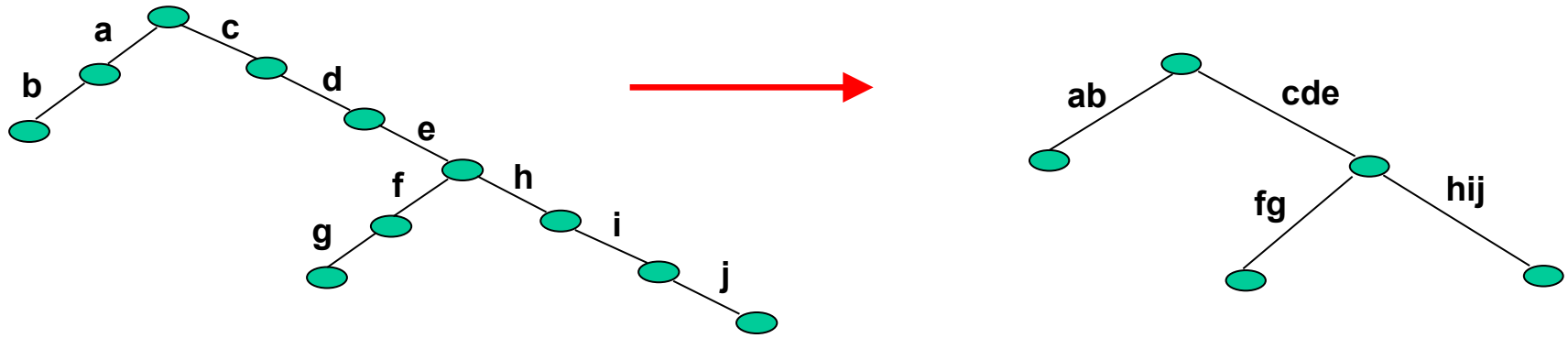
Suffix trees

A *suffix tree* for a string S can be derived from a suffix trie by

- collapsing each path consisting of *unary* branch nodes into a single edge
- labelling each new edge with the concatenation of the labels from the corresponding trie edges

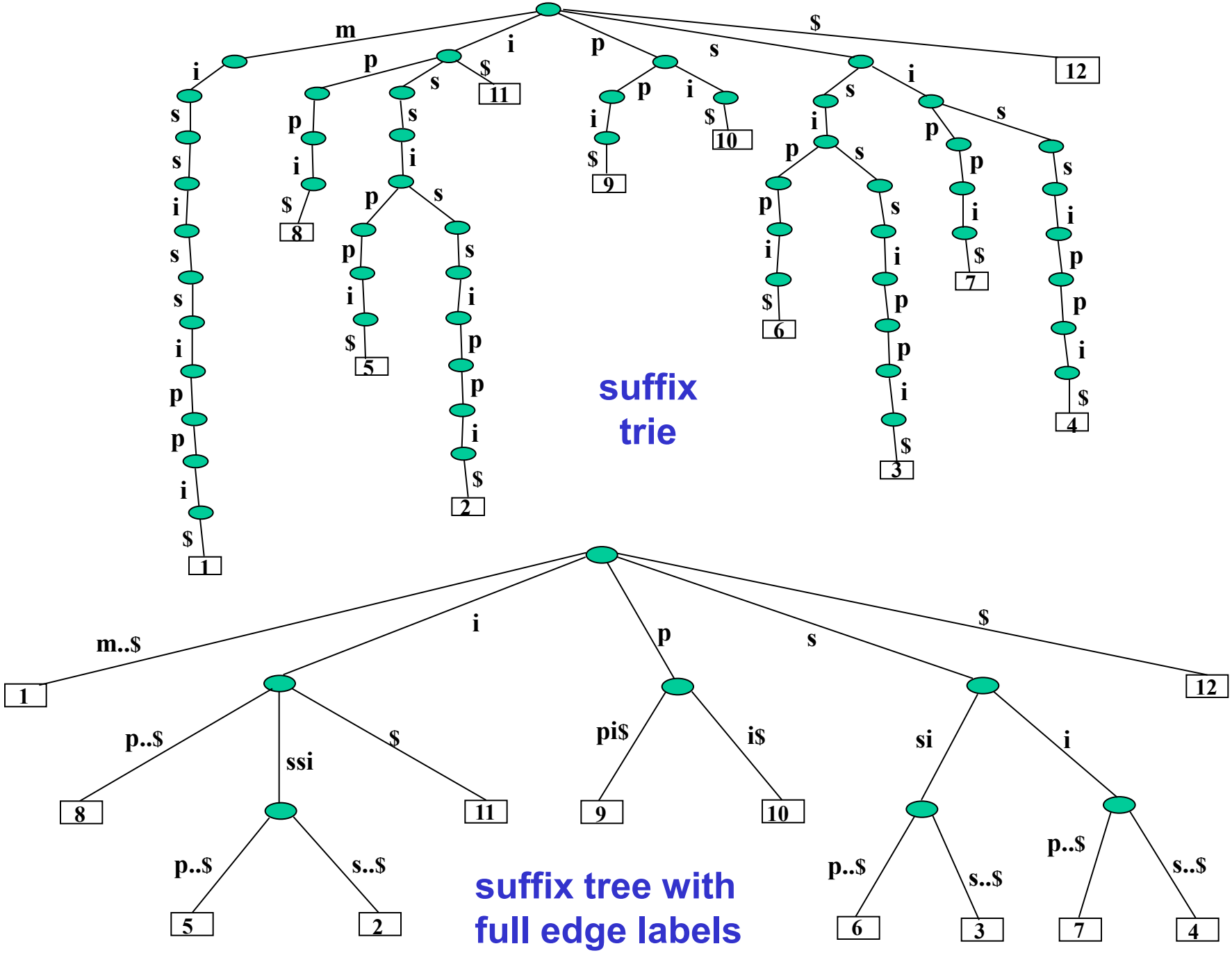


The suffix tree of a string S

First construct S* from S by appending a character ($) that does not appear in S.

