Part 2
String and Text Algorithms

- Suffix tries and suffix trees
- Applications of suffix trees
  - variants of string searching
  - longest common substrings
- Matching regular expressions
- Global similarities in strings
  - longest common subsequence
  - the technique of memoisation

String / Text Algorithms

Some notation and terminology

- an *alphabet* Σ=`{A,C,G,T} (DNA)`
  Σ= `ASCII, Unicode, etc.`
- a *string*: `ACAGTCCGGGACTGACG`

Throughout this section, all strings are indexed from position 1 (for consistency with the literature)

- Let S,T be two strings
  - |S| is the *length* of S
  - S(i) denotes the ith symbol of S
  - S(i . . j) denotes S(i)S(i+1) . . S(j) (strings indexed from 1)
  - ST is the *concatenation* of S and T
- ε – the *empty string*, length 0

Further notation and terminology

- a ***substring***: ACAGTCCGG*GACTG*ACG
  - a substring of S is ε or S(i . . j) (i ≤ j)
  - U is a substring of S if and only if S = TUV for some strings T, V
- a ***common substring***: TCCACT*GACTG*CTGC

  ACAGTCCGG*GACTG*ACG
  - U is a common substring of S and T if U is a substring of both S and T

- *a* ***subsequence***: AC*A*G*T*CC*G*GGA*C*T*GA*CG
  - obtained by deleting zero or more characters from the string
- a ***common subsequence***:

  A*CA*G*T*CC*G*GG*ACTG*A*CG*

  T*C*C*A*C*TGACTGC*T*G*C
  - U is a common subsequence of S and T if U is a subsequence of both S and T

Notation and terminology (continued)

- a *prefix*: ***ACAGT***CCGGGACTGACG
  - S(1 . . k) for some k (1 ≤ k ≤ n), where n = |S|

- a *suffix*: ACAGTCCGGG***ACTGACG***
  - S(k . . n) for some k (1 ≤ k ≤ n), where n = |S|

- Σ^* = {ε,A,C,G,T,AA,AC,AG,AT,CA,...}
  - set of all strings composed of symbols from the alphabet Σ

- some tree terminology:
  - a *leaf* node: no children
  - a *branch* node: one or more children (includes root)
  - a *unary* node: exactly one child
  - a *binary* node: exactly two children

Suffix trees and applications

Some sample motivating problems

Multiple searches

- **How would you search one long text for occurrences of 1000 short strings?**
  - **Use KMP or BM algorithm**
  - **but each search involves scanning the long text**
  - **what if we *preprocess* the text to build an exploitable data structure – effectively an *index* for the text?**

Repeated substrings

- **How would you find the longest repeated substring in a gene (DNA string)?**
- **or the longest piece of text common to two written works?**

Problem: finding a longest repeated substring

Example: S=cabdababdc   **Longest repeat is:** abd

Finding a longest repeat: naïve solution - $O(n^3)$

Faster solution:

- for a string **S** of length **n**, build an $n \times n$ array **A** with $A(i, j) = 1$ if $S(i) = S(j)$, and $A(i, j) = 0$ otherwise

E.g. find a longest repeat in S=cabdababdc

		1	2	3	4	5	6	7	8	9	0
		c	a	b	d	a	b	a	b	d	c
1	c	–	0	0	0	0	0	0	0	0	1
2	a	–	–	0	0	1	0	1	0	0	0
3	b	–	–	–	0	0	1	0	1	0	0
4	d	–	–	–	–	0	0	0	0	1	0
5	a	–	–	–	–	–	0	1	0	0	0
6	b	–	–	–	–	–	–	0	1	0	0
7	a	–	–	–	–	–	–	–	0	0	0
8	b	–	–	–	–	–	–	–	–	0	0
9	d	–	–	–	–	–	–	–	–	–	0
0	c	–	–	–	–	–	–	–	–	–	–

Finding a longest repeat: naïve solution (cont.)

