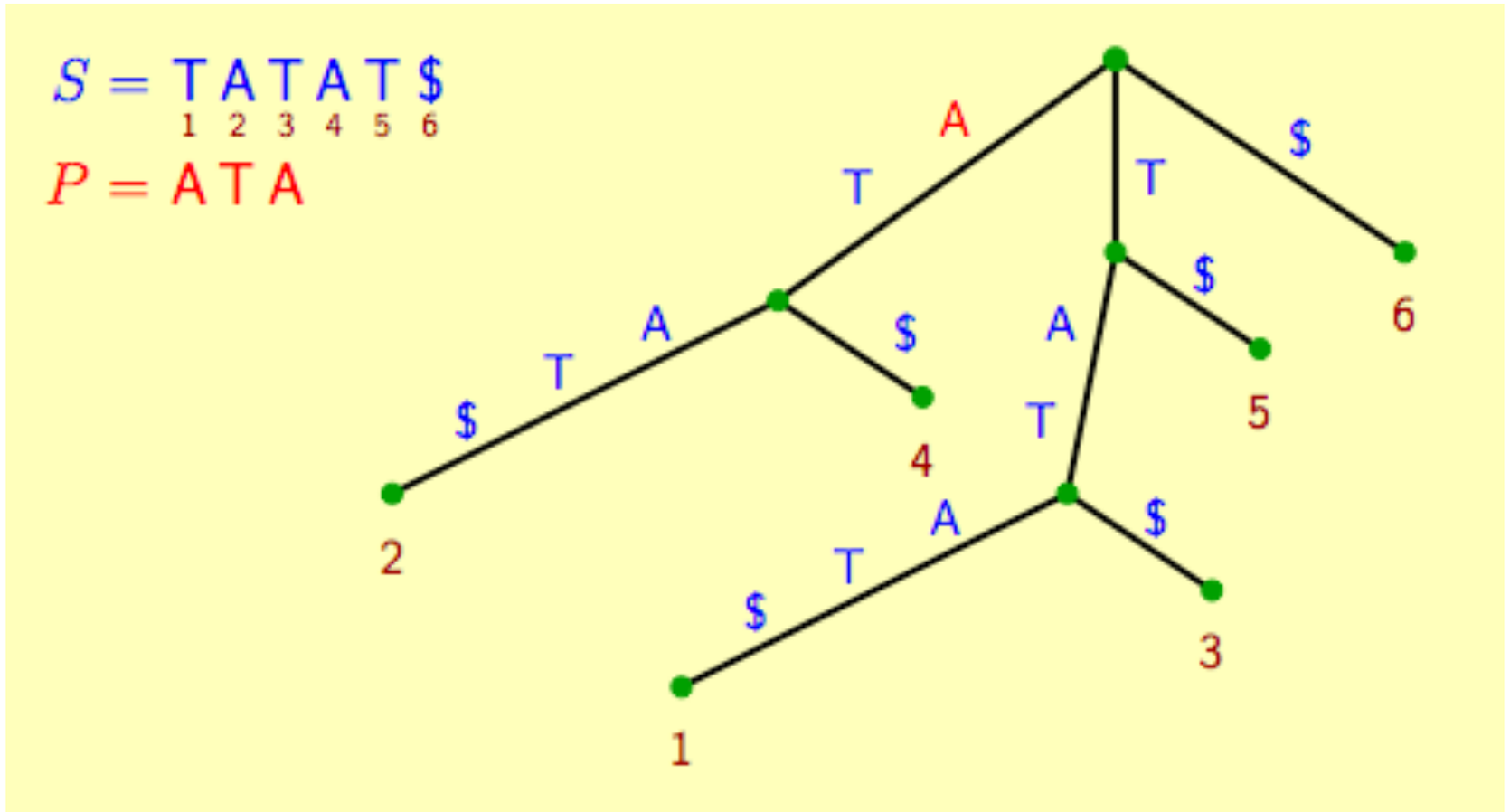
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



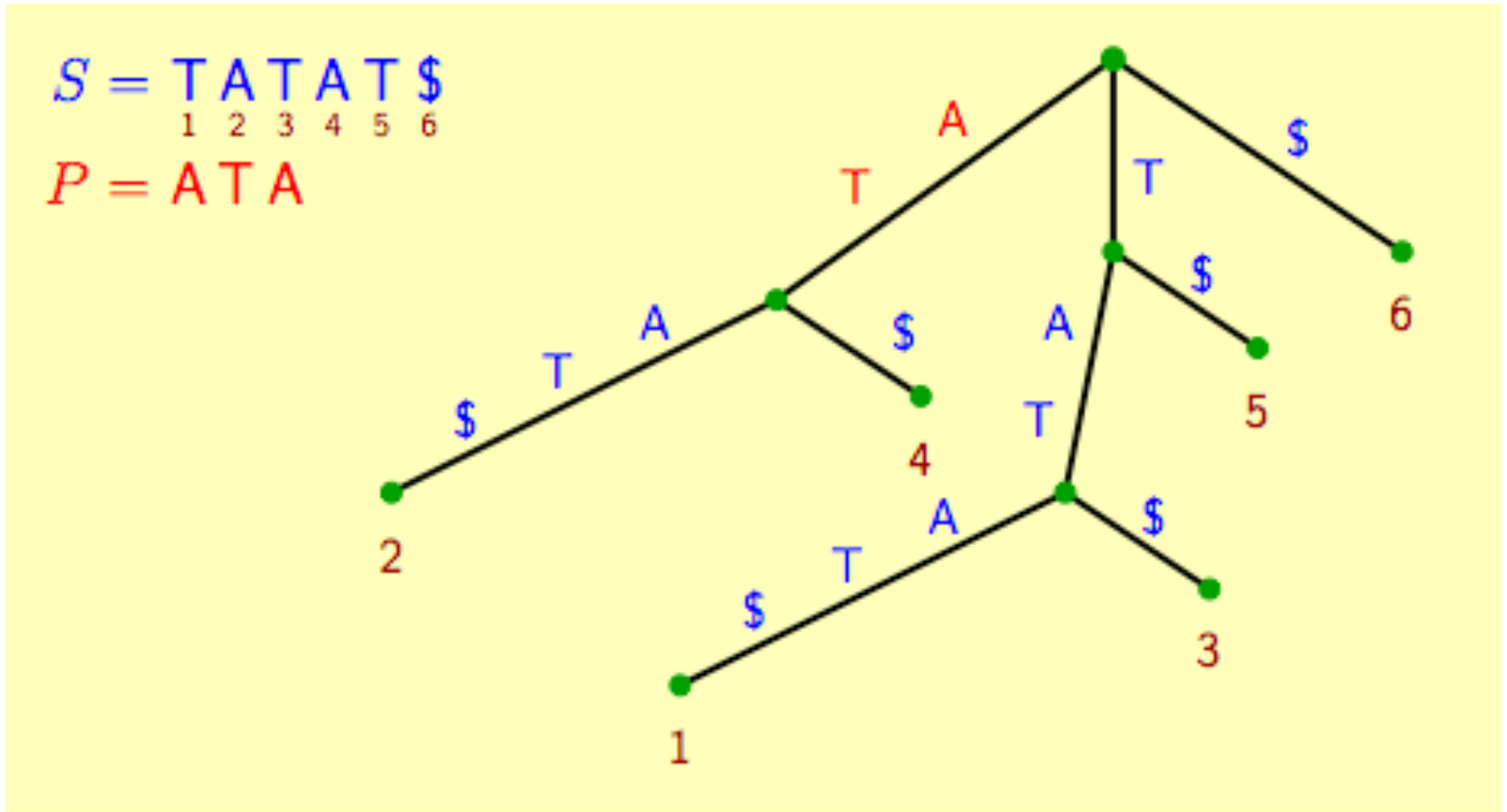
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



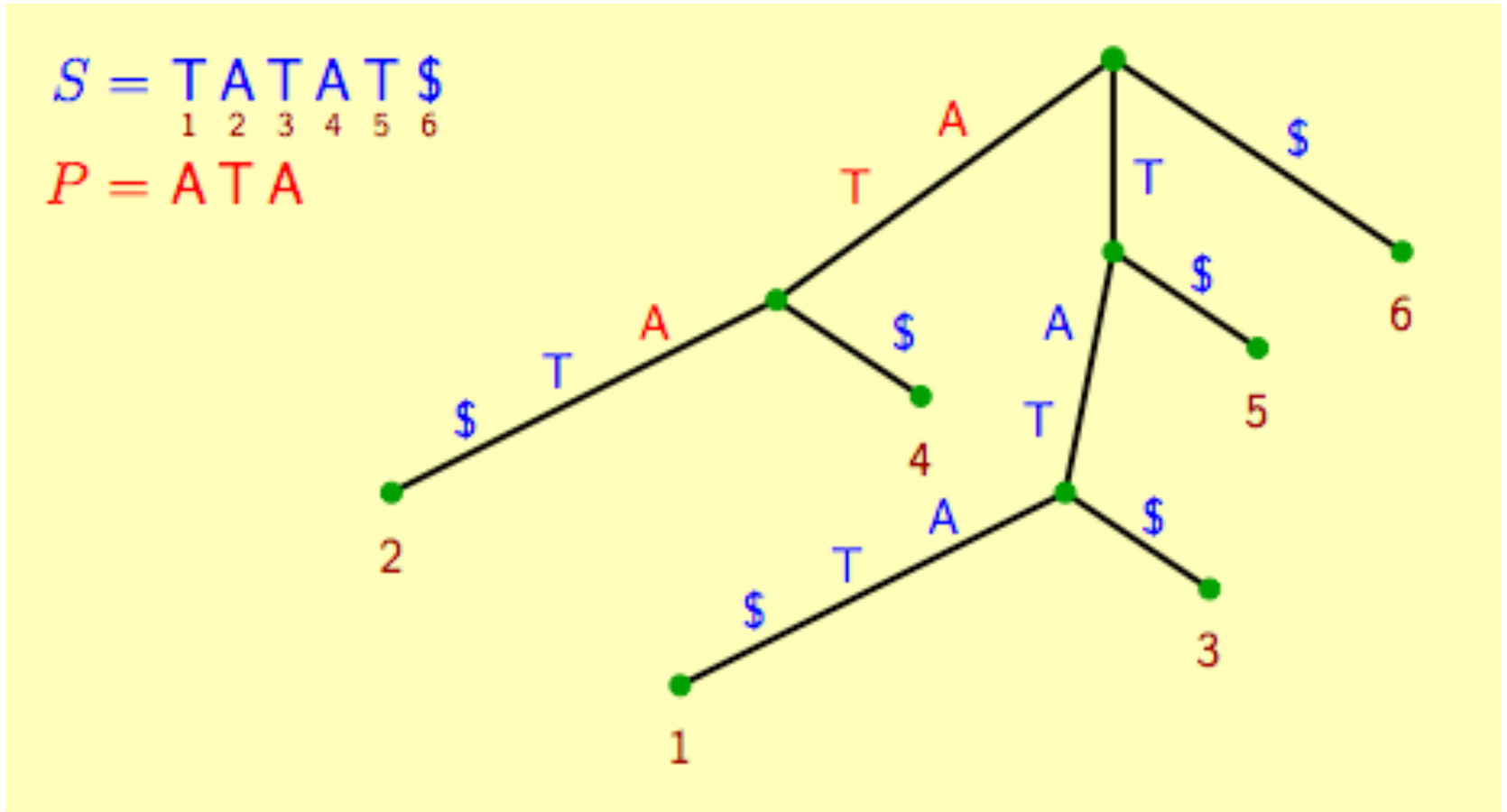
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



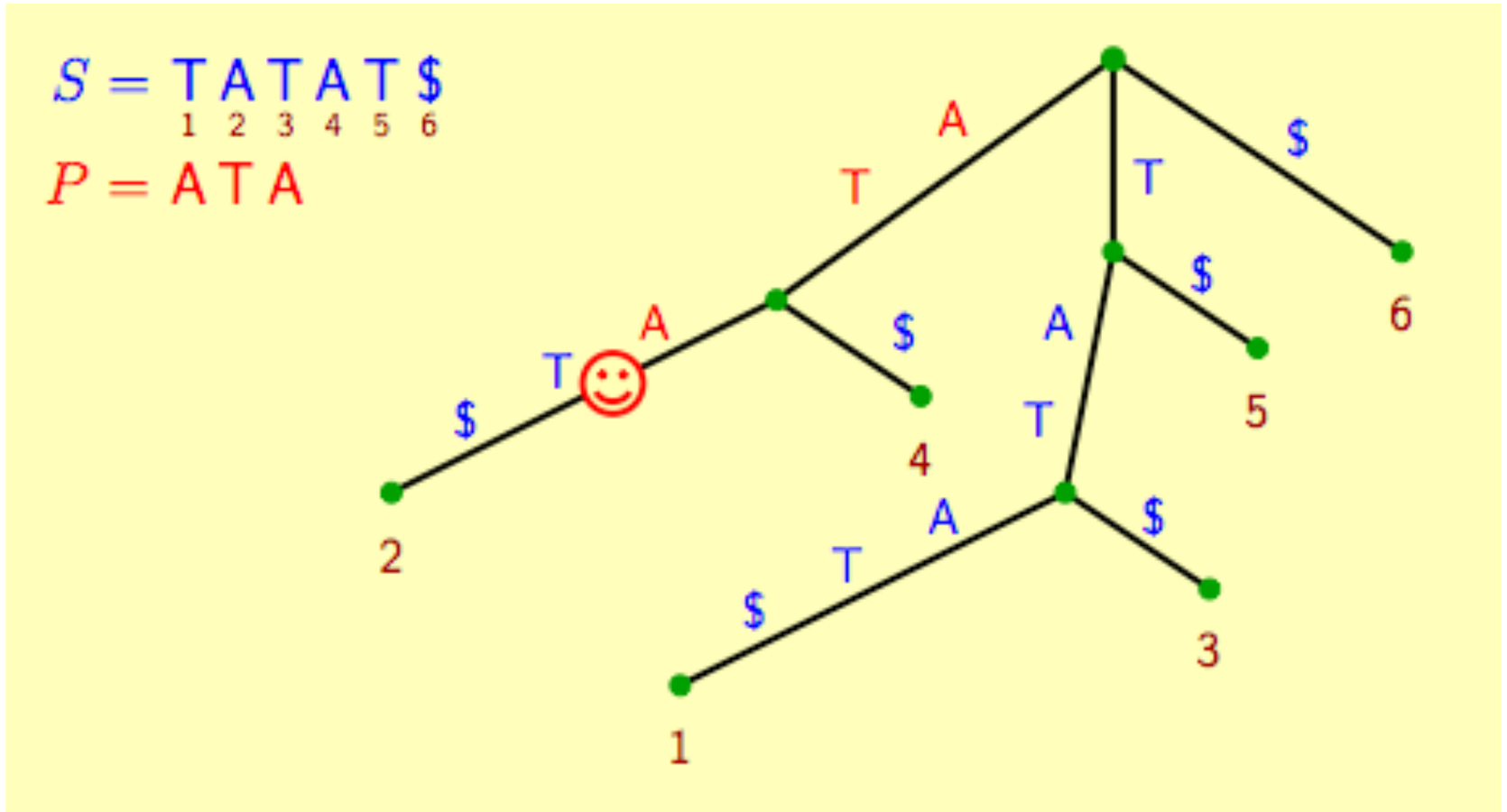
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***