The *suffix tree* of S* (loosely, of S) is a rooted tree in which

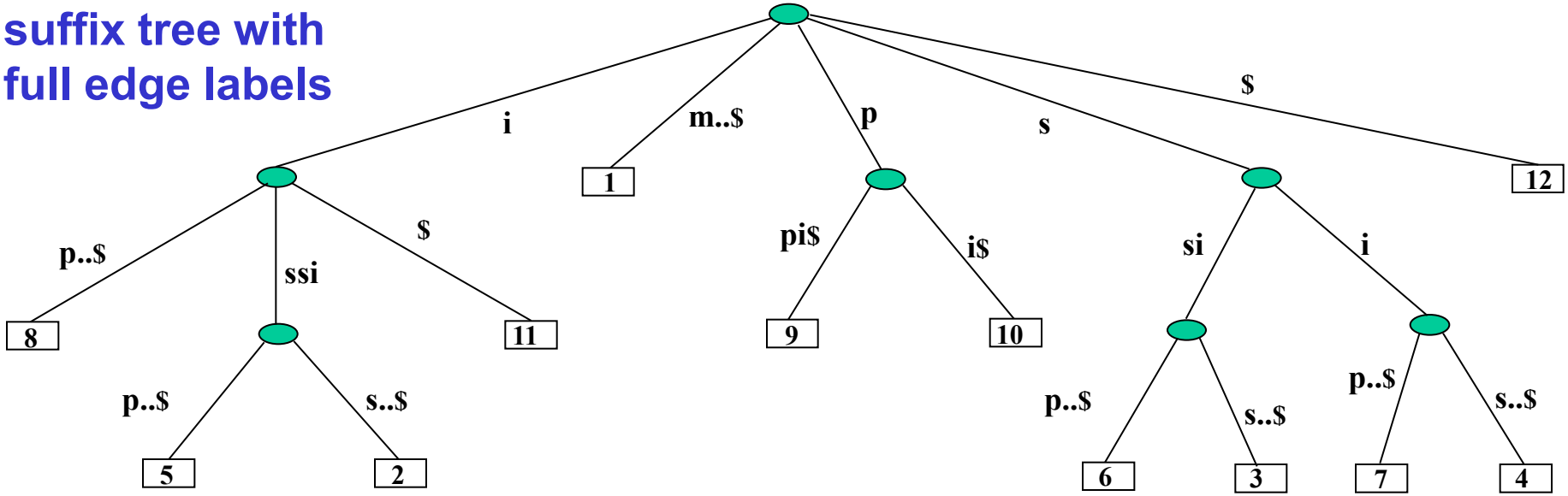
- each edge is labelled by a string (its *edge label*)
- all branch nodes have ≥ 2 children
- no two children of a node have edge labels beginning with the same character
- there is a one-to-one correspondence between suffixes of S* and leaf nodes v, in the sense that the path label of v is precisely that suffix
  - the *path label* of a node is the concatenation of edge labels on the path from the root to that node


suffix
trie
m..$
i
p
s
$
1
p..$
ssi
$
8
11
pi$
i$
9
10
si
i
p..$
s..$
5
2
p..$
s..$
6
3
p..$
s..$
7
4
12
suffix tree with
full edge labels