- repeated substrings are represented by sequences of 1's on a diagonal
- scan all diagonals to find longest repeat
- requires $O(n^2)$ time and space
- NB. repeated substrings may overlap -

  e.g. `S = c a b a b a d`

Common substrings

Find longest common substring of `S=ababdc`, and `T=ccbab`

- longest common substring is `bab`
- for two strings `S`, `T` of lengths m and n, build a similar $m \times n$ array of 0's and 1's - gives a longest common substring of `S` and `T` in $O(mn)$ time and space

Suffixes

Let S be a string of length n

- the k^{th} suffix of S is the suffix $S(k..n)$
- S has n suffixes, including S itself

Aim: define data structures to store the n suffixes of S

- Suffix trie - $O(n^2)$ space
- Suffix tree - $O(n)$ space

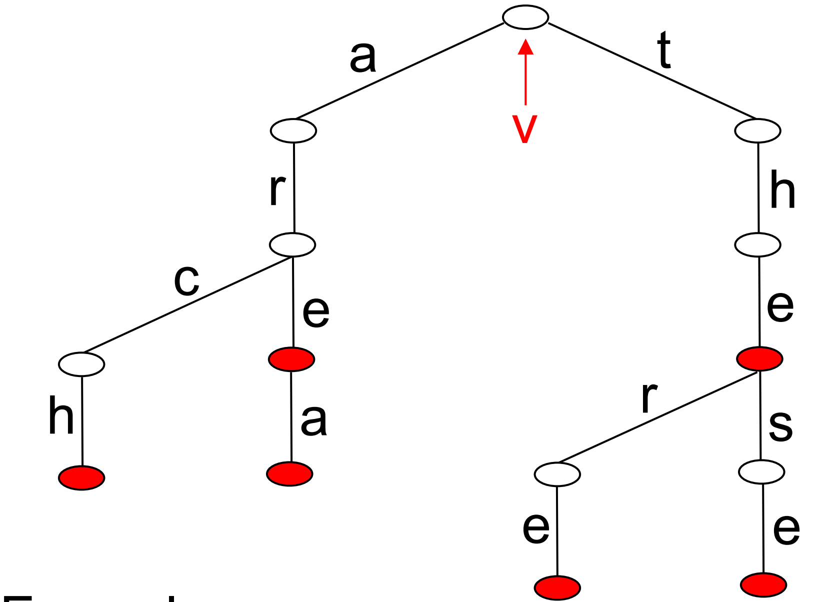
Use the suffix tree to solve the following problems:

- Longest repeat of S in $O(n)$ time and space
- Longest common substring in $O(m+n)$ time and space for two strings S,T, where $m=|S|$ and $n=|T|$
- Multiple string searching in $O(n+r)$ time and $O(n)$ space, for long piece of text T ($n=|T|$) and strings of total length r

Suffix tries

Tries (as in Algorithmics I)

A *trie* is a multiway branching tree T which may be used to store a set of strings C over an alphabet Σ. It has the following properties:



Example:
trie for C={are,arch,area,
the,there,these}

implies path label $\in$ C

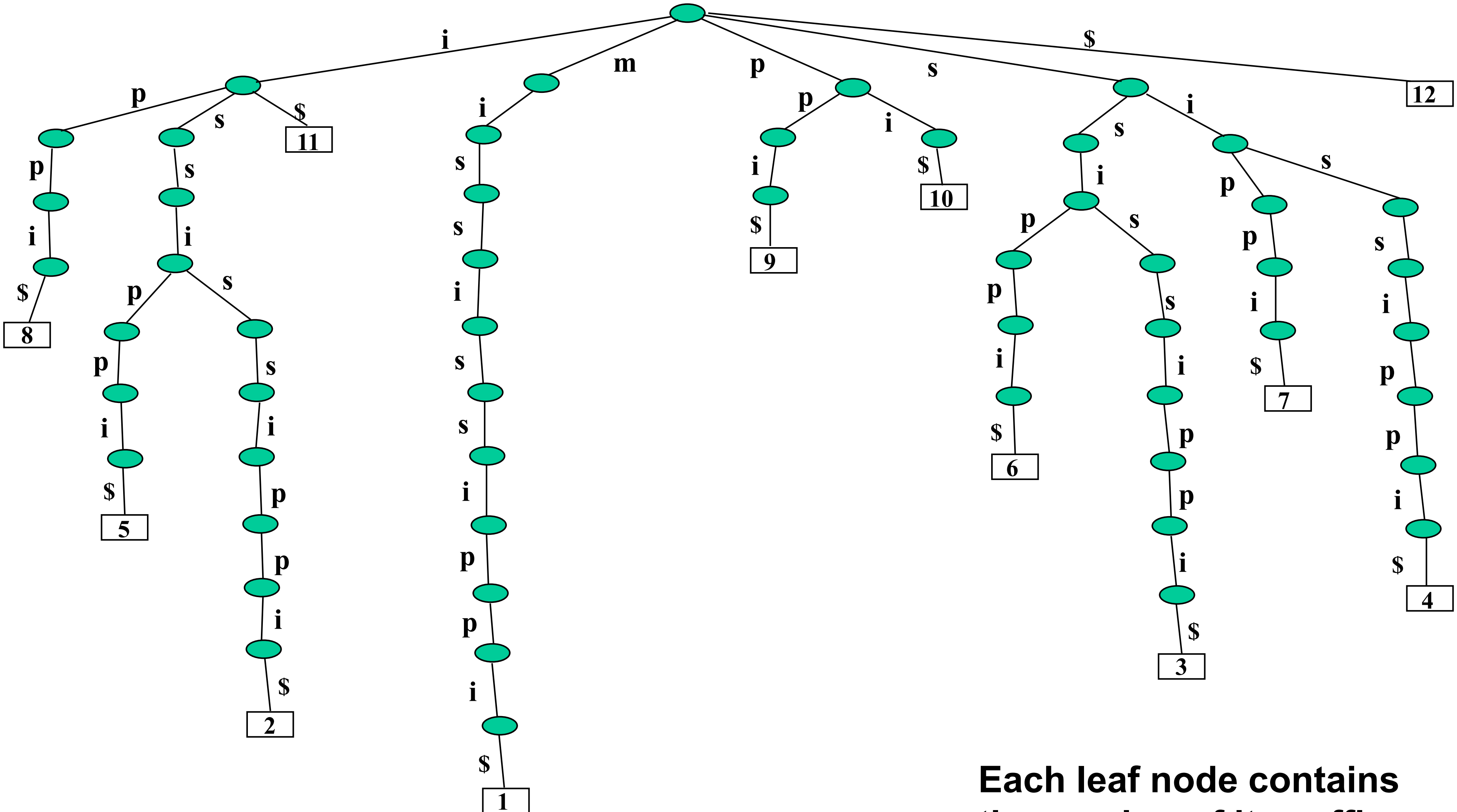
- **T is rooted at some vertex v**
- **Each edge of T is labelled with some $\sigma \in \Sigma$**
- **No two children of a node of T have the same edge label**
- **Each node w corresponds to a string $S \in \Sigma^*$ (concatenation of edge labels on the path from v to w) - S is the *path label* of w**
- **Each node is marked according to whether it corresponds to a string $S \in C$**

- The *suffix trie* T for string S is a trie that is used to store all suffixes of S
- Each suffix of S must be represented by a unique leaf node of T
- If some suffix of S is a prefix of another suffix, then a suffix trie for S may not exist, e.g. consider S = queue
- We can ensure that the suffix trie exists by appending to S a character $ not appearing in S, before constructing T