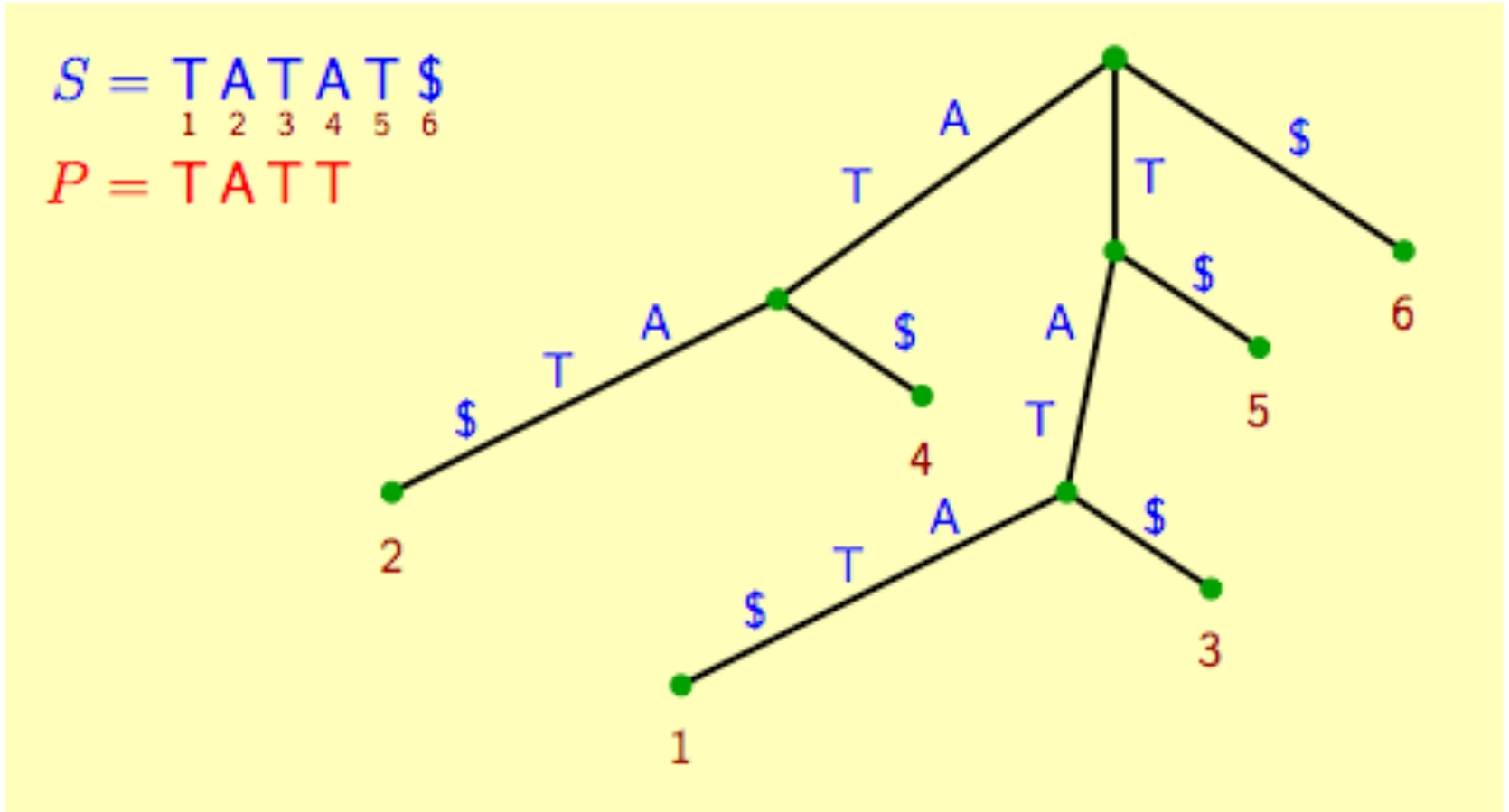


Pattern `ata` occurs at position 2 in `tatat`

Time: O(m) using the suffix tree $T(S)$

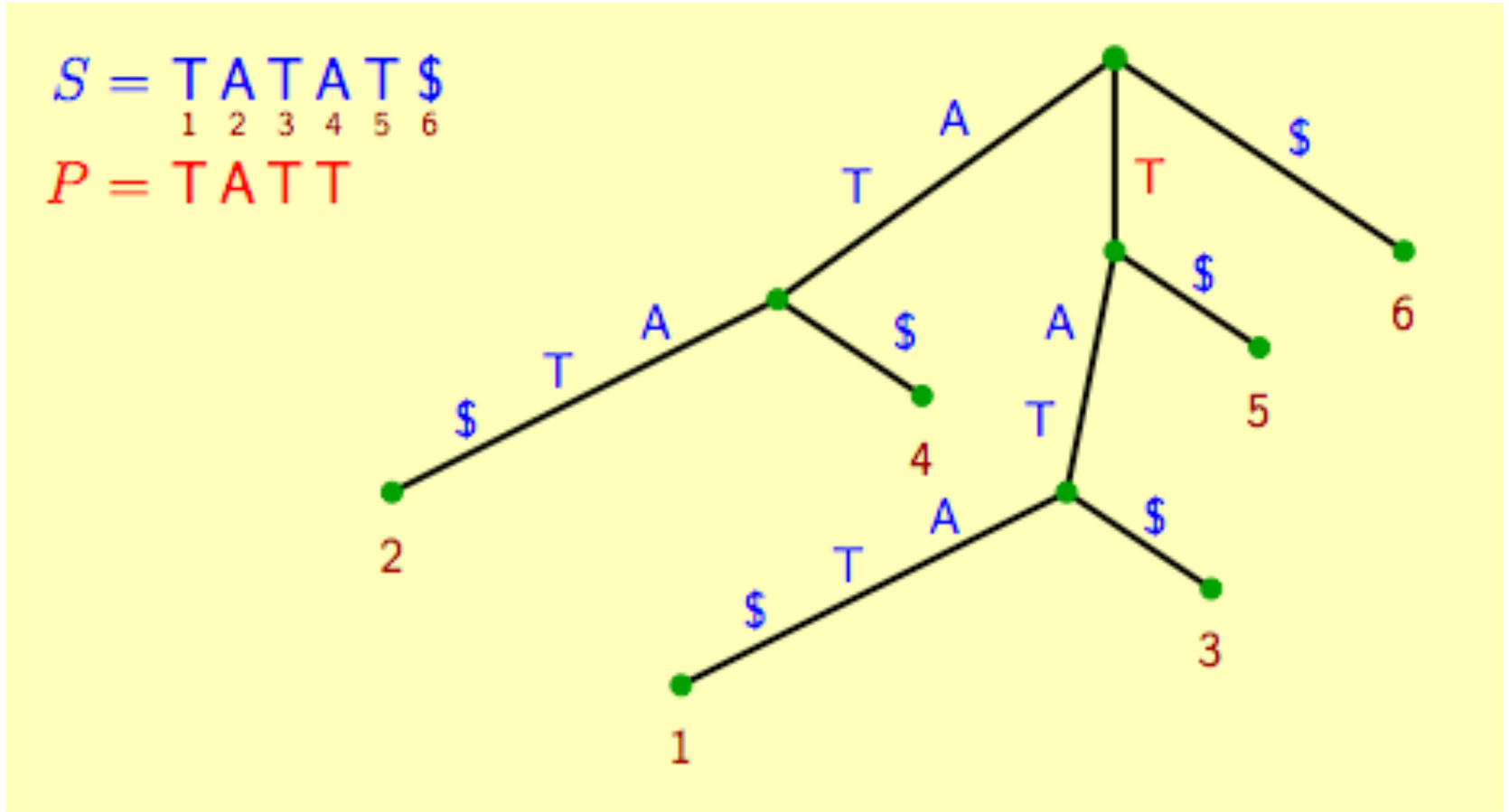
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



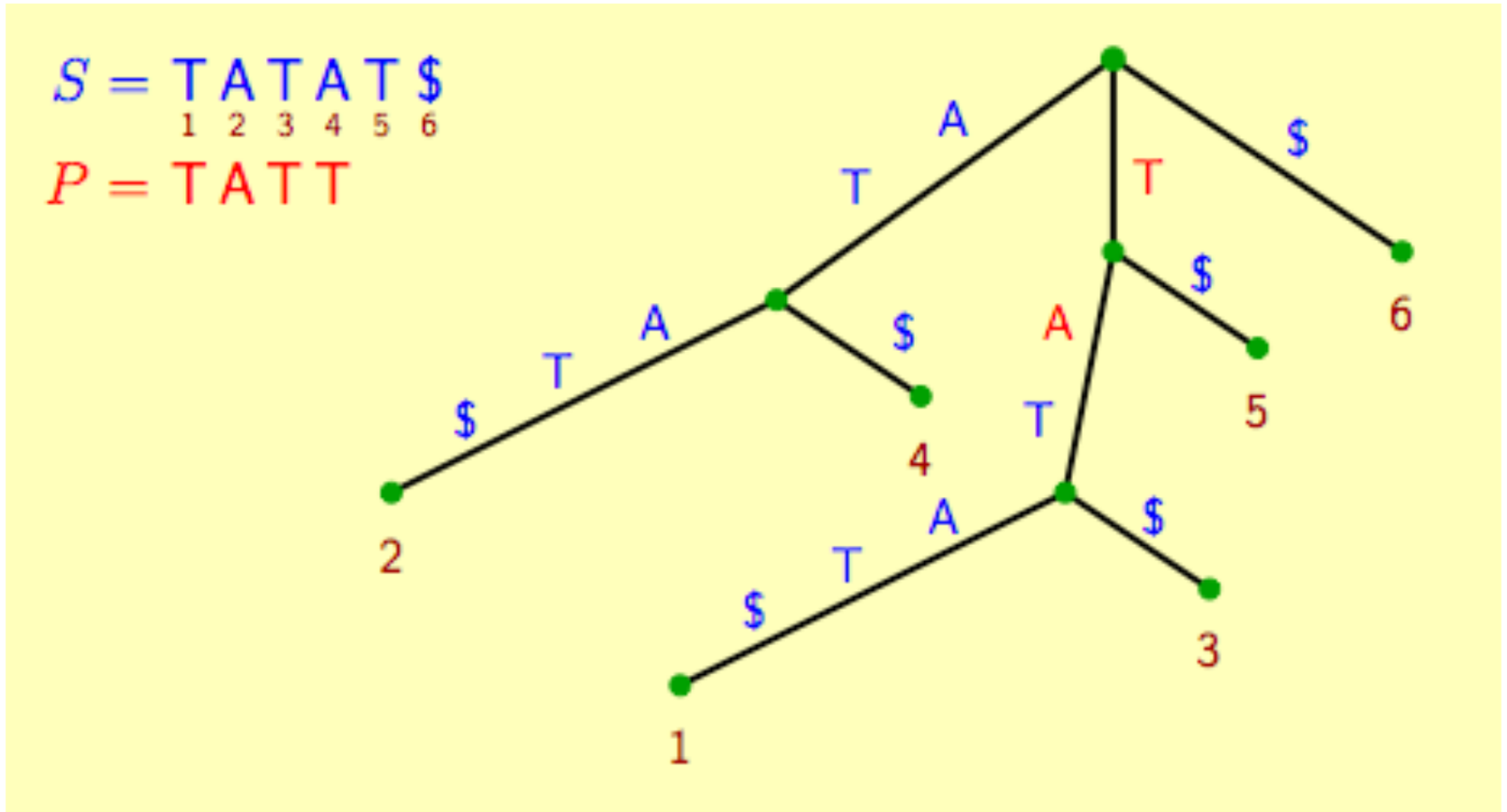
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



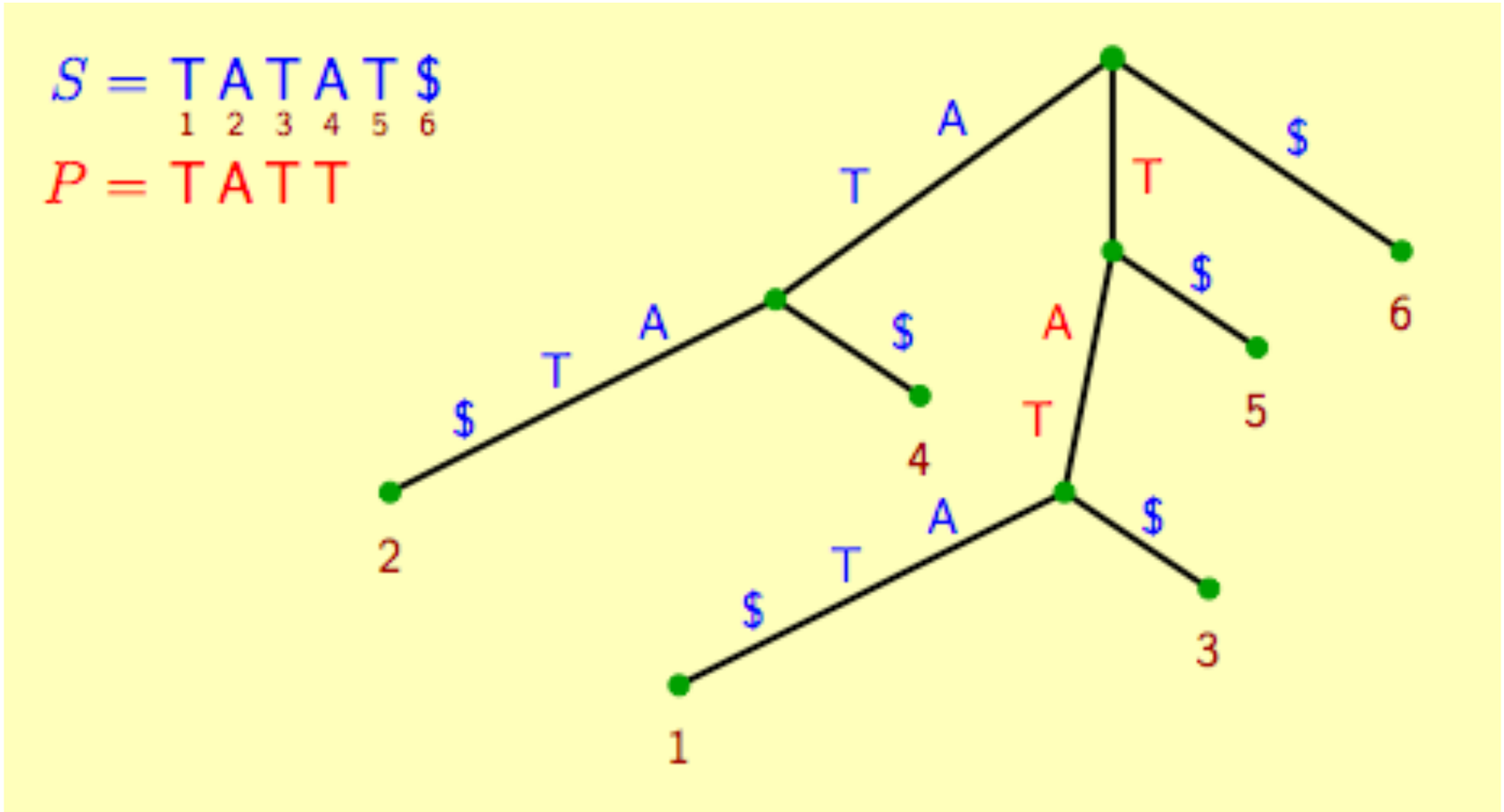
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



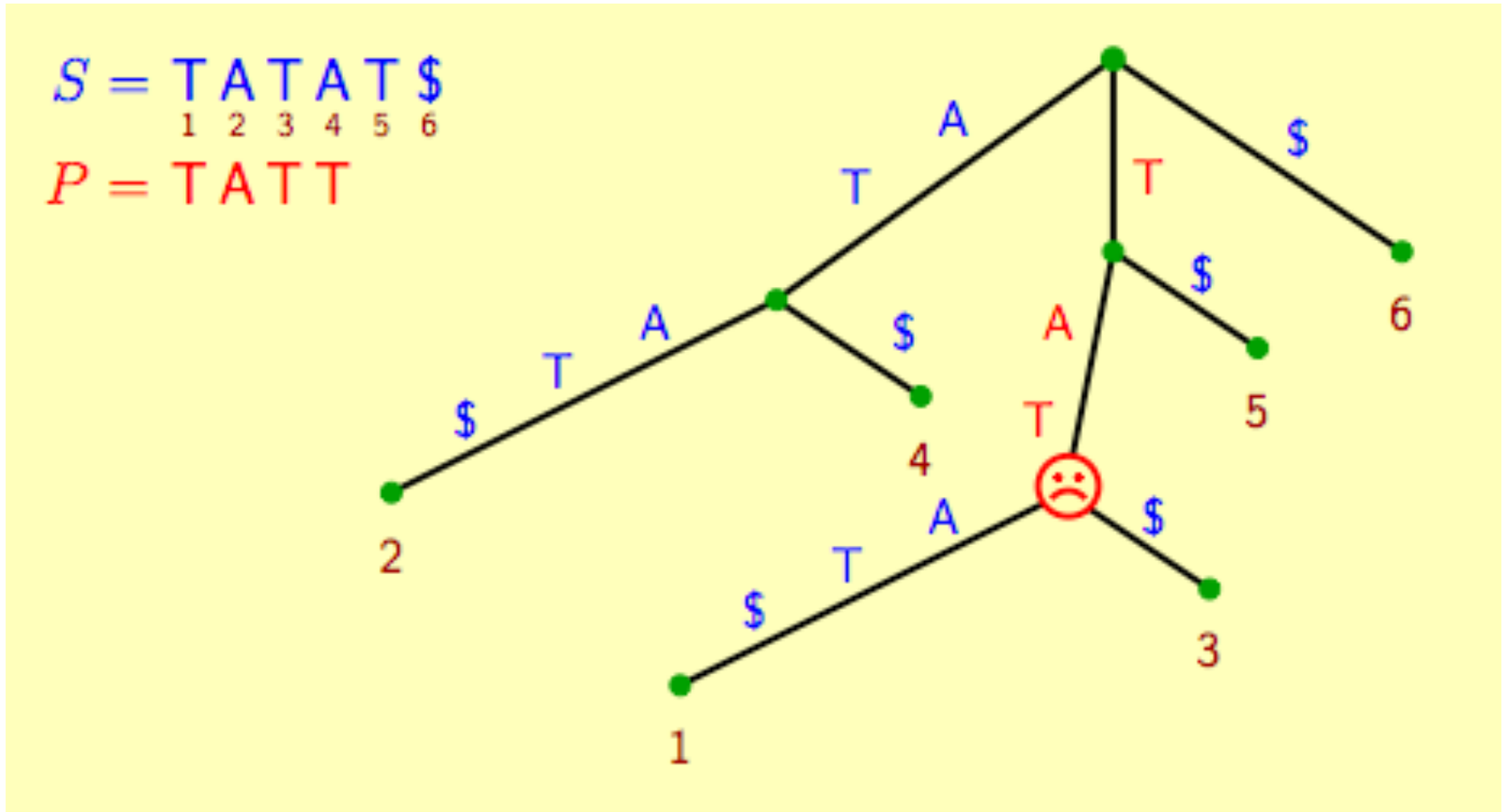
Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



Exact matching

Given string ***x*** and pattern ***y***, report where ***y*** occurs in ***x***



Pattern `tatt` does not occur in `tatat`

Time: O(m) using the suffix tree $T(S)$

Repeats

A pair of substrings R=(S[i_1..j_1], S[i_2..j_2]) is a ...

→ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

Repeats

A pair of substrings R=(S[i_1..j_1], S[i_2..j_2]) is a ...

→ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

→ *k-mismatch* repeat if there are k mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$

Repeats

A pair of substrings R=(S[i_1..j_1], S[i_2..j_2]) is a ...