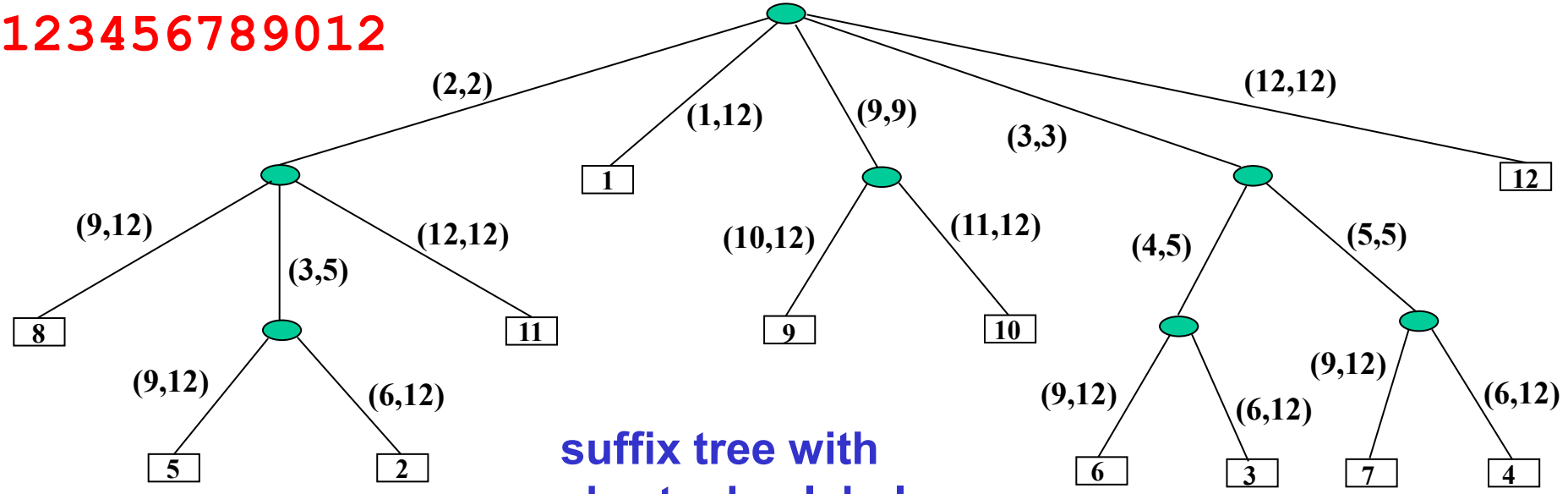
Properties of suffix trees

- The number of leaf nodes is n+1 (n = |S|)
- The number of branch nodes is ≤ n
  - proof is an exercise
- So the total number of nodes is O(n)
- The sum of the lengths of the edge labels is the same as in the suffix trie, so no better than $O(n^2)$ in the worst case
- To avoid this space overhead, use *short edge labels*
  - replace a string X by a pair of integers (i, j) such that X = S*(i . . j)
  - each short edge label has length O(1)
  - so the sum of the lengths of the edge labels is O(n)

suffix tree with full edge labels



mississippi$
123456789012



suffix tree with short edge labels

Suffix tree construction 1

Let S be a string of length n. Append a $ character to form S*

Naïve method for creating suffix tree T:

1. Create suffix trie for S*

```
SuffTrie suffTrie = new SuffTrie();
for (int j=1; j<=n+1; j++)
  insert(suffTrie, S*[j..n+1]);
```

2. Amalgamate edges of paths containing unary branch nodes into single edges

3. Create short edge labels

Worst-case time complexity:

1. $O(n^2)$ 2. $O(n^2)$ 3. $O(n^2)$

Overall: $O(n^2)$. Space complexity: $O(n^2)$

Suffix tree construction 2

Better method for creating suffix tree T:

Create suffix tree (with short labels) for S* directly

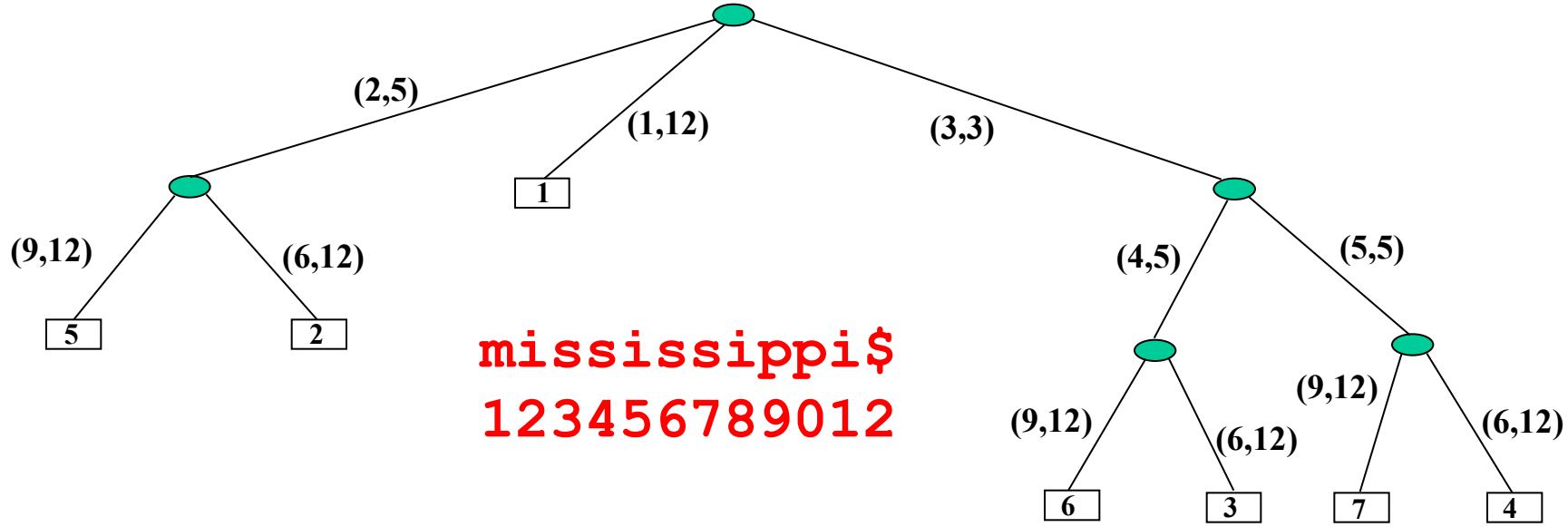
```
SuffTree suffTree = new SuffTree();
for (int j=1; j<=n+1; j++)
  insert(suffTree, S*[j..n+1]);
```