Using a suffix trie

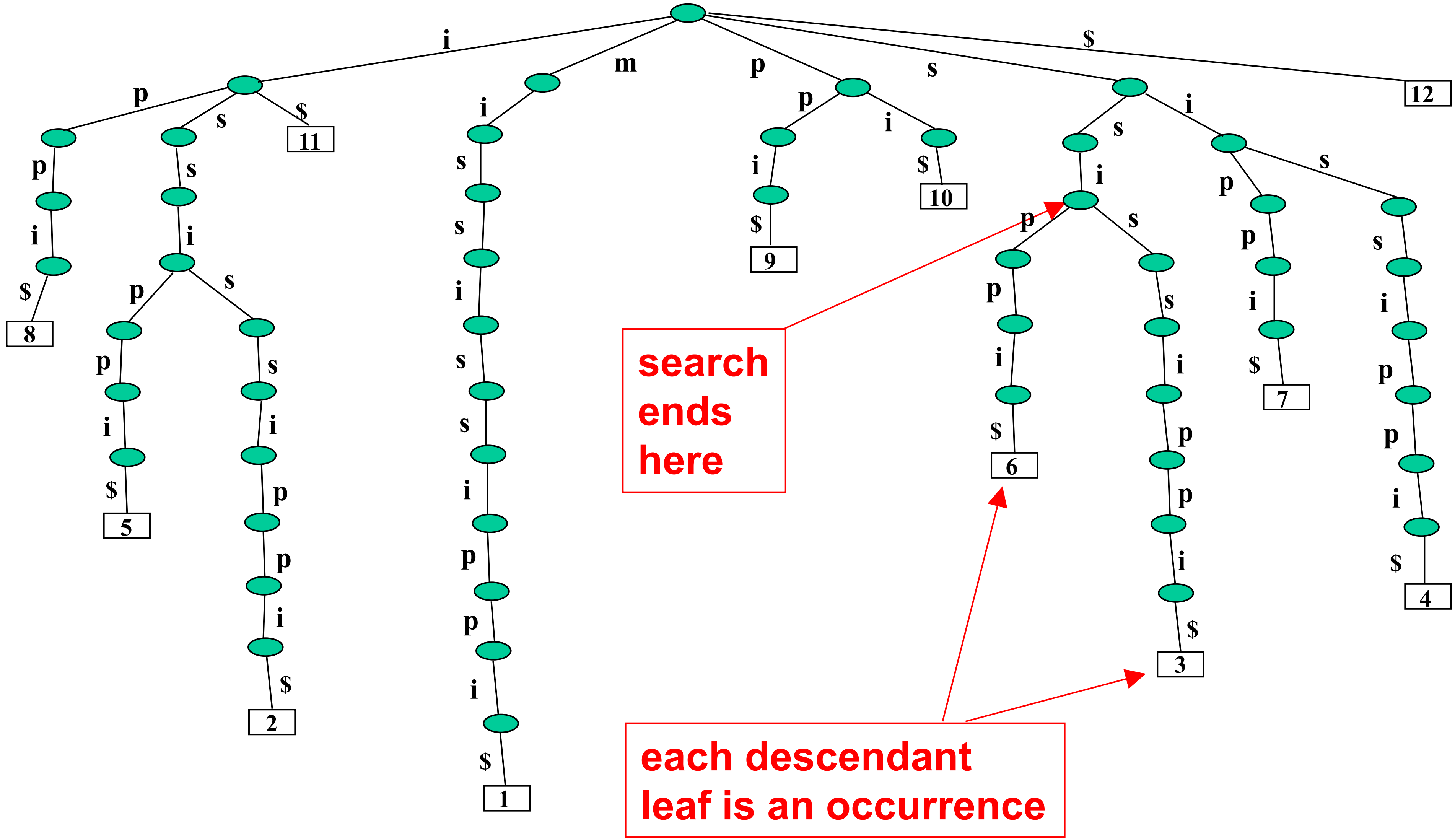
Once a suffix trie has been built for a text T of length n

- we can search for a string S of length m in (essentially) O(m) time
- we can find a longest repeat in T by traversing the trie
  - find a node with ≥2 children that is furthest from the root
  - distance from root = length of repeat