→ *exact* repeat if $S[i_1, j_1] = S[i_2, j_2]$

→ *k-mismatch* repeat if there are k mismatches between $S[i_1, j_1]$ and $S[i_2, j_2]$

→ *k-differences* repeat if there are k differences (mismatches, insertions, deletions) between $S[i_1, j_1]$ and $S[i_2, j_2]$

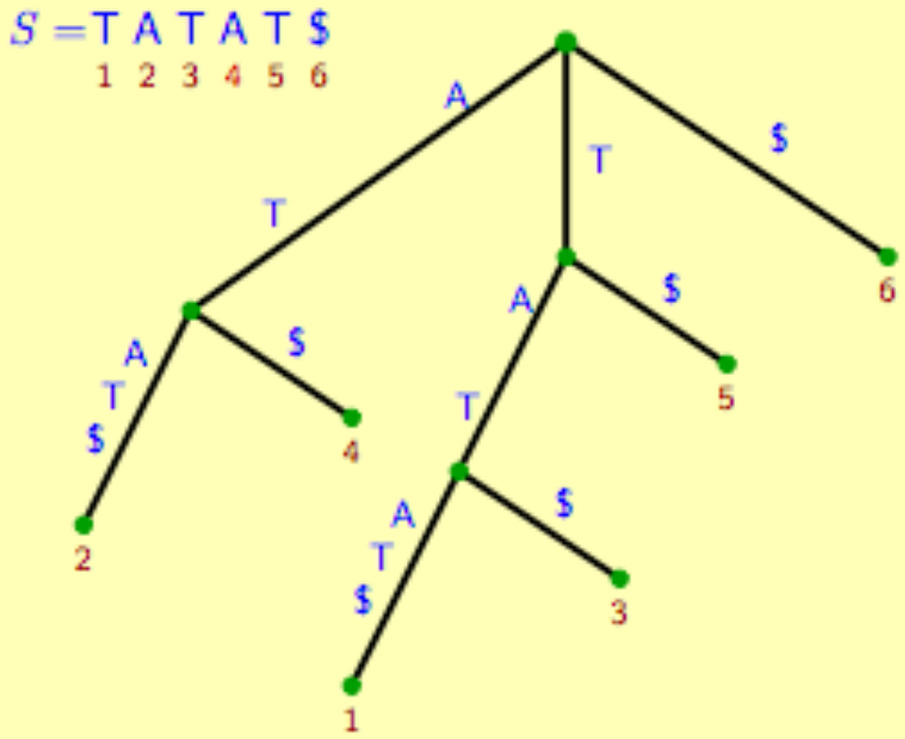
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



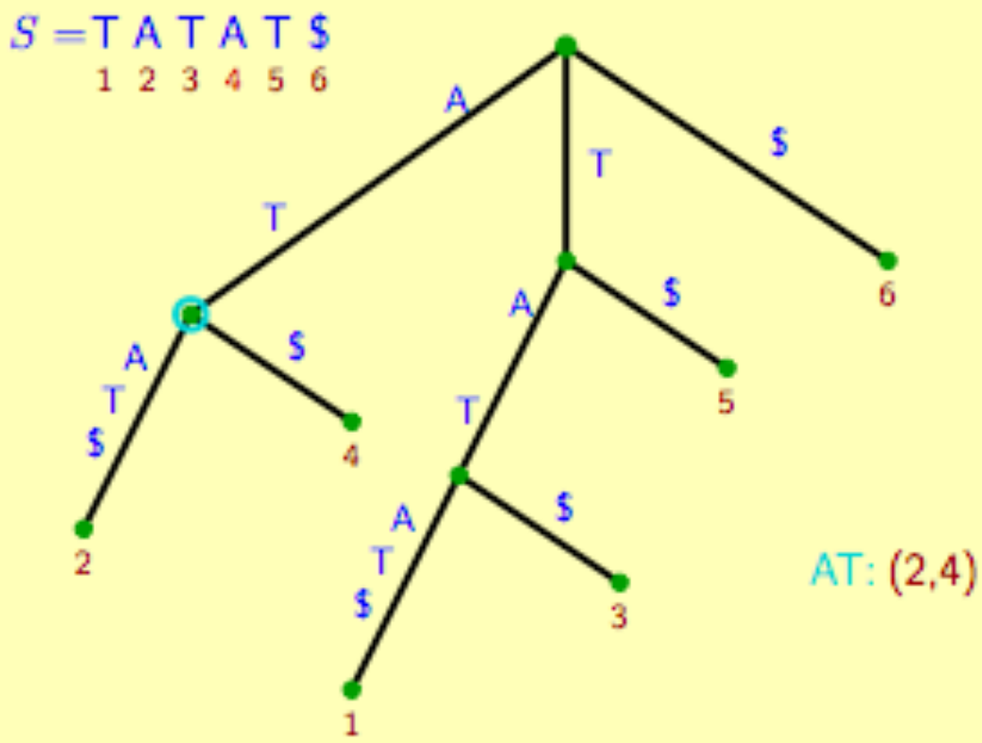
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



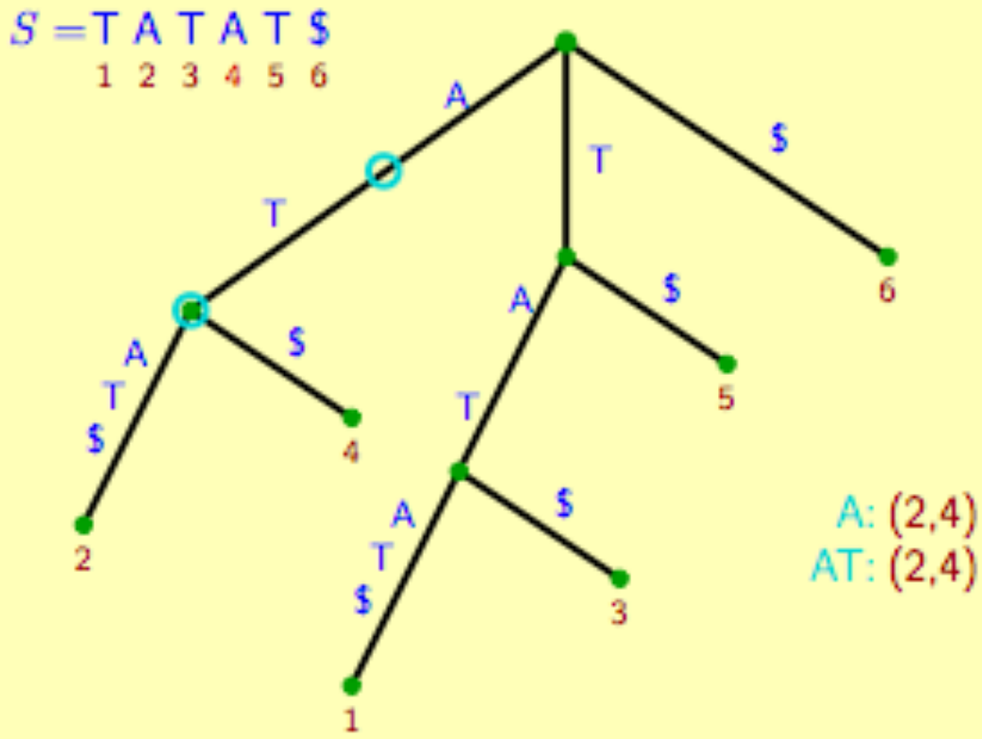
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



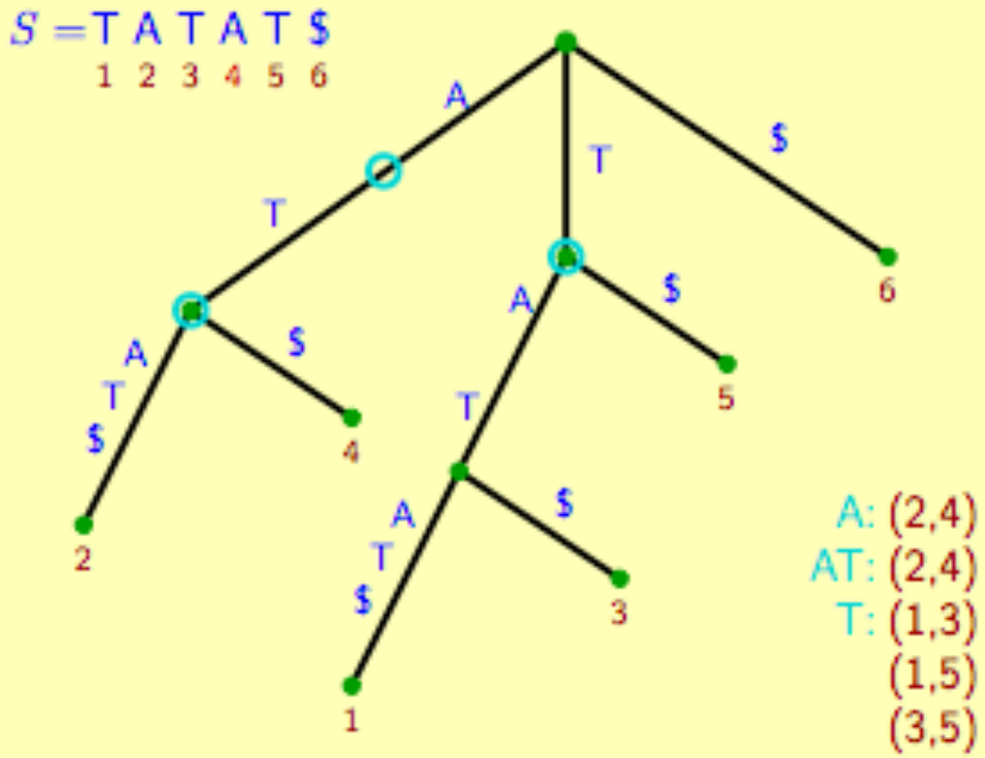
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



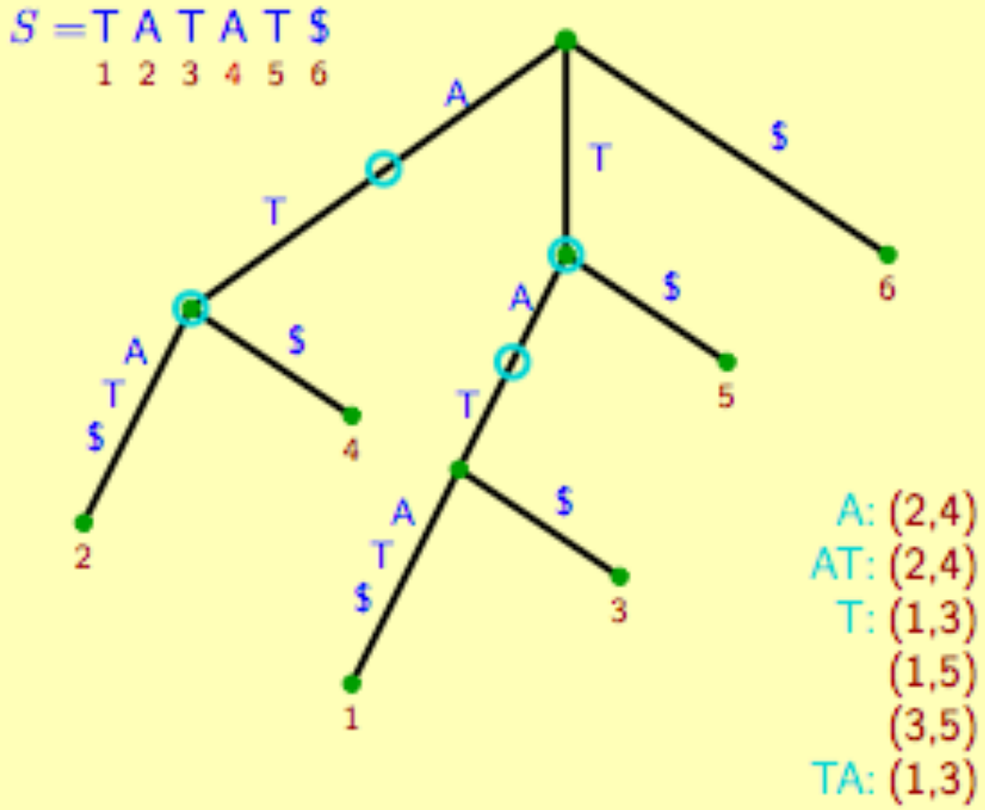
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



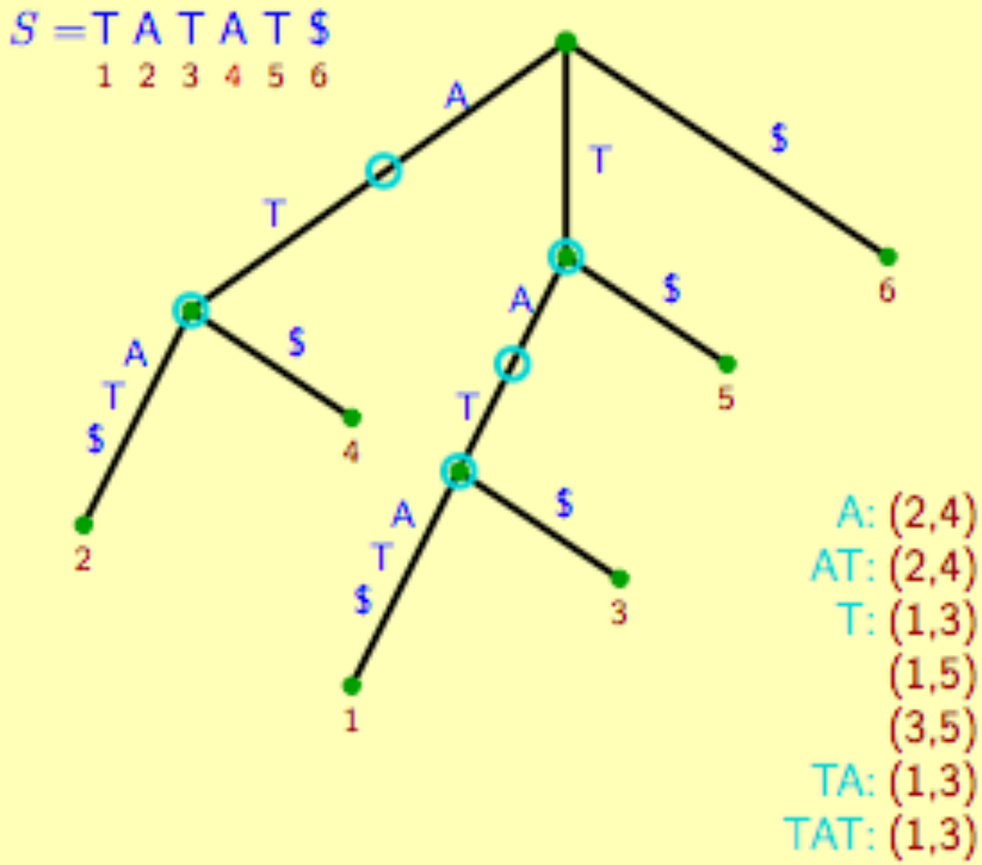
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



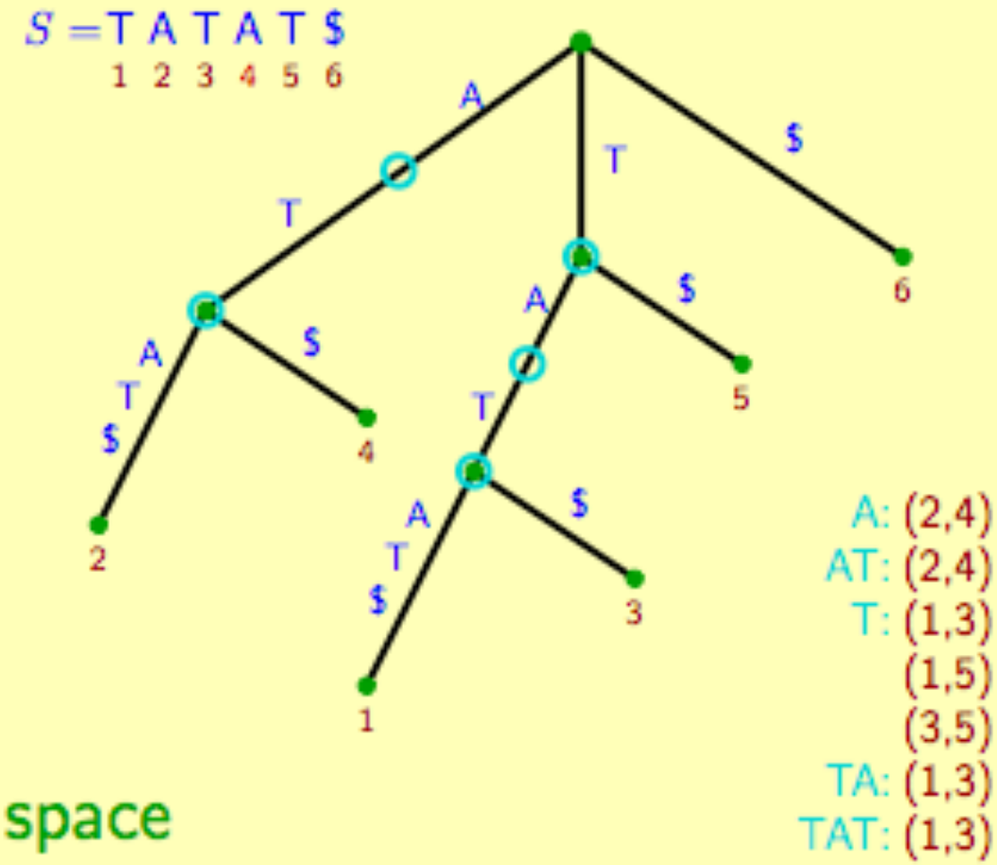
Finding exact repeats

Folklore: (see e.g. Gusfield, 1997)

- It is possible to find all pairs of repeated substrings (repeats) in S in linear time.

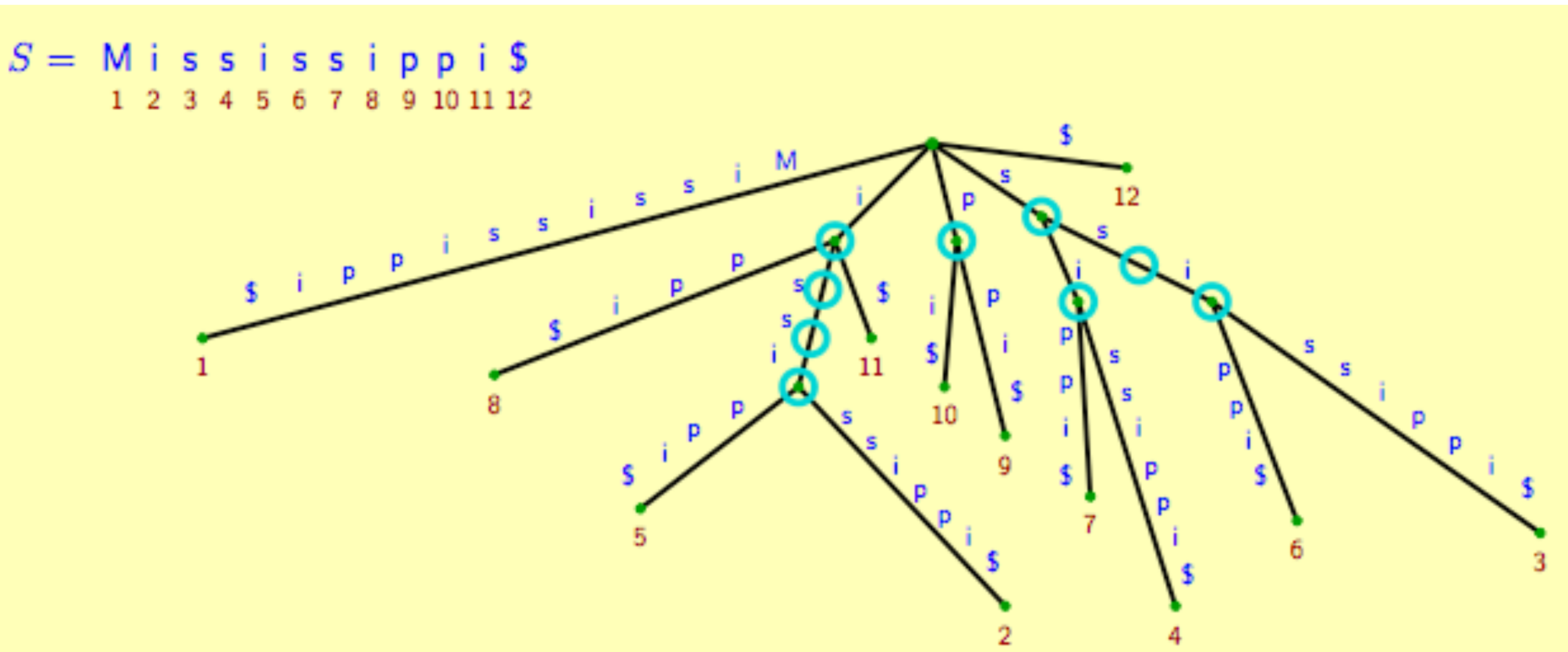
Idea:

- consider string S and its suffix tree $T(S)$.
- repeated substrings of S correspond to *internal locations* in $T(S)$.
- leaf numbers tell us positions where substrings occur.