To insert a suffix, follow an existing path and split from a certain point

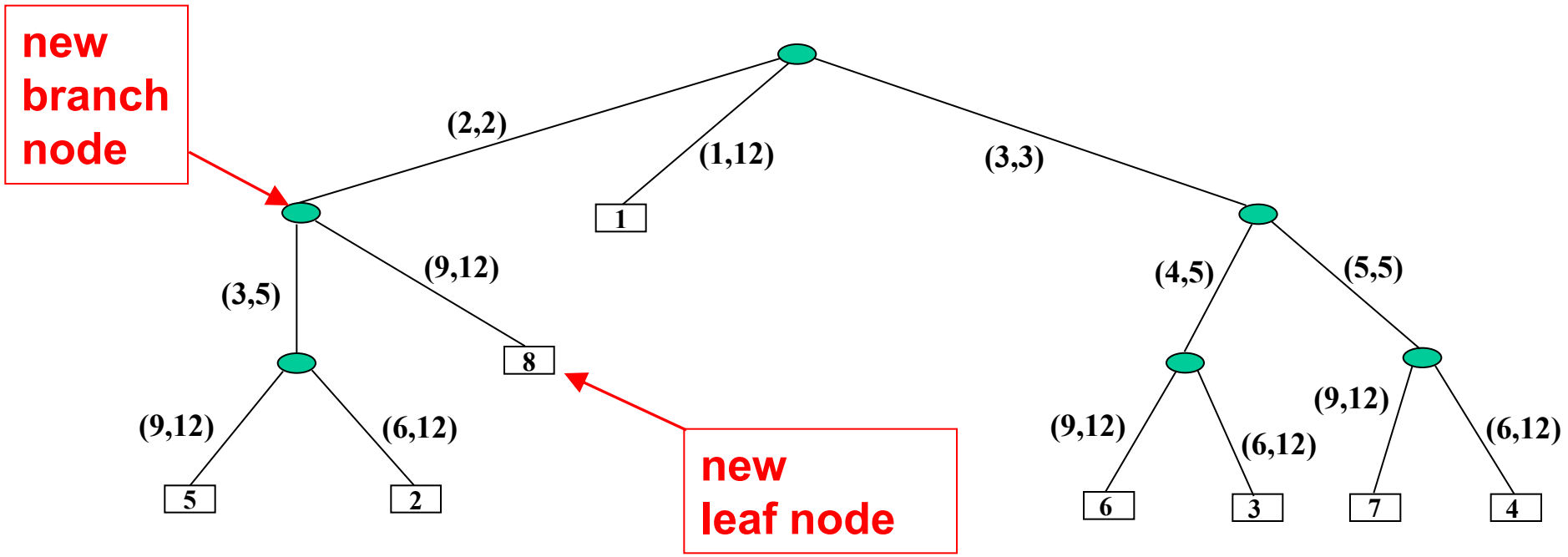
Worst-case time complexity still $O(n^2)$

But space complexity O(n)

And average time complexity O(n log n)



First 7 suffixes inserted; insert suffix 8



Suffix tree construction 3

Even better algorithms

For a string of length n over an alphabet Σ:

Weiner (1973), *McCreight* (1976), *Ukkonen* (1995)

All O(n log $|\Sigma|$) time and O(n) space for suffix tree construction

More recently:

Farach (1997)

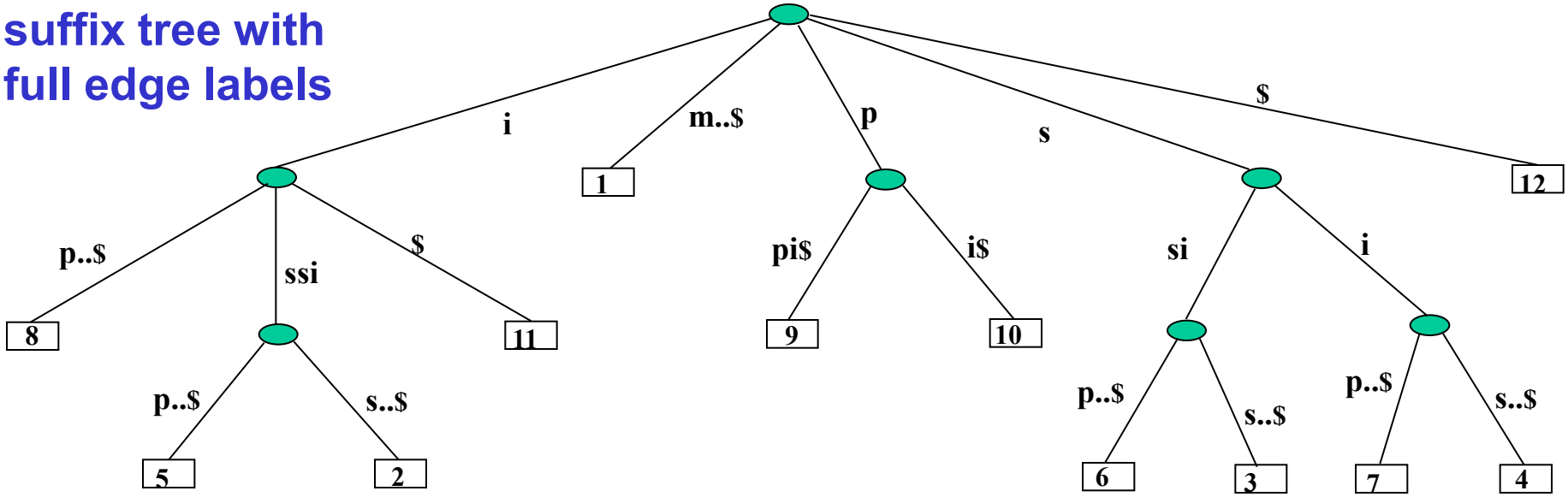
O(n) time and space for suffix tree construction