Analysis: $\mathcal{O}(n+z)$ time with $z = |\text{output}|$, $\mathcal{O}(n)$ space

A larger example



i: (8,5)	is: (5,2)	p: (10,9)	s: (7,4)	si: (7,4)
(8,2)			(7,6)	
(8,11)	iss: (5,2)		(7,3)	ss: (6,3)
(5,2)			(4,6)	
(5,11)	issi: (5,2)		(4,3)	ssi: (6,3)
(2,11)			(6,3)	

Finding *maximal* exact repeats

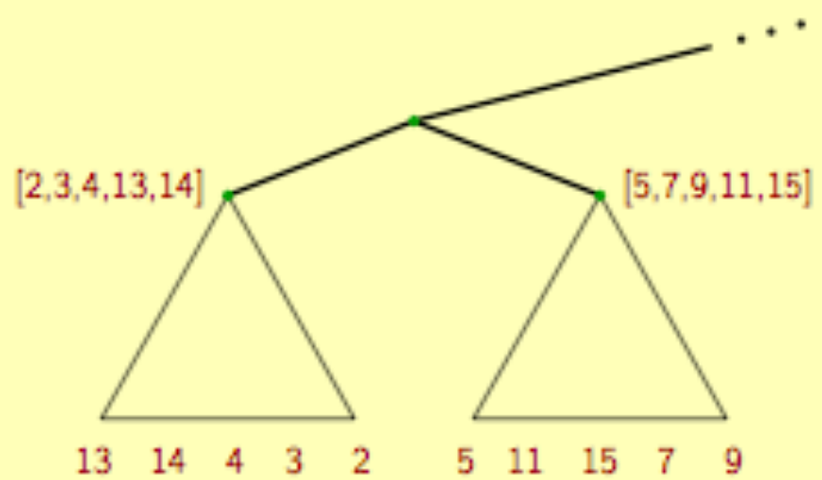
Finding *maximal* exact repeats

Finding *maximal* exact repeats

Finding *maximal* exact repeats

Idea:

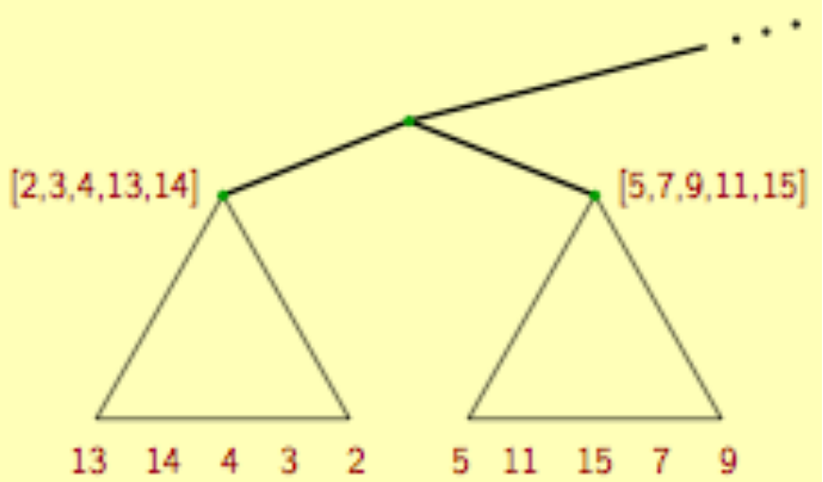
- For right-maximality ($X \neq Y$)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)



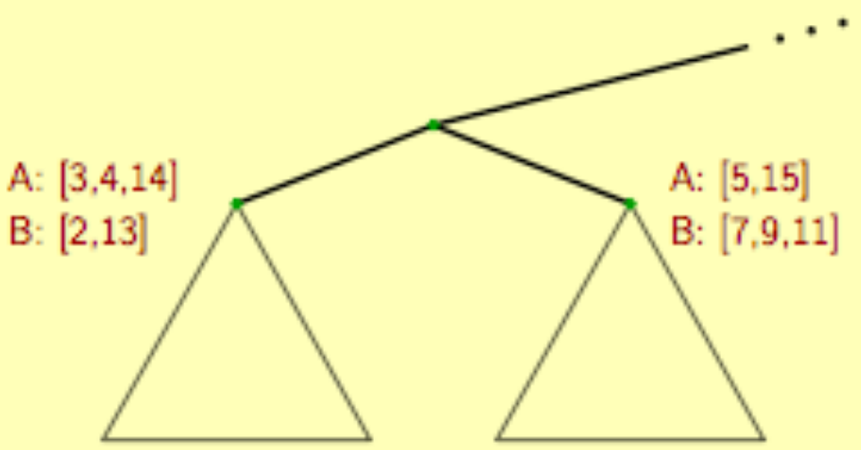
Finding *maximal* exact repeats

Idea:

- For right-maximality ($X \neq Y$)
  - consider only internal nodes of $T(S)$
  - report only pairs of leaves from different subtrees (or from different leaf-lists)



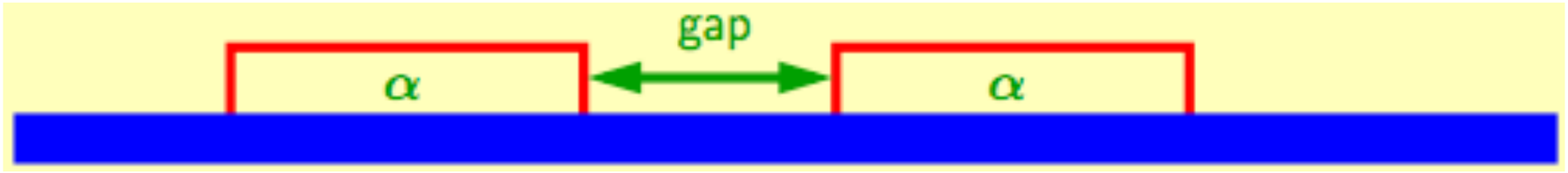
- For left-maximality ($A \neq B$)
  - keep lists for the different left-characters
  - report only pairs from different lists



Analysis: $\mathcal{O}(n+z)$ time with $z = |\text{output}|$, $\mathcal{O}(n)$ space

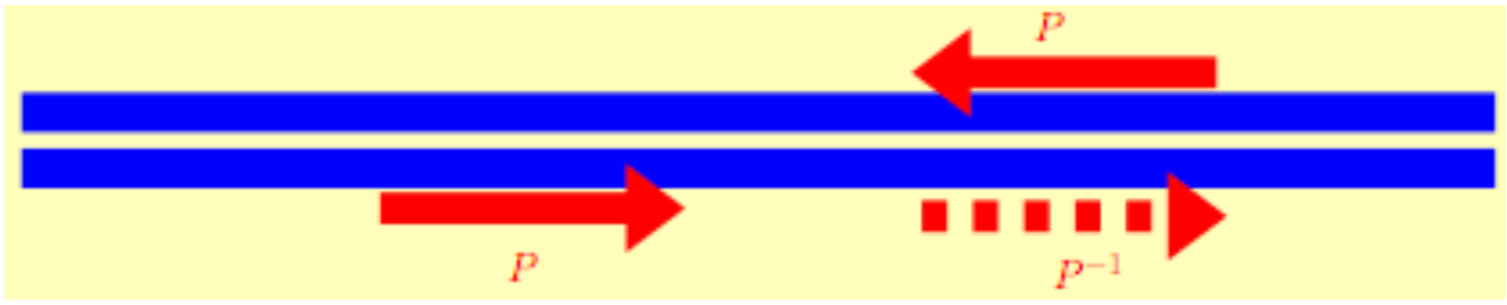
Other repeats

Maximal repeats with bounded gap in time O(n log n + z)



Tandem repeats in time O(n log n + z)

Palindromic repeats in O(n + z)



... all using suffix trees ...

More strings

The ***longest common substring*** of ***x*** and ***y*** is the longest string ***z*** which occurs in both ***x*** and ***y*** ...

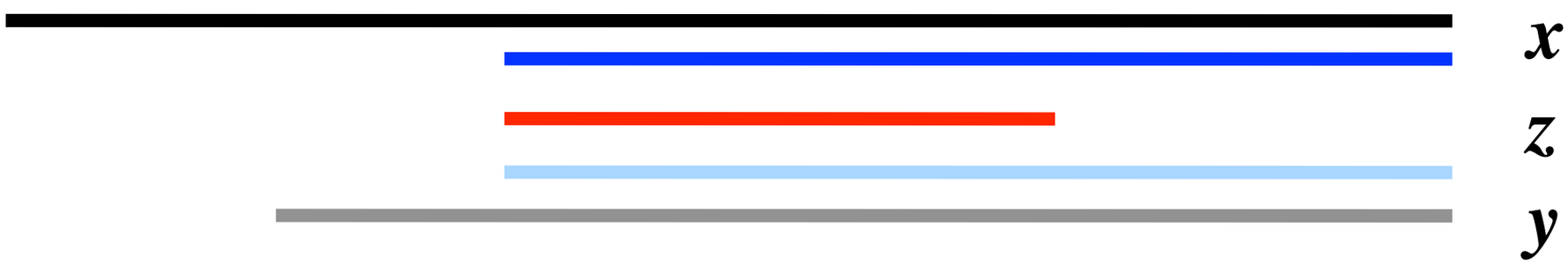
Can this be found efficiently using a suffix tree?

More strings

The ***longest common substring*** of ***x*** and ***y*** is the longest string ***z*** which occurs in both ***x*** and ***y*** ...

Can this be found efficiently using a suffix tree?

z is the longest common prefix of any pair of suffixes ***x***[*i*:*n*] and ***y***[*j*:*m*]



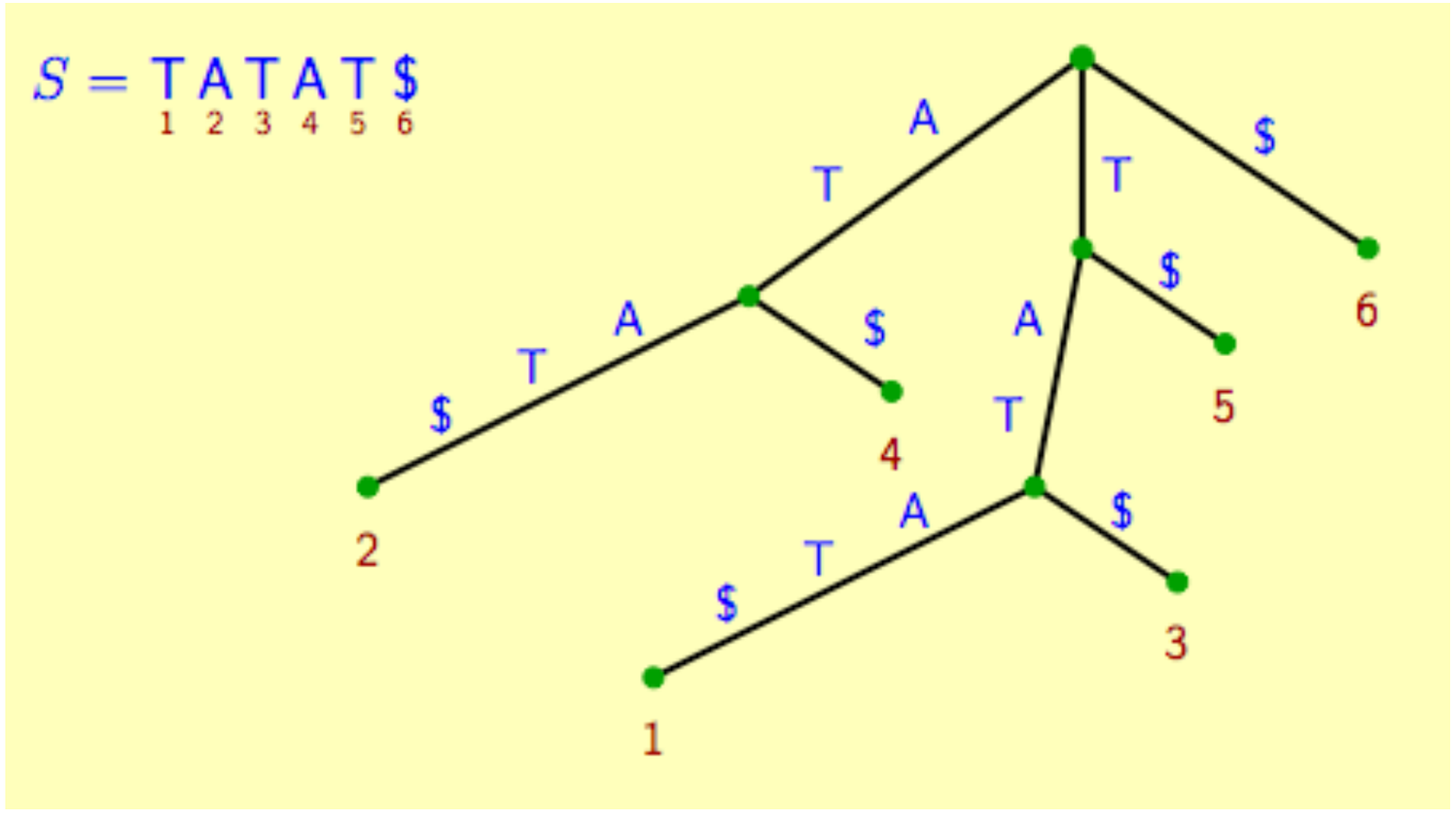
More strings

z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```



More strings

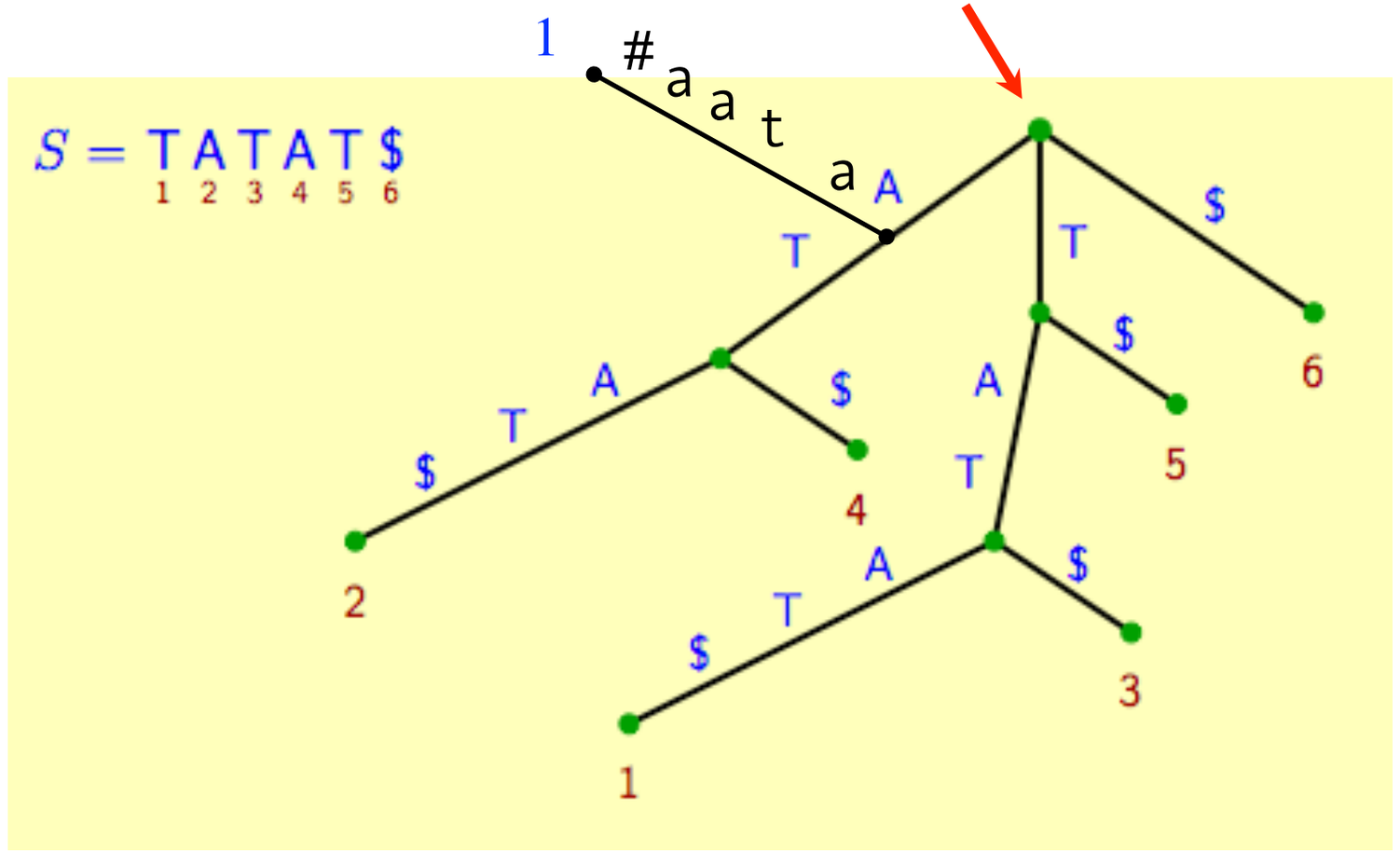
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```

←



More strings

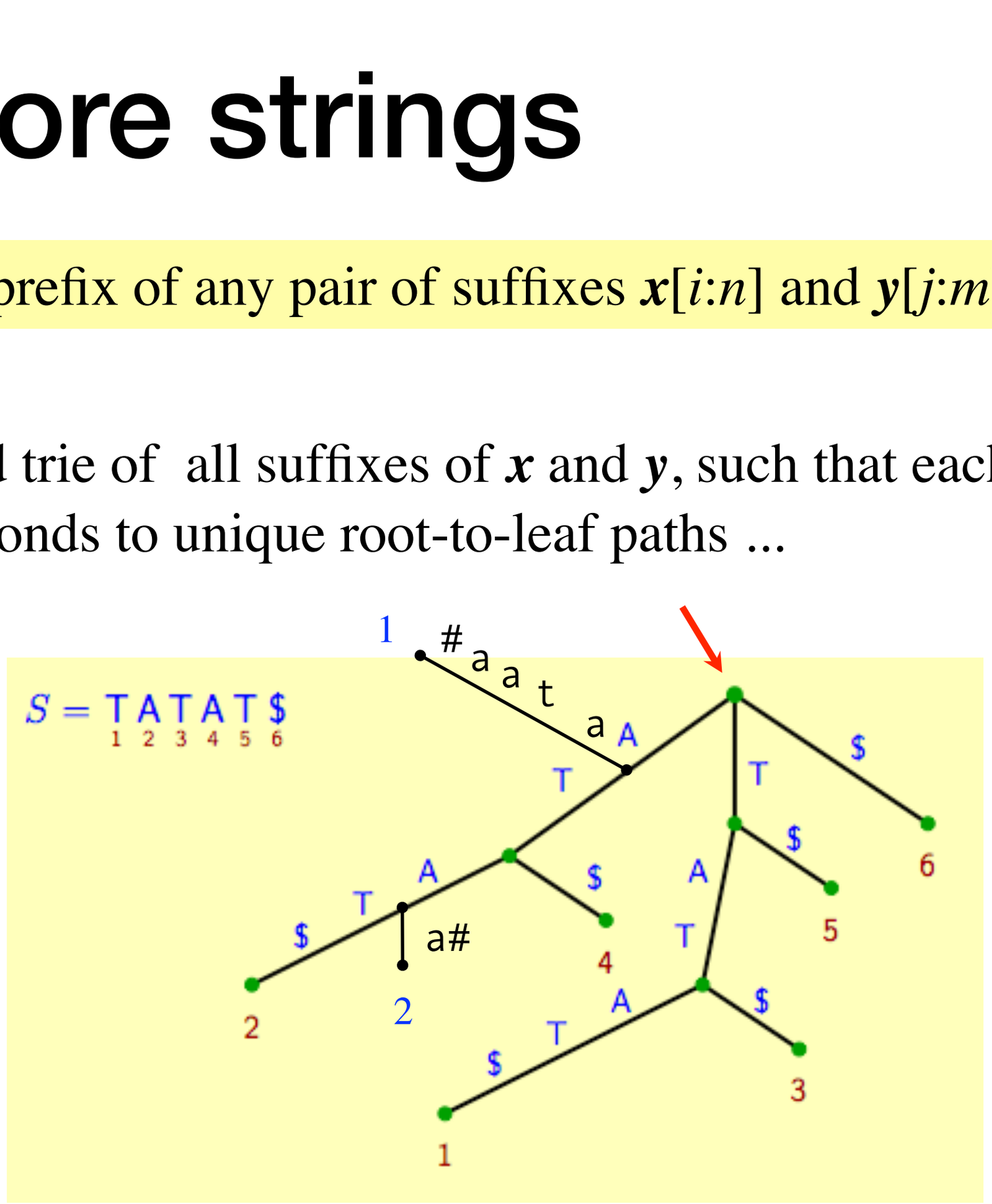
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```

←



More strings

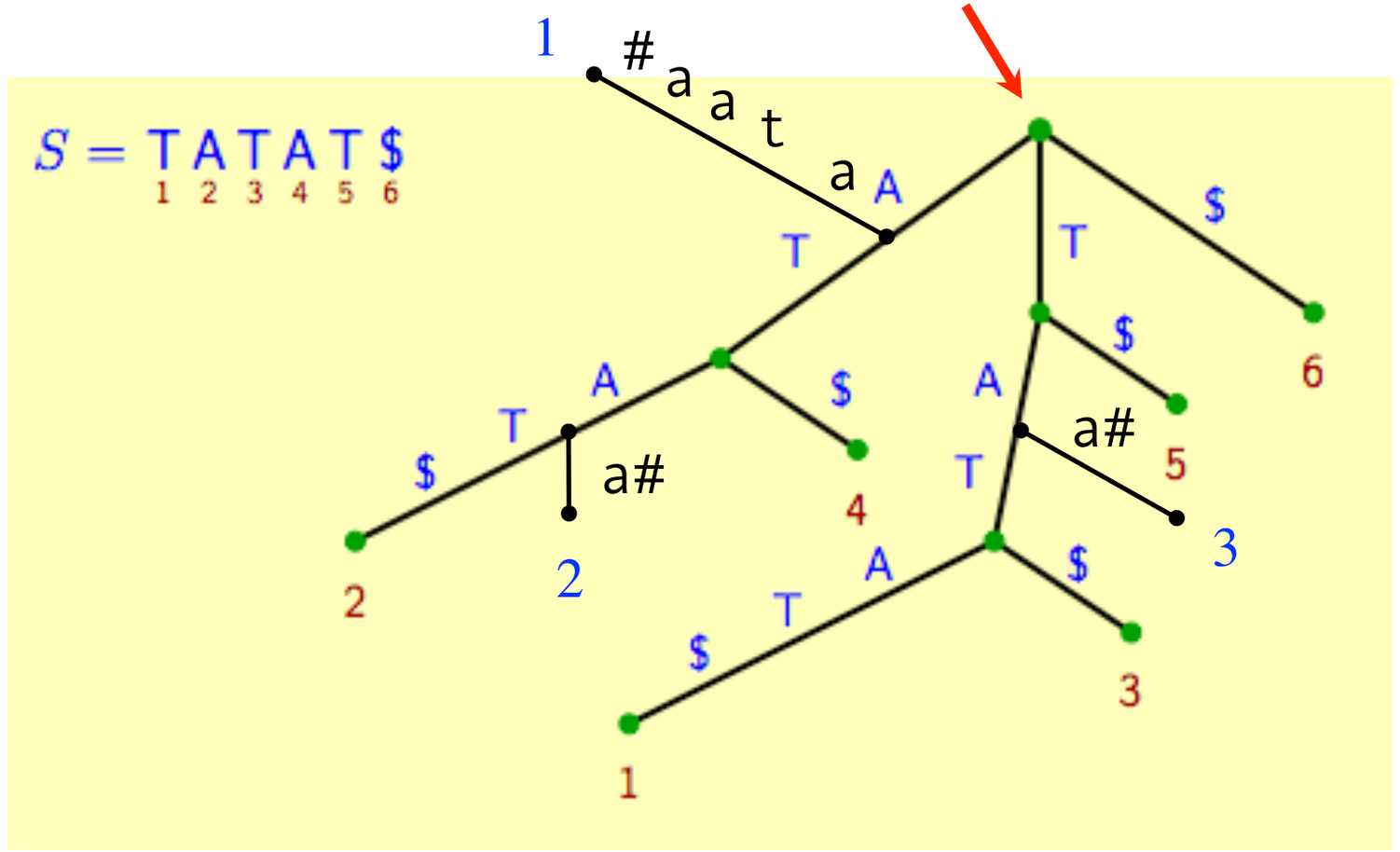
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```

←



More strings

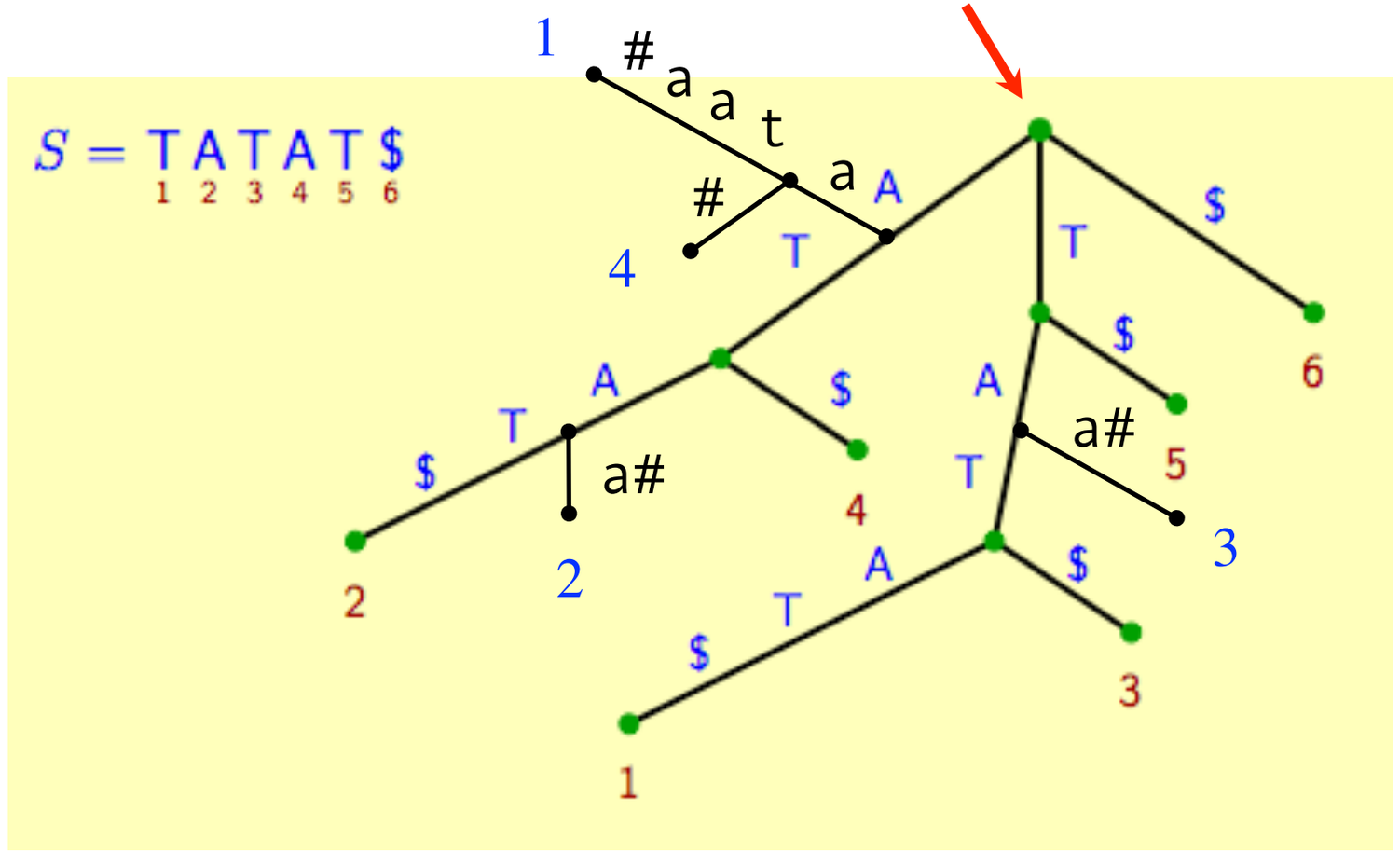
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```

←



More strings

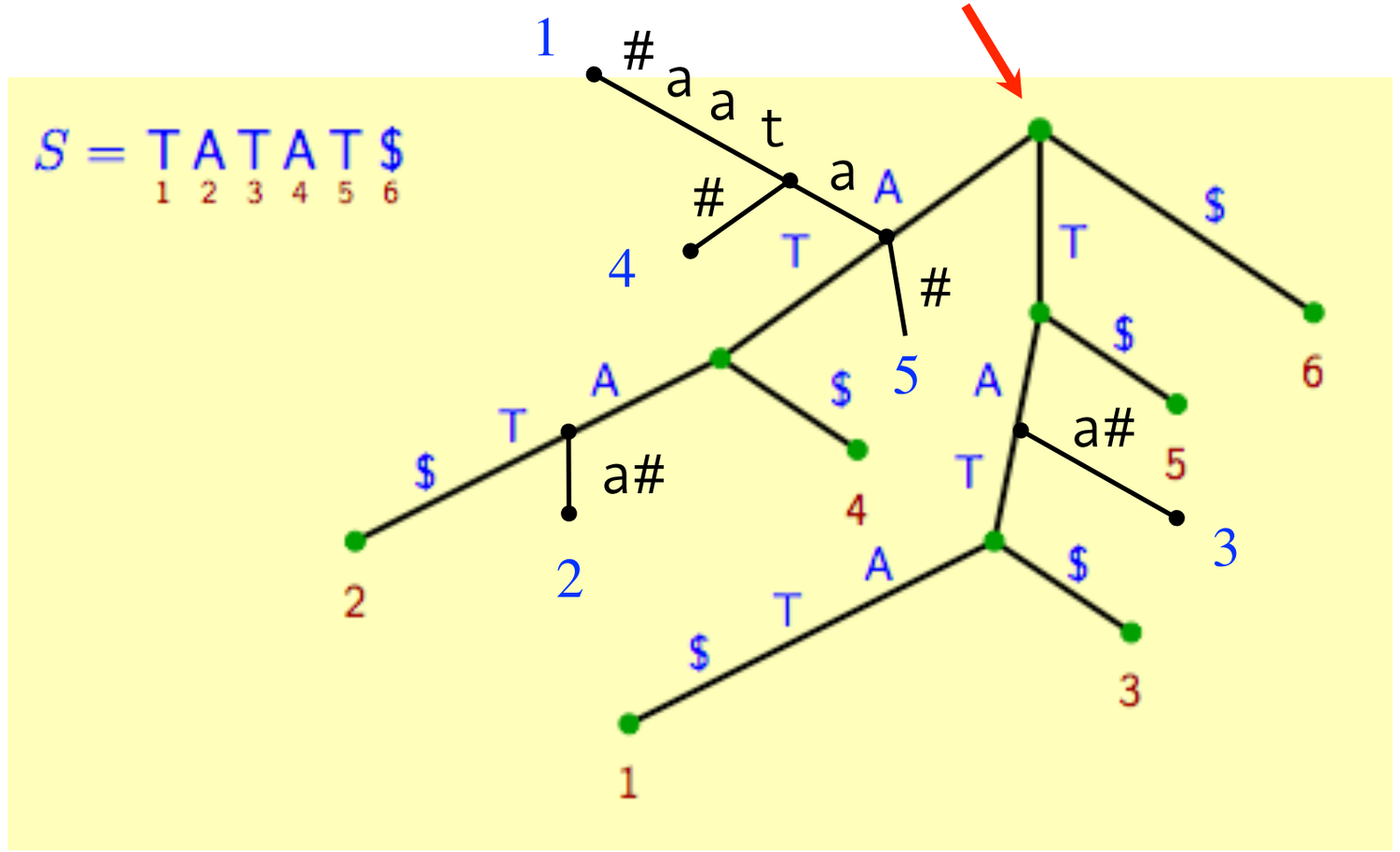
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
    ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
    ε#
```

←



More strings

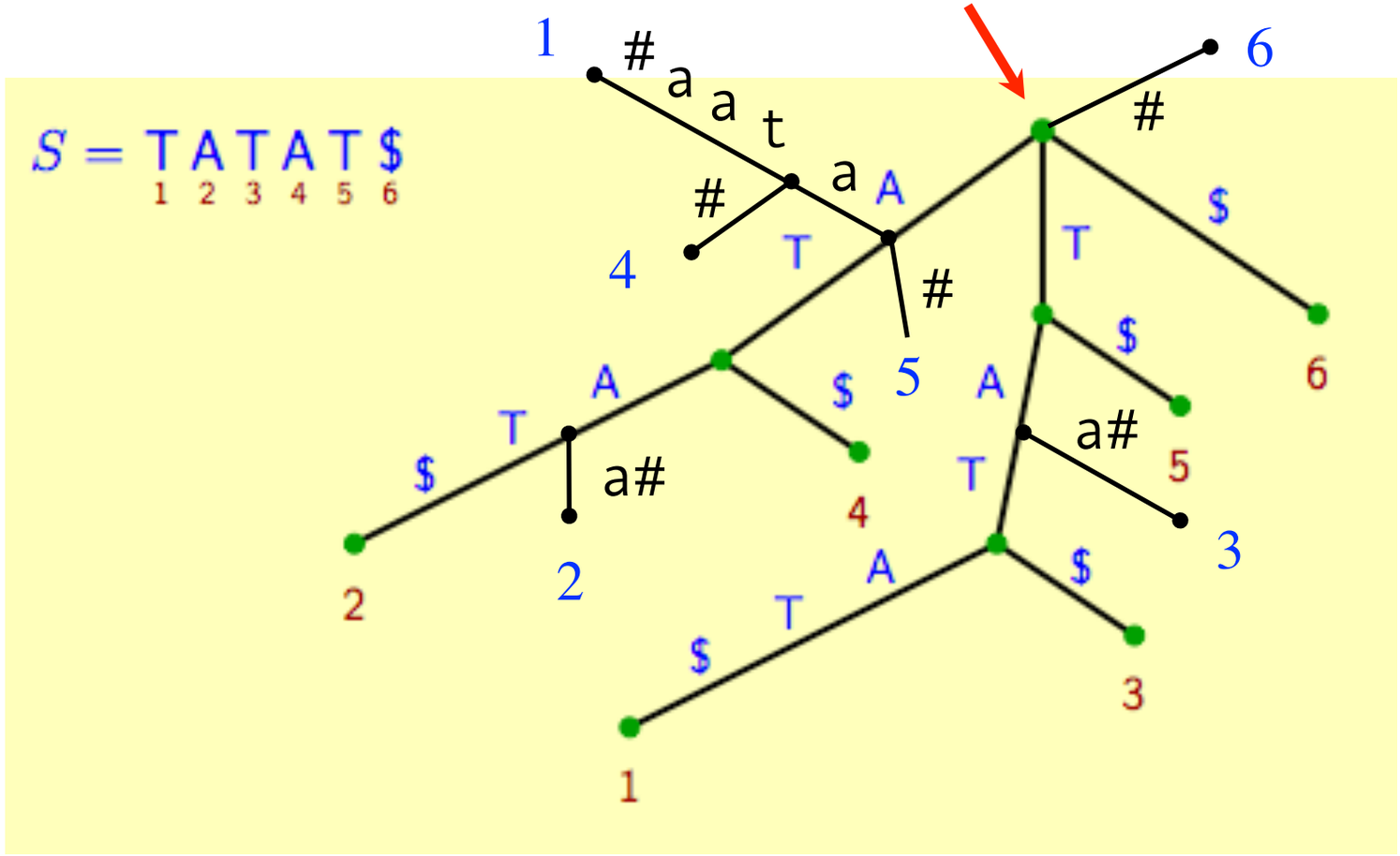
z is the longest common prefix of any pair of suffixes $\boldsymbol{x}[i{:}n]$ and $\boldsymbol{y}[j{:}m]$

Idea: Build a compacted trie of all suffixes of $\boldsymbol{x}$ and $\boldsymbol{y}$, such that each suffix of $\boldsymbol{x}$ and $\boldsymbol{y}$ corresponds to unique root-to-leaf paths ...