These algorithms are beyond the scope of the course

Problem solving with a suffix tree 1

- *Searching* a text T of length n, for a string S of length m
  - build the suffix tree for T
    - O(n) time
  - follow a path from the root matching characters
    - O(m) time
  - all occurrences are represented by the leaf node descendants from the point where the search ends
  - can search a text of length n for k short strings of total length r in O(n + r) time
  - e.g. find all occurrences of is in mississippi

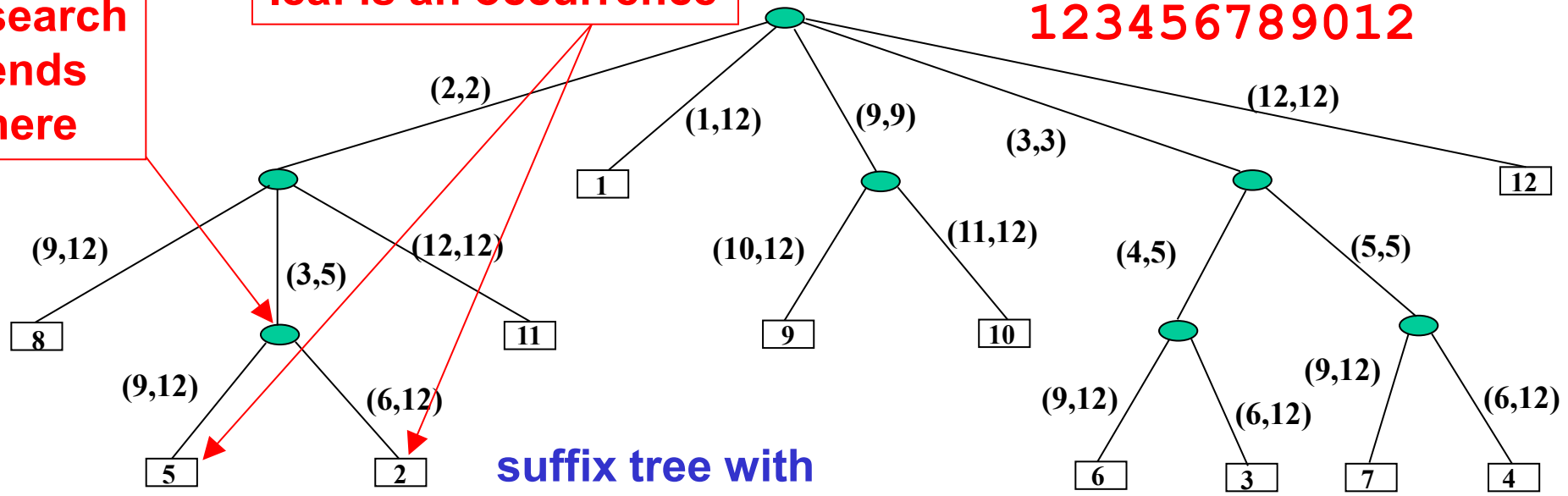
suffix tree with full edge labels



each descendant leaf is an occurrence

search ends here

mississippi$
123456789012

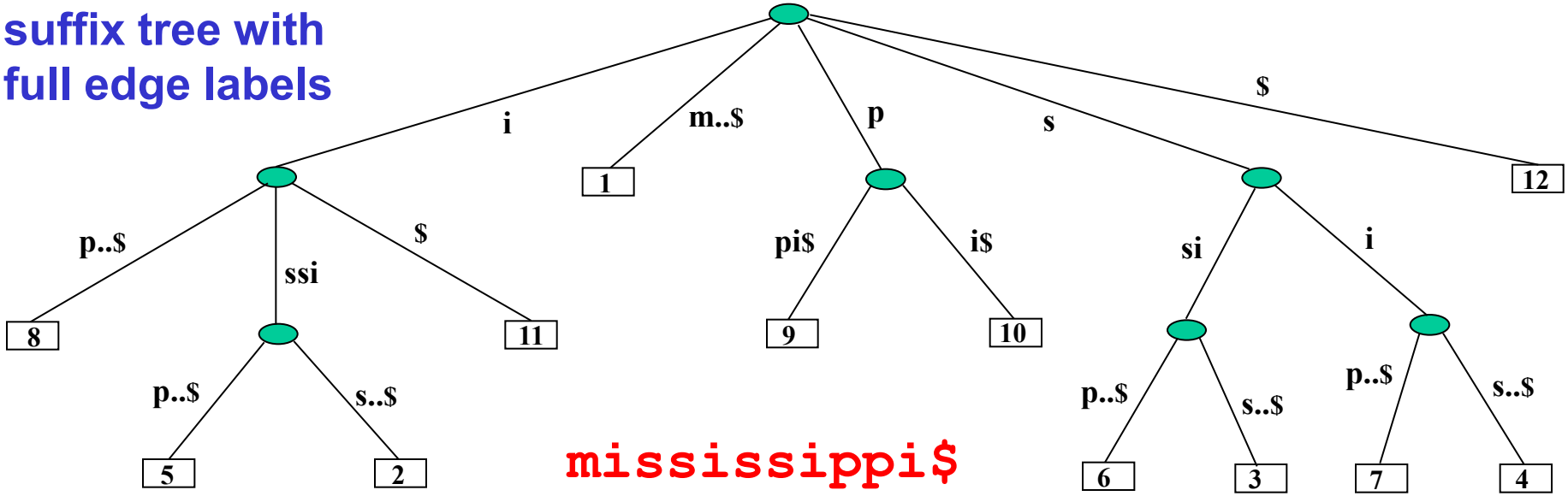


suffix tree with short edge labels

Problem solving with a suffix tree 2

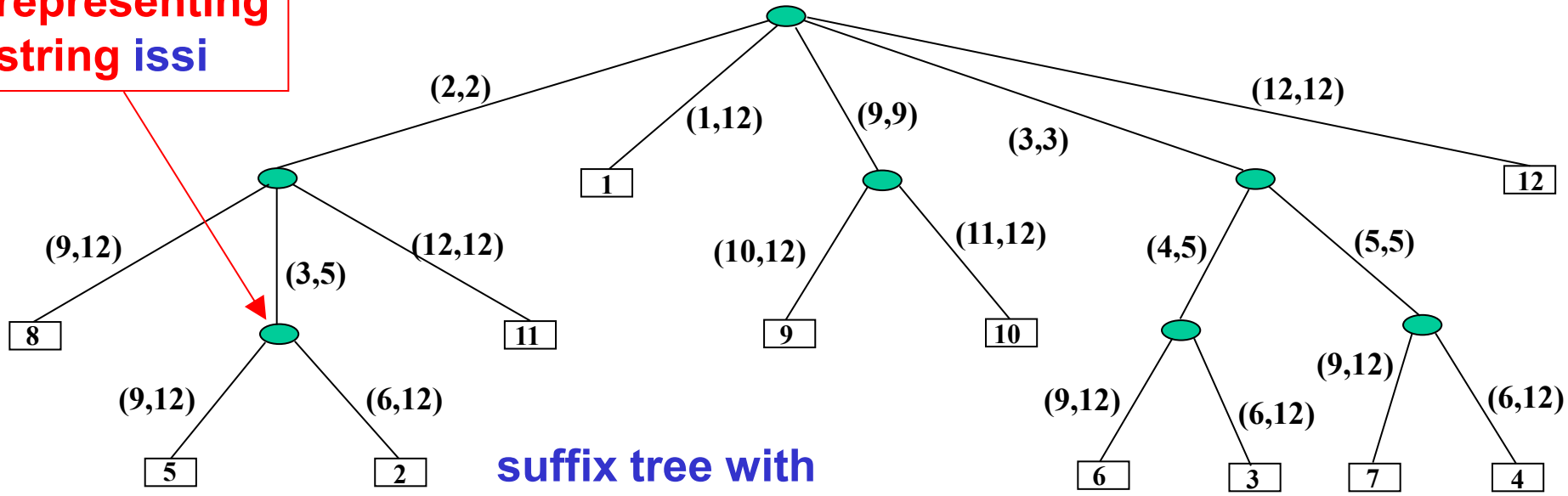
- Finding a *longest repeated substring* in a text T, length n
  - build the suffix tree for T
    - O(n) time
  - traverse the tree
  - a longest repeat is represented by a branch node having greatest string depth
    - the *string depth* of a node is the sum of the lengths of the edge labels on the path from the root to that node
    - i.e., it is the length of the path label of that node
  - traversal is also O(n), so the entire algorithm is O(n)