```
tatat$
 atat$
  tat$
   at$
    t$
     ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
     ε#
```

←



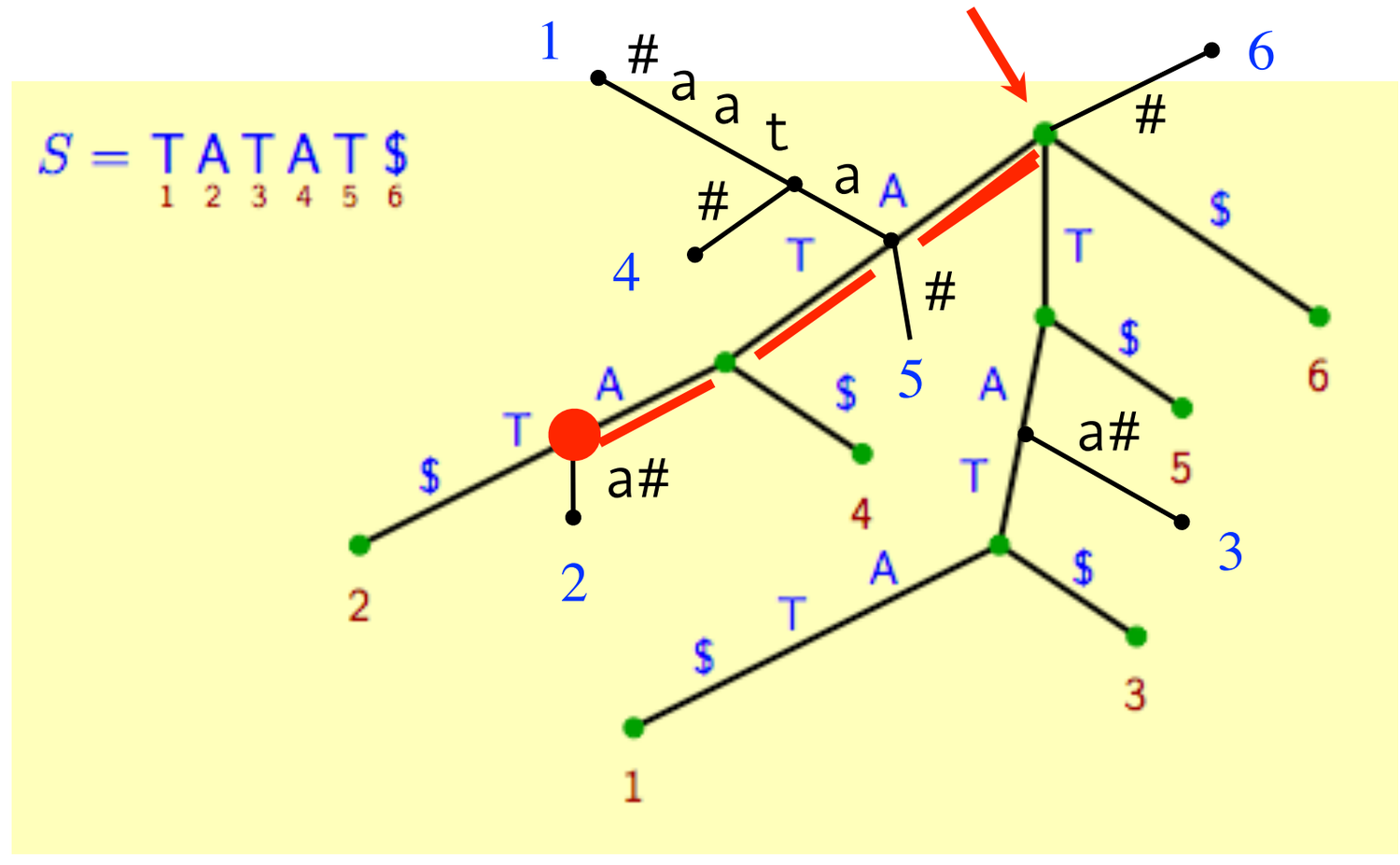
More strings

z is the longest common prefix of any pair of suffixes $x[i{:}n]$ and $y[j{:}m]$

```
tatat$
 atat$
  tat$
   at$
    t$
    ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
    ε#
```

$S = \underset{1}{T}\underset{2}{A}\underset{3}{T}\underset{4}{A}\underset{5}{T}\underset{6}{\$}$



Observe: z is the path-label of the deepest node with suffixes from both x and y as leaves in its sub-tree ...

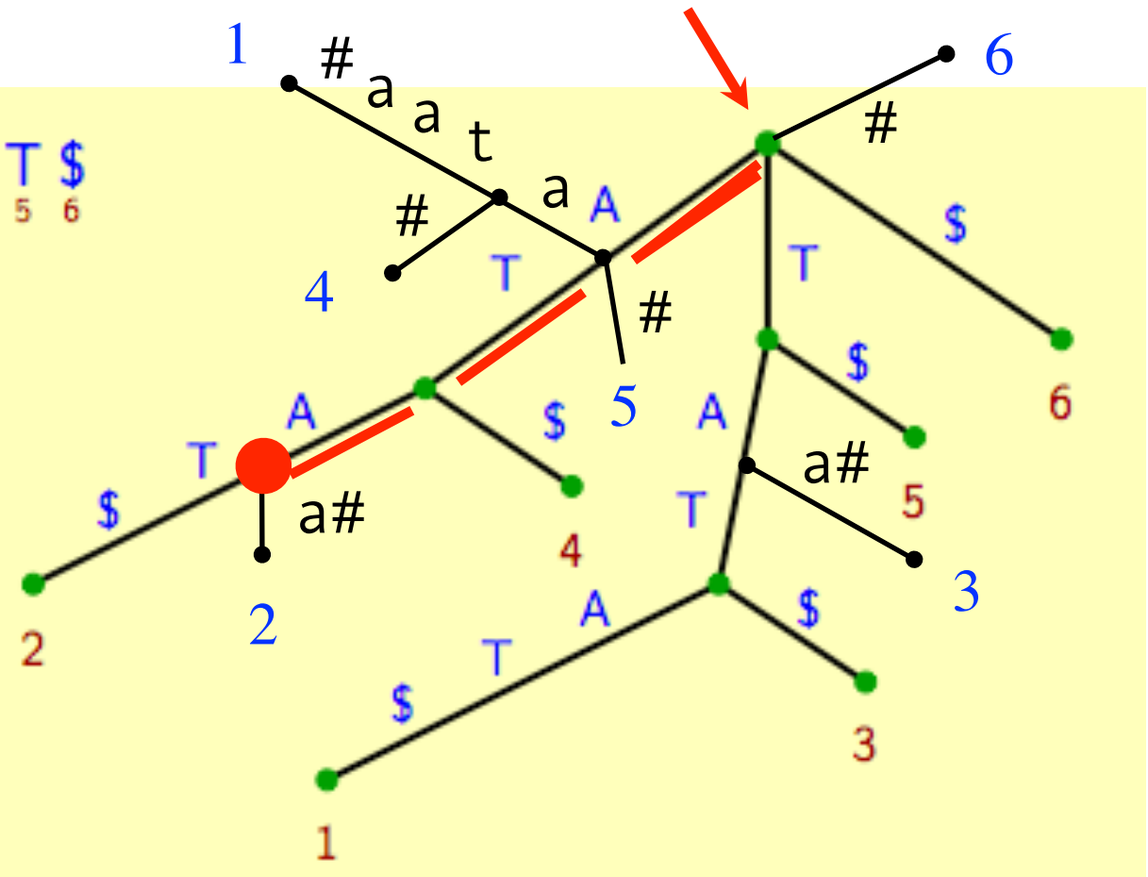
More strings

z is the longest common prefix of any pair of suffixes $x[i{:}n]$ and $y[j{:}m]$

```
tatat$
 atat$
  tat$
   at$
    t$
    ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
    ε#
```

$S = \underset{1}{T}\underset{2}{A}\underset{3}{T}\underset{4}{A}\underset{5}{T}\underset{6}{\$}$



Observe: z is the path-label of the deepest node with suffixes from both x and y as leaves in its sub-tree ... **Time**: O(n+m)

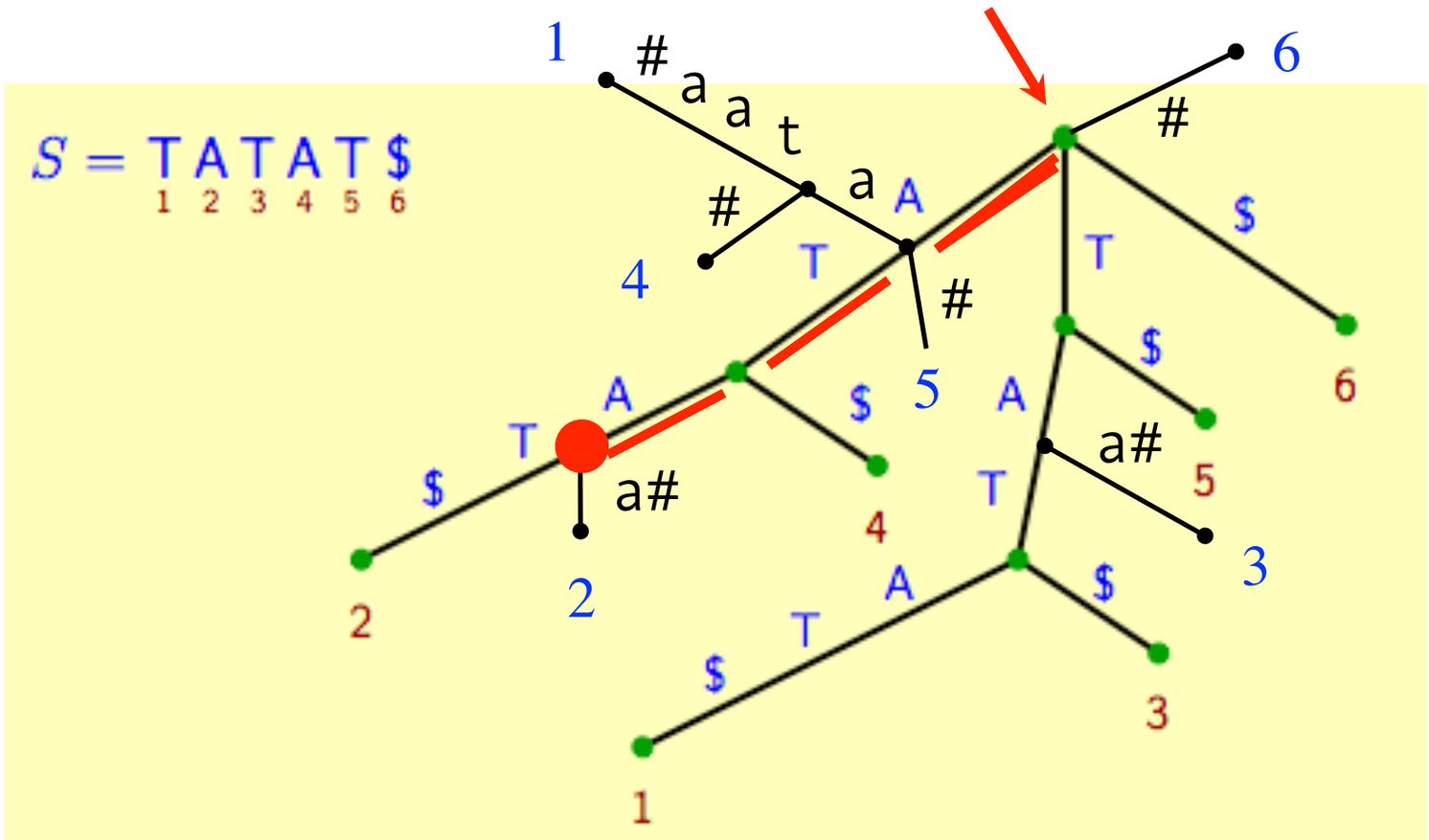
Generalized suffix tree

This is the **generalized suffix tree** of `tatat` and `aataa`