suffix tree with full edge labels



mississippi$
123456789012

Here it is – representing string issi



suffix tree with short edge labels

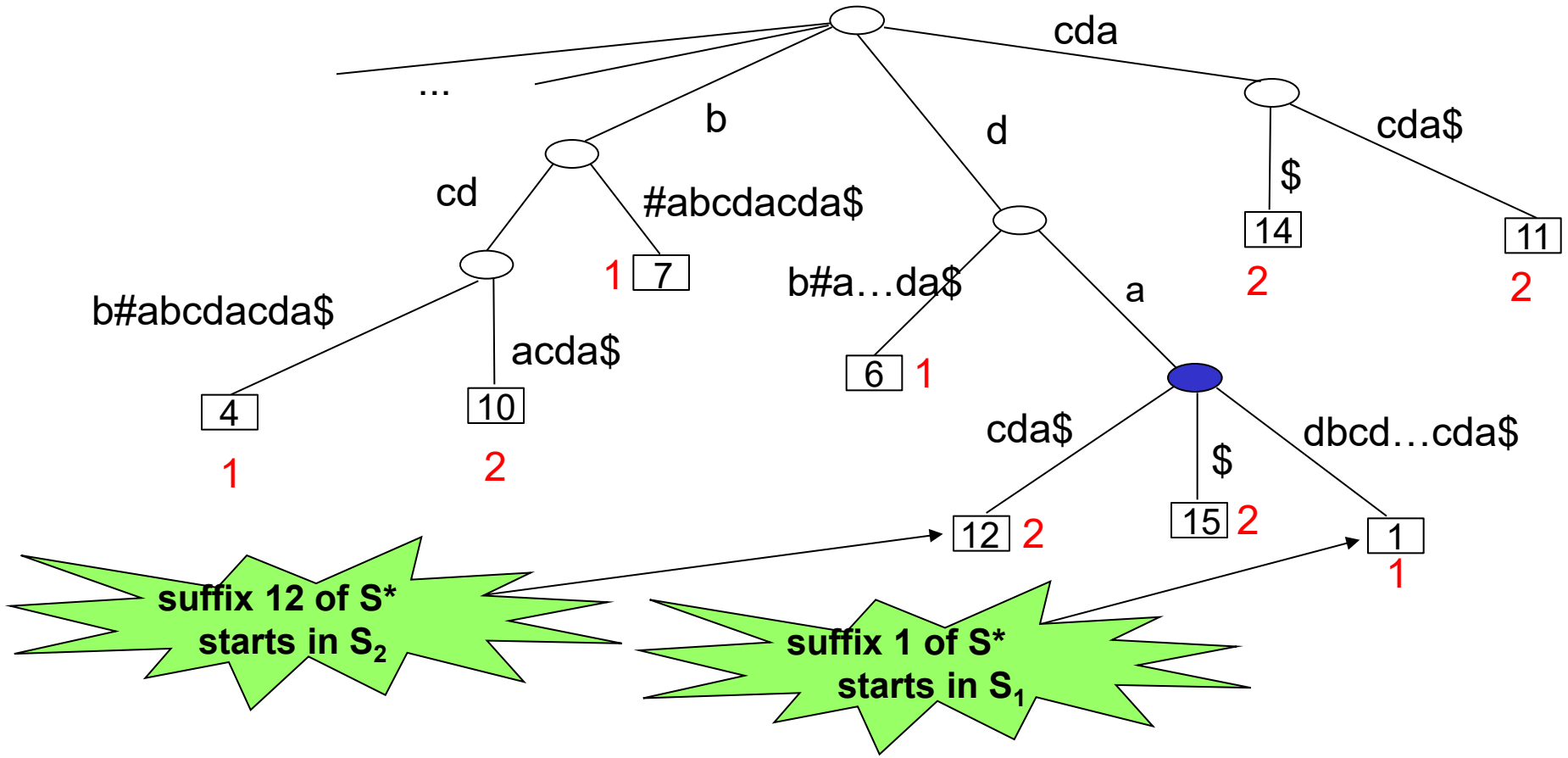
Problem solving with a suffix tree 3

- Finding a *longest common substring (LCSt)* of strings S_1, S_2 of lengths m, n
  - example - strings S_1 = dadbcdb and S_2 = abcdacda
  - build the *generalised suffix tree* T for S_1 and S_2
  - this is the suffix tree for $S = S_1 \# S_2$, i.e., T stores the suffixes of the concatenation of S_1, #, S_2, $ in that order (where neither # nor $ occurs in S_1 or S_2)
  - traverse the tree — a longest common substring is represented by a *common branch* node with greatest string depth
  - a branch node is *common* if it has two descendant leaf nodes representing positions starting in S_1 and S_2

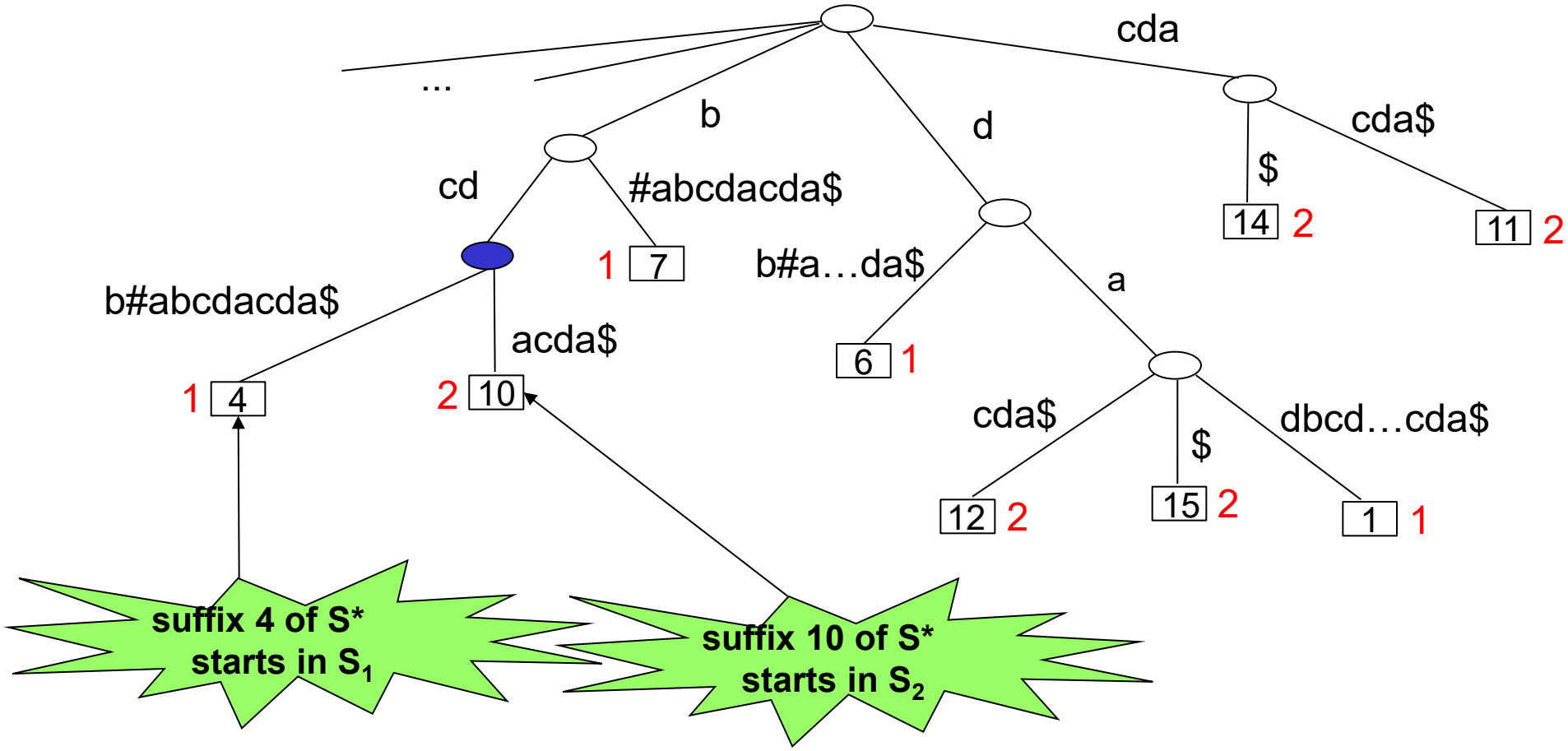
Example - strings S_1 = dadbcdb and S_2 = abcdacda.