```
tatat$
 atat$
  tat$
   at$
    t$
    ε$
```

```
aataa#
 ataa#
  taa#
   aa#
    a#
    ε#
```

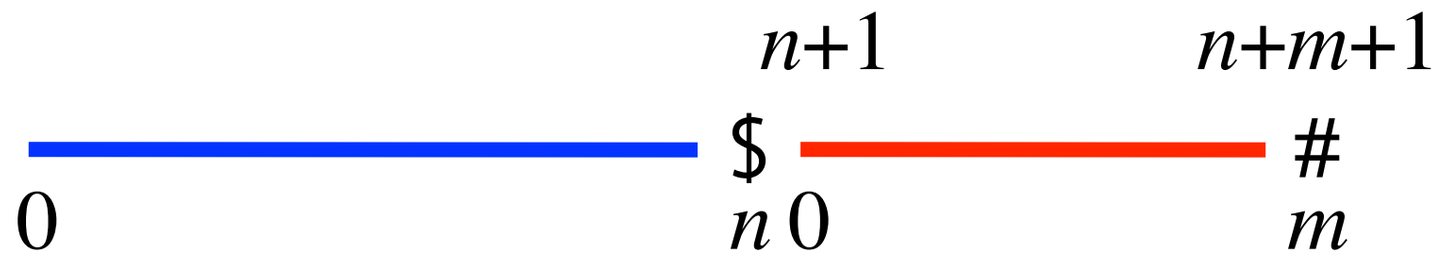
$S = \underset{1\ 2\ 3\ 4\ 5\ 6}{\mathrm{TATAT\$}}$



Can be constructed by constructing the suffix tree of ...

`tatat$aataa#`

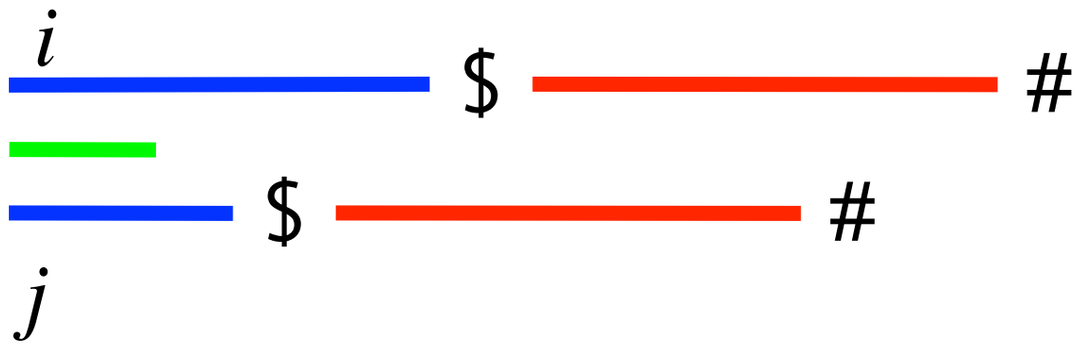
Generalized suffix tree



... we must argue that we get the same branching structure ...

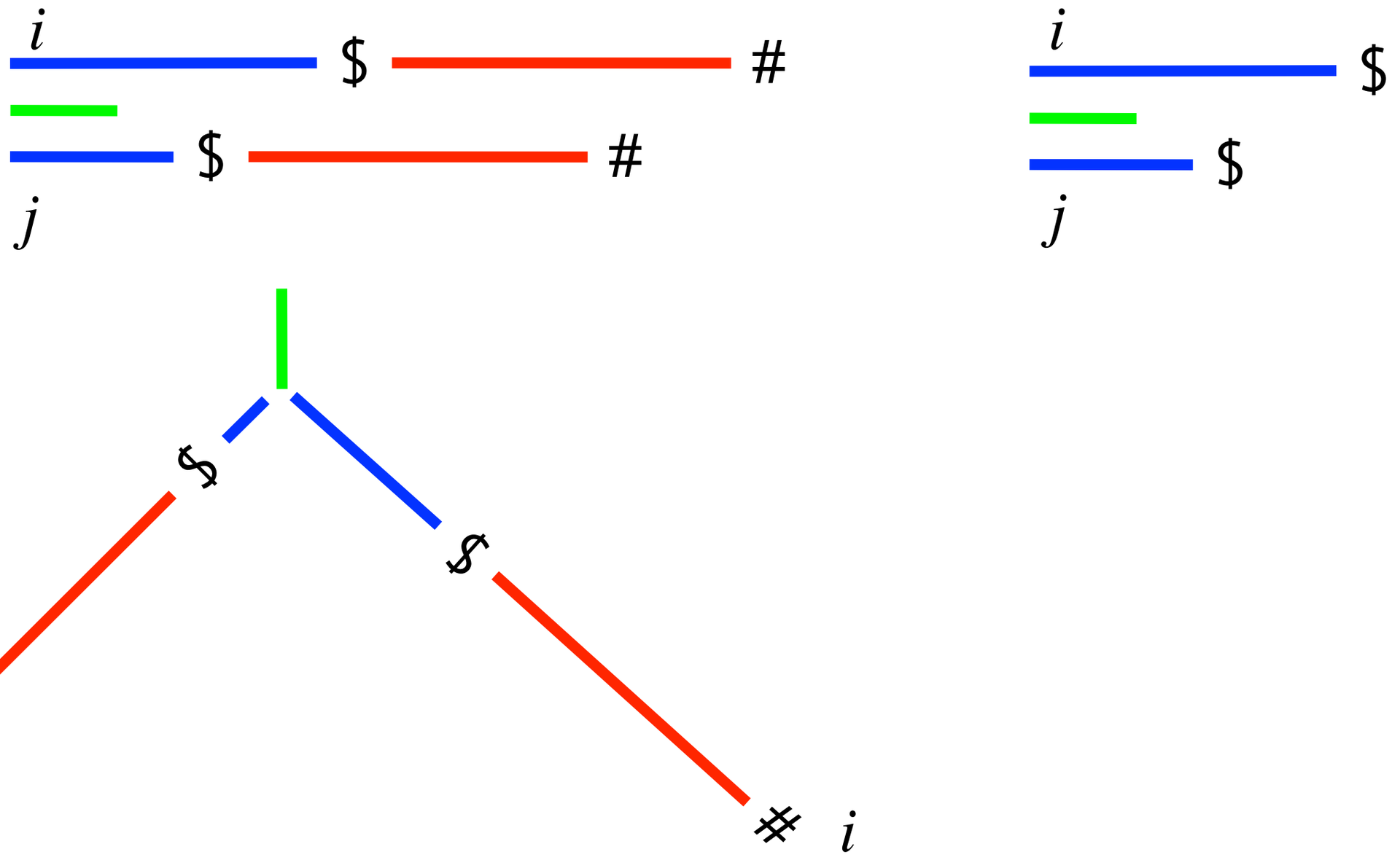
Generalized suffix tree

Case 1:



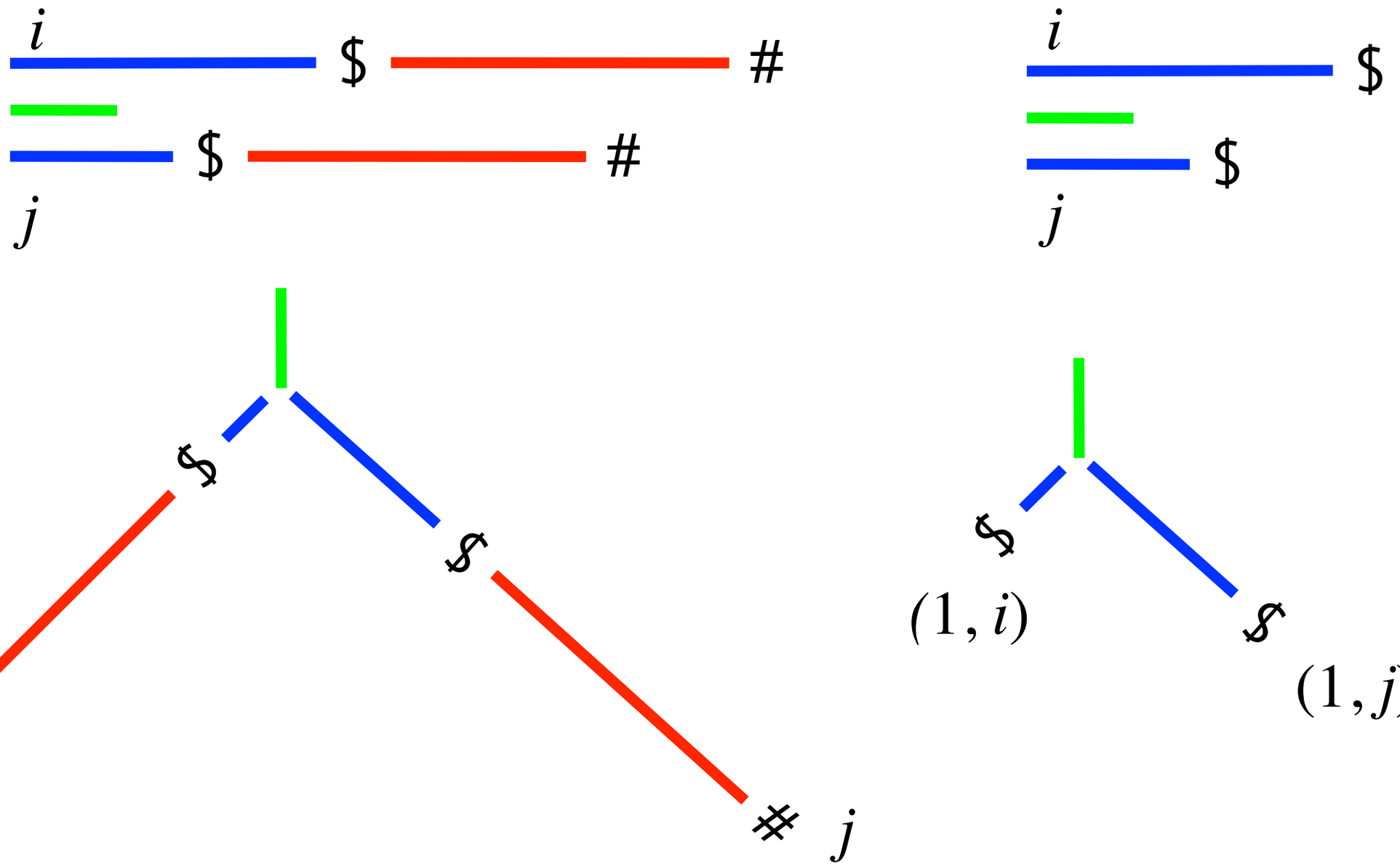
Generalized suffix tree

Case 1:



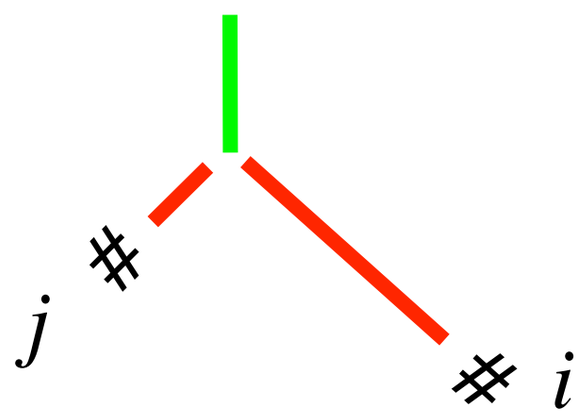
Generalized suffix tree

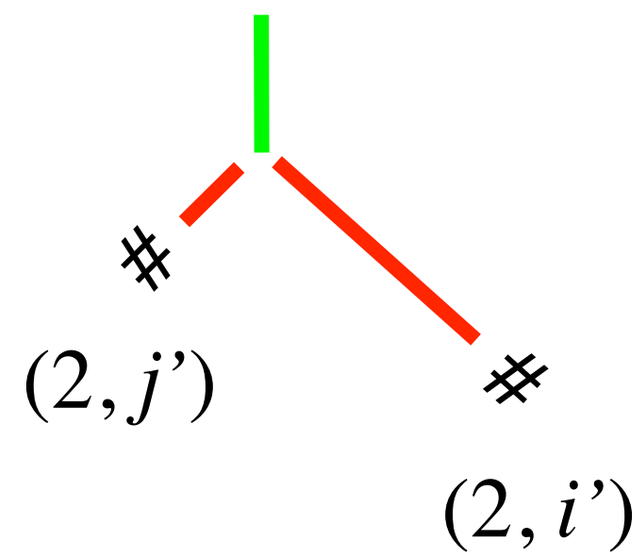
Case 1:



Generalized suffix tree

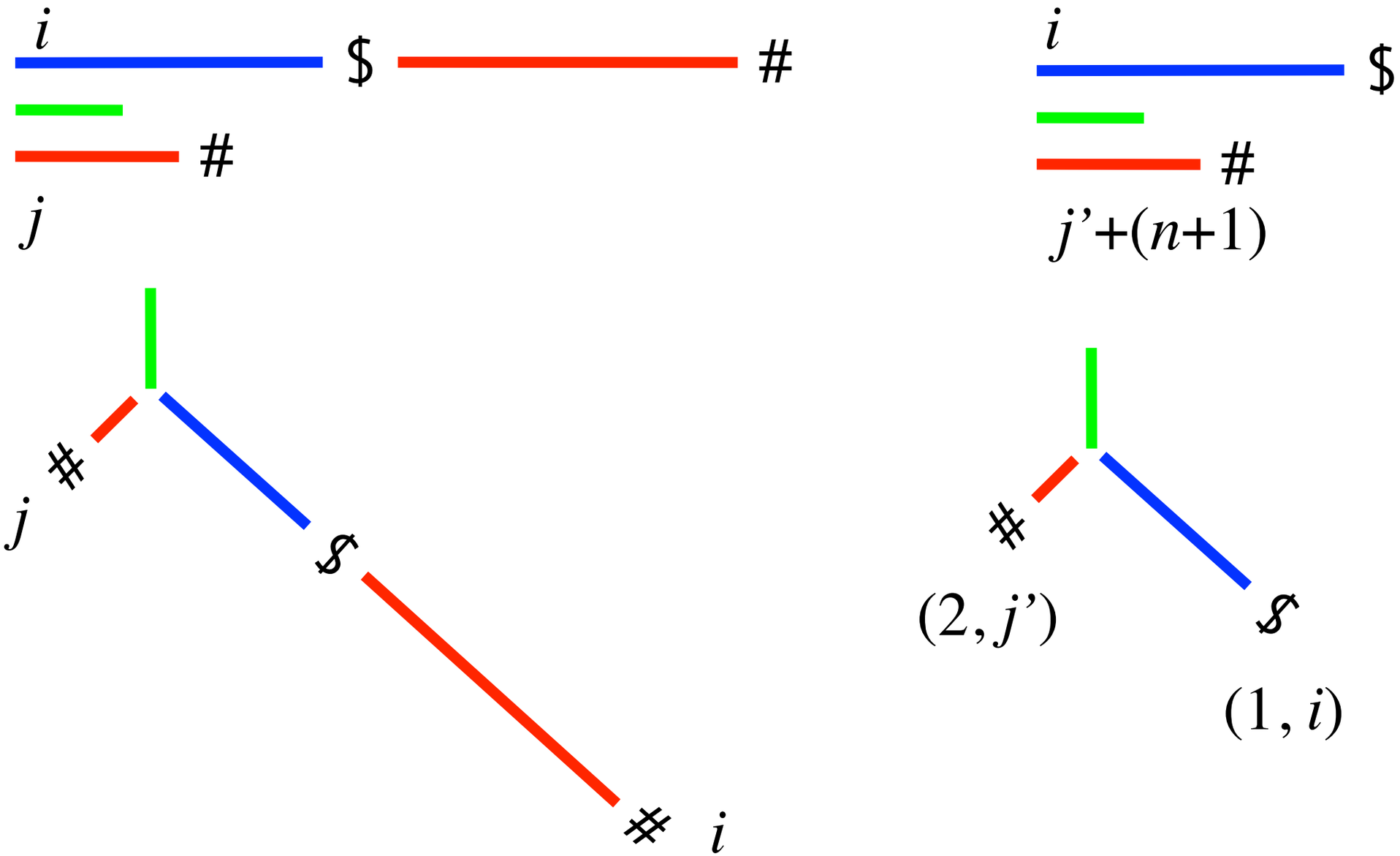
Case 2:





Generalized suffix tree

Case 3:

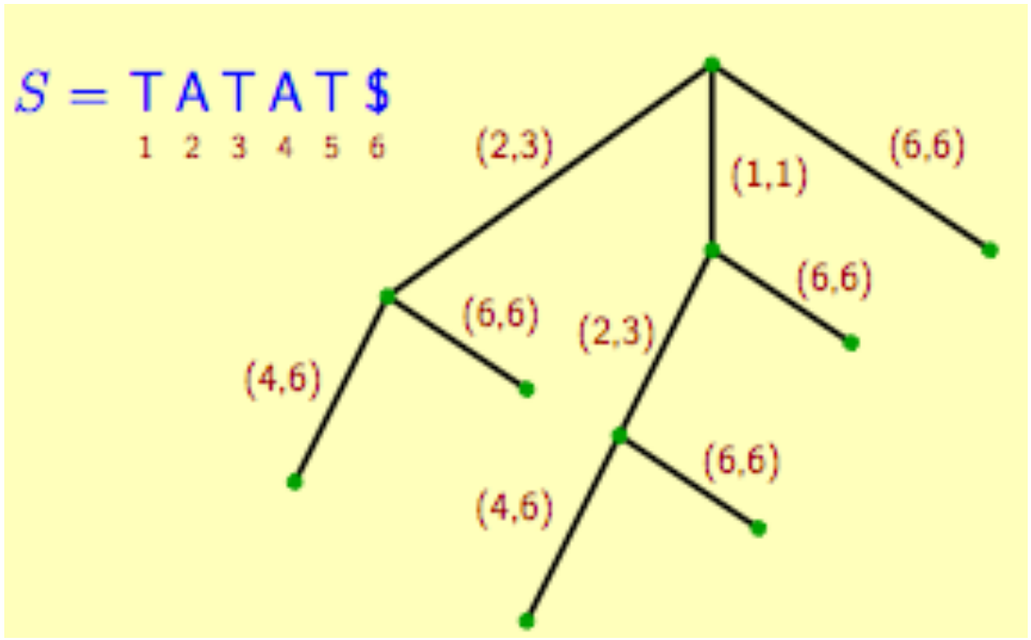


Is everything great?

What are the caveats of suffix trees?

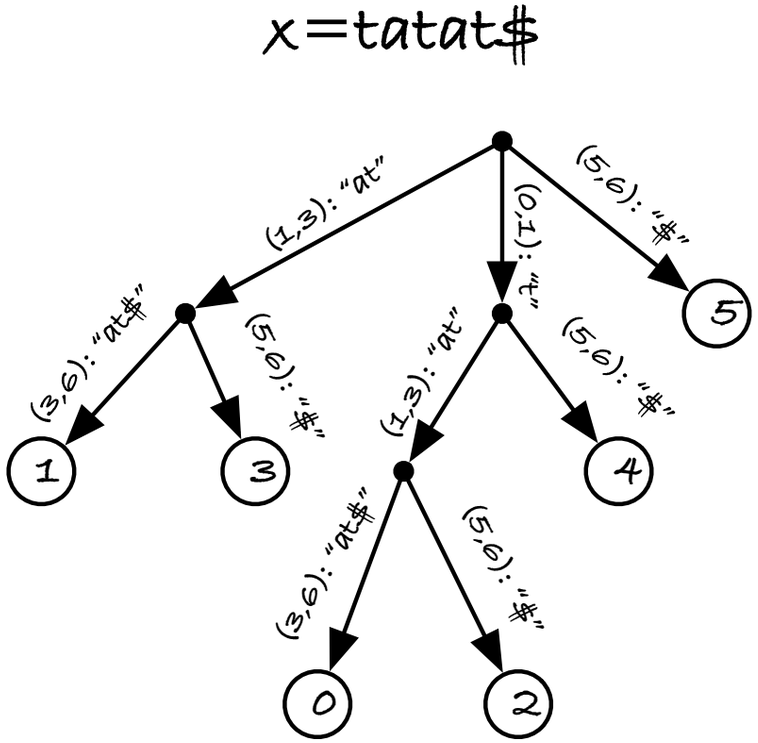
Space consumption

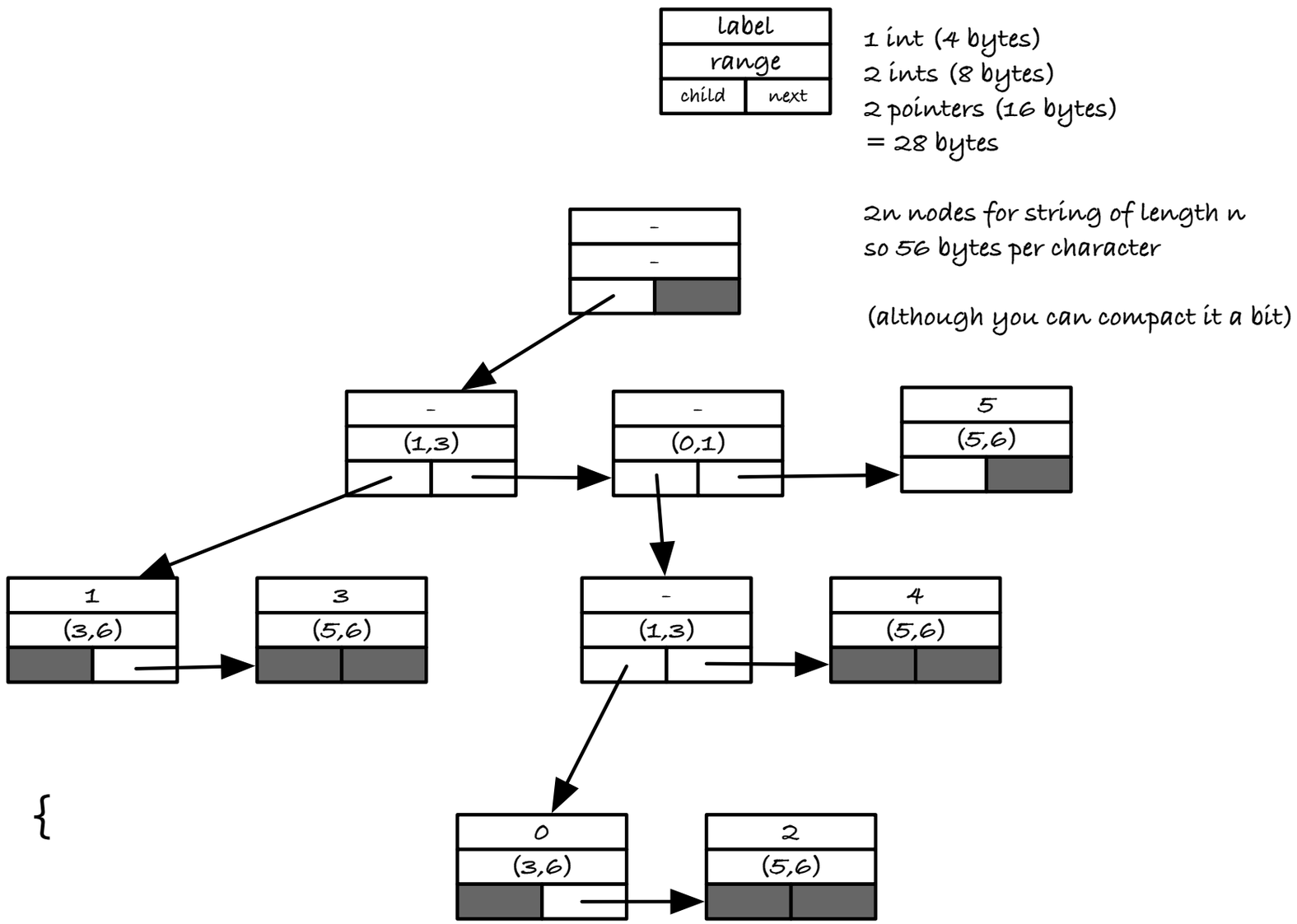
Fact: $T(\boldsymbol{x})$ requires $O(n)$ space, where $n=|\boldsymbol{x}|$



... but how much space does it consume in “practice”?

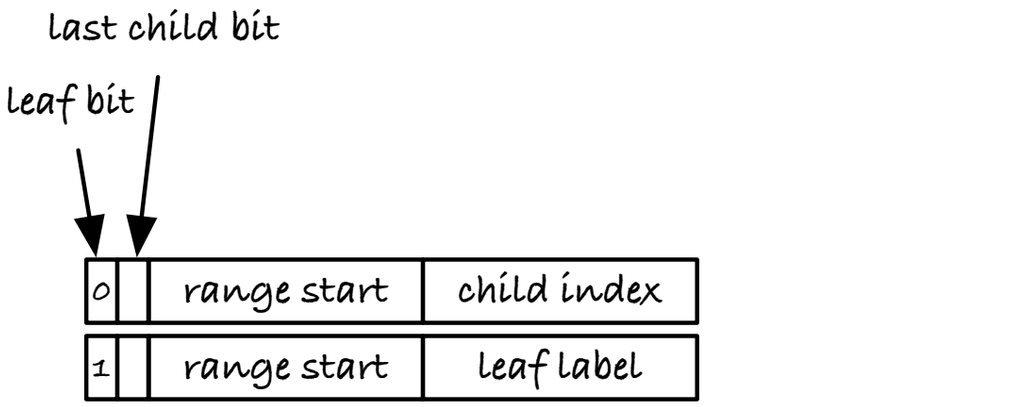
Linked list representation



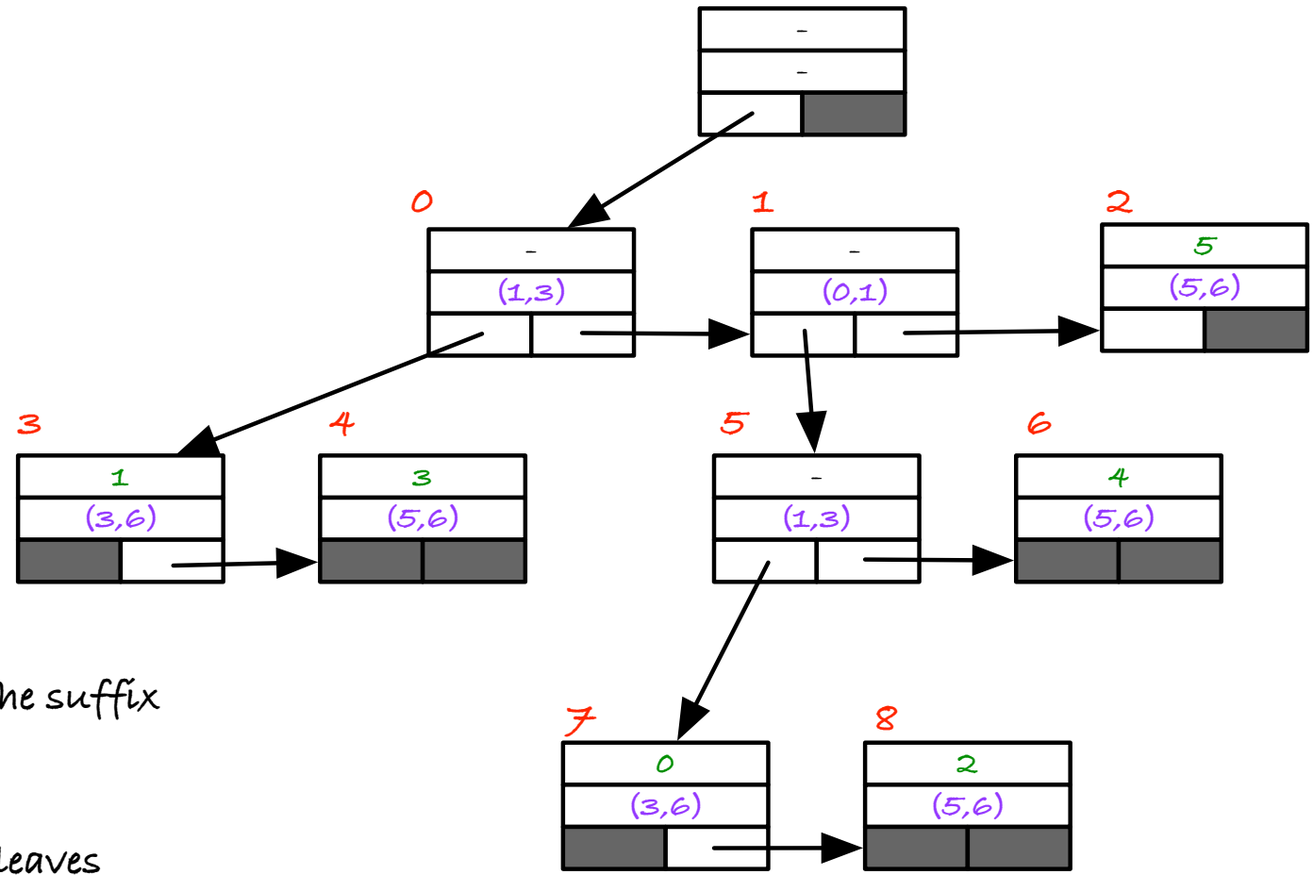



```
struct suffix_tree_node {
    int leaf_label;
    int range_from, range_to;
    struct suffix_tree_node *next;
    struct suffix_tree_node *child;
};
```

Compaction is possible

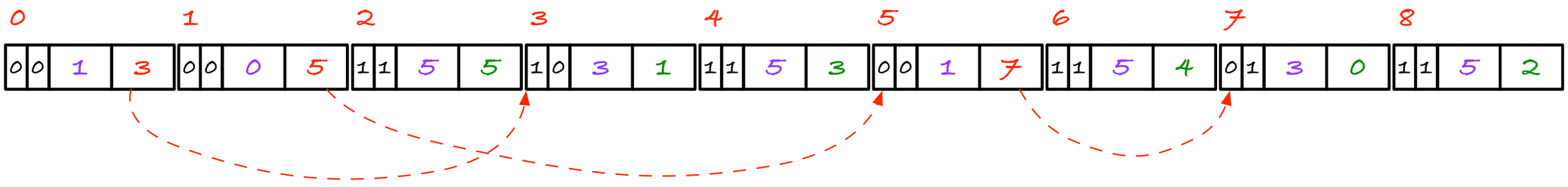


E.g. 64 bit integer (31 bits for range + child/label)
8 bytes per node, or 16 bytes per character.



Get range end from first child (always possible to have a child where you can get it, but you must construct the suffix tree to ensure it, not all algorithms will give you this.

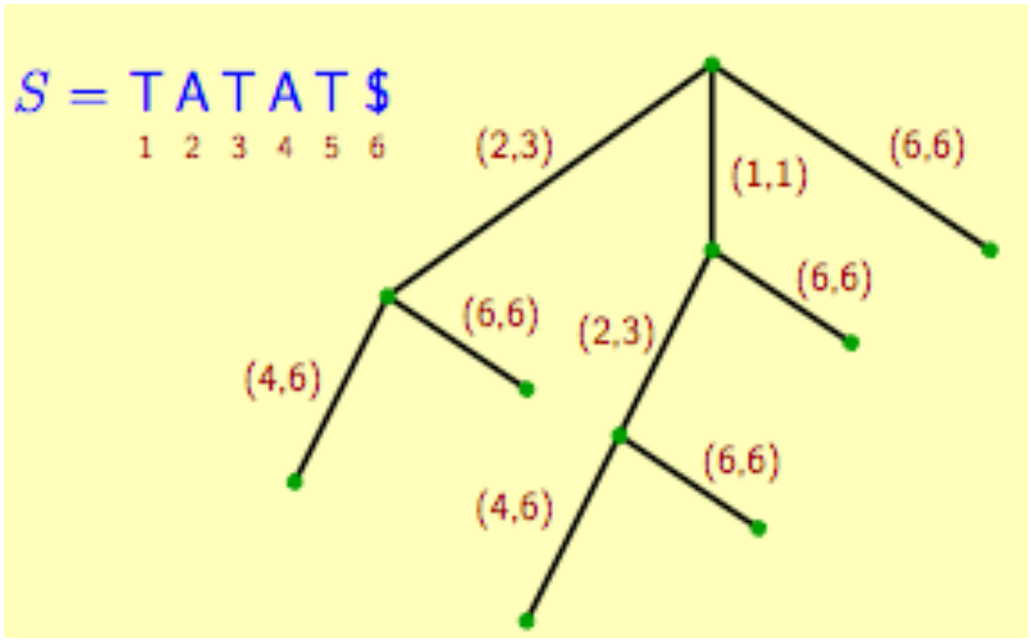
The range end of a leaf is always the end of the string (all leaves end at the end of the string) so you don't need a child there.



(show code)

Space consumption

Fact: $T(\boldsymbol{x})$ requires O(n) space, where n=|$\boldsymbol{x}$|, but ....

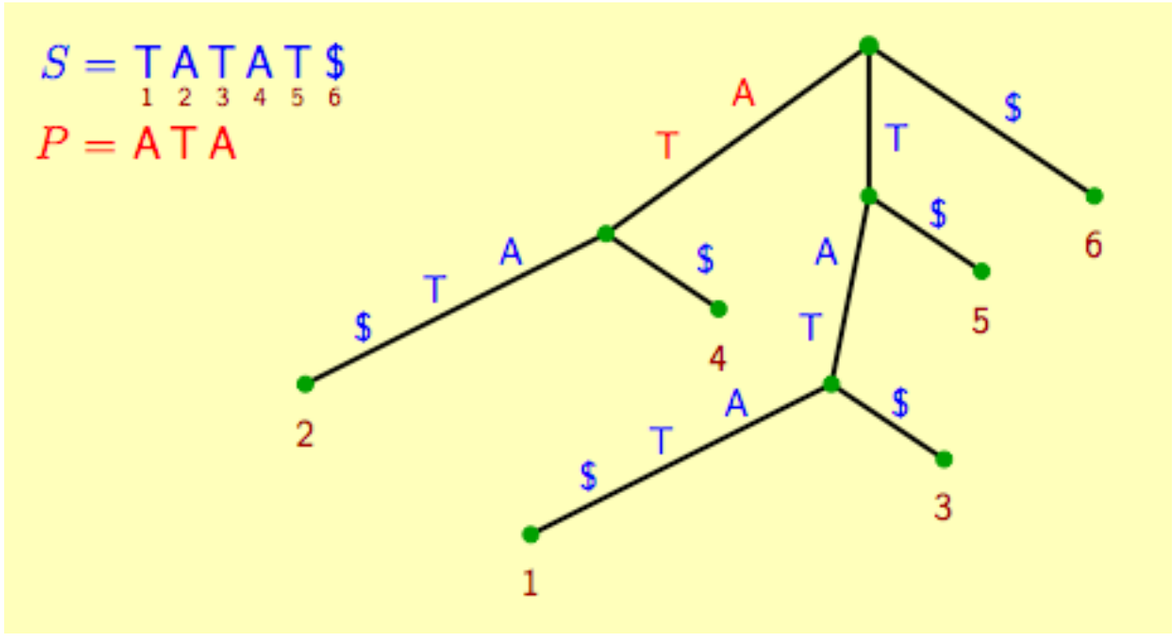


... in practice somewhere between 10 and 60 bytes per letter in $\boldsymbol{x}$ …
(depending on representation)

Is this a problem? Depends on n, if ≈500,000,000 then yes…
(and then we have other ways to represent a string)

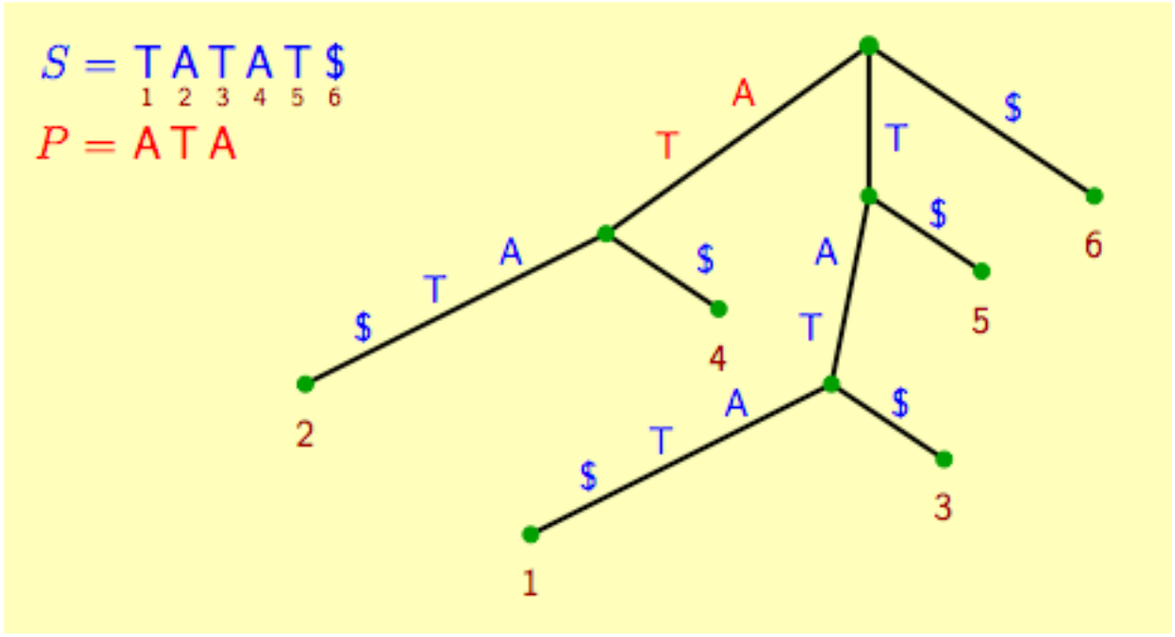
Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?



Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?



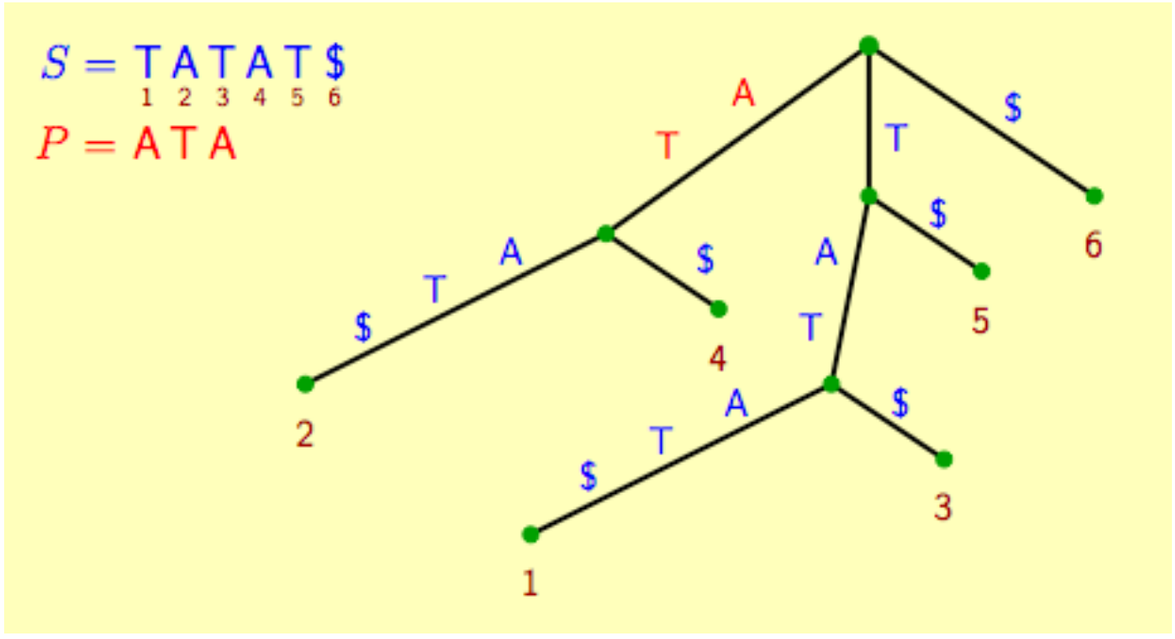
Time proportional to the out-degree of the node ≤ σ...

... search time in "practice" could be O(σ·*m*) ...

If σ is large, e.g. 256, this matters!!

Alphabet size

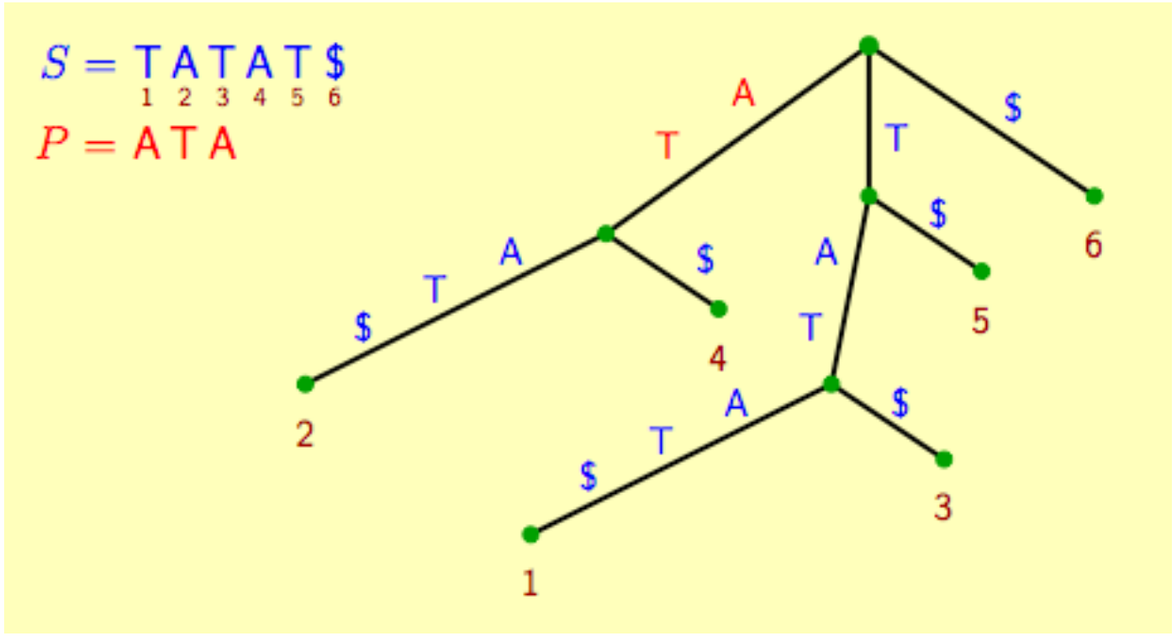
How much time does it take to find the proper edge out from a node when searching in a suffix tree?



Idea 1: Organising children in a search-tree, reduces search time from σ to O(log σ) ... (requires an ordered alphabet)

Alphabet size

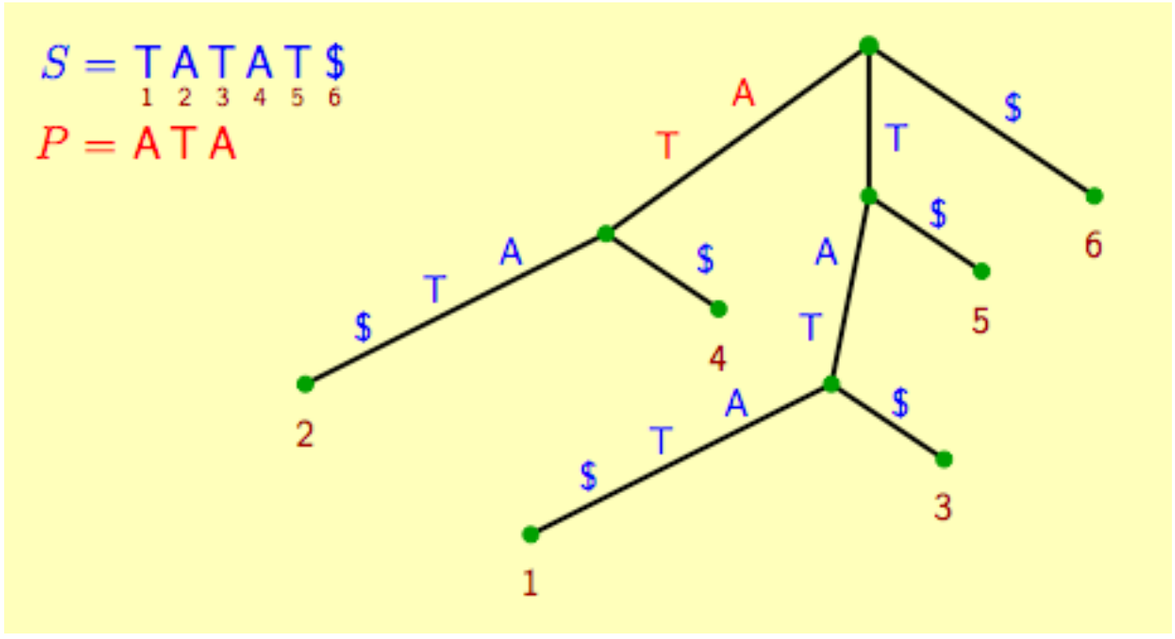
How much time does it take to find the proper edge out from a node when searching in a suffix tree?



Idea 2: Organising children in an array of size σ indexed by letters, reduces search time from σ to O(1) ... (requires a finite alphabet)

Alphabet size

How much time does it take to find the proper edge out from a node when searching in a suffix tree?



Idea 3: Use some other dictionary for mapping letters to children ...

... the alphabet size matters in practice …
(but for our readmapper, σ=4 (or 5 with $) so we won't worry much)

Enough about applications…

- Next week, we learn how to construct suffix trees in linear time!

That's all Folks!