1234567890123456 7

Form S*=dadbcdb#abcdacda$ and the generalised suffix tree T:



= common branch node, string depth 2



= common branch node, string depth 3

Longest common substring is `bcd`

Identifying common branch nodes

Maintain booleans at each node v of T as follows:

Define $b_i(v)$ to be true if and only if the path label of a descendant leaf node of v starts in string S_i $(i \in \{1, 2\})$

A branch node v of T is common if and only if $b_1(v) \wedge b_2(v)$

Computing the b_i values

Let v be a leaf node corresponding to suffix no. j of S^*, so path label of v is $S^*(j \,..\, m+n+2)$

$b_1(v)$ = true if and only if $1 \leq j \leq m$
$b_2(v)$ = true if and only if $m+2 \leq j \leq m+n+1$

For a branch node v, $b_i(v) = b_i(w_1) \vee b_i(w_2) \vee \ldots \vee b_i(w_r)$

where $w_1, \ldots, w_r$ are the children of v $(i \in \{1,2\})$

LCSt - putting it all together

Given strings S_1, S_2 of lengths m, n.

1. Build the generalised suffix tree T — O(m+n) time

2. Calculate the $b_i(v)$ values — O(m+n) time using a traversal of T

3. Output the path label of a common branch node w of T with maximum string depth — O(m+n) time

(In practice, w may be computed during the traversal in 2)

Overall O(m+n) time and space to find LCSt of S_1, S_2

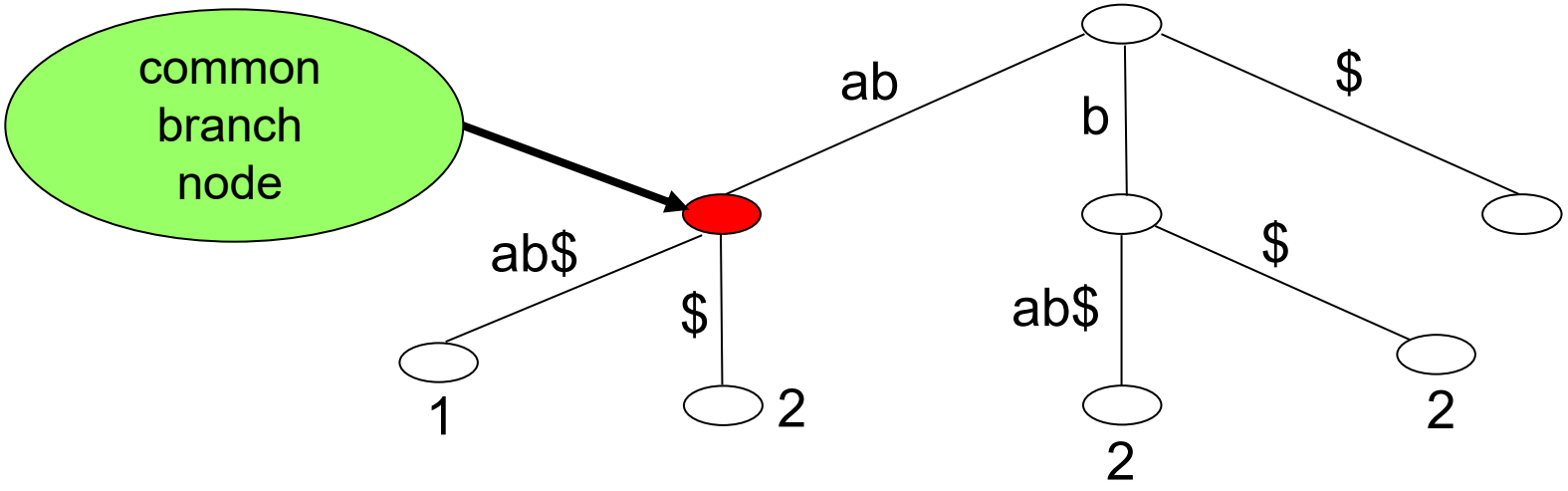
Finding an LCSt of S_1 and S_2:

Why did we need to append # to S_1?
Why not just consider $S_1S_2$$?

Example - suppose S_1=a and S_2=bab

Clearly the unique LCSt of S_1 and S_2 is a

Consider the suffix tree for $S_1S_2$$ = abab$ rather than S_1# $S_2$$ as before



Algorithm incorrectly reports that ab is an LCSt of S_1 and S_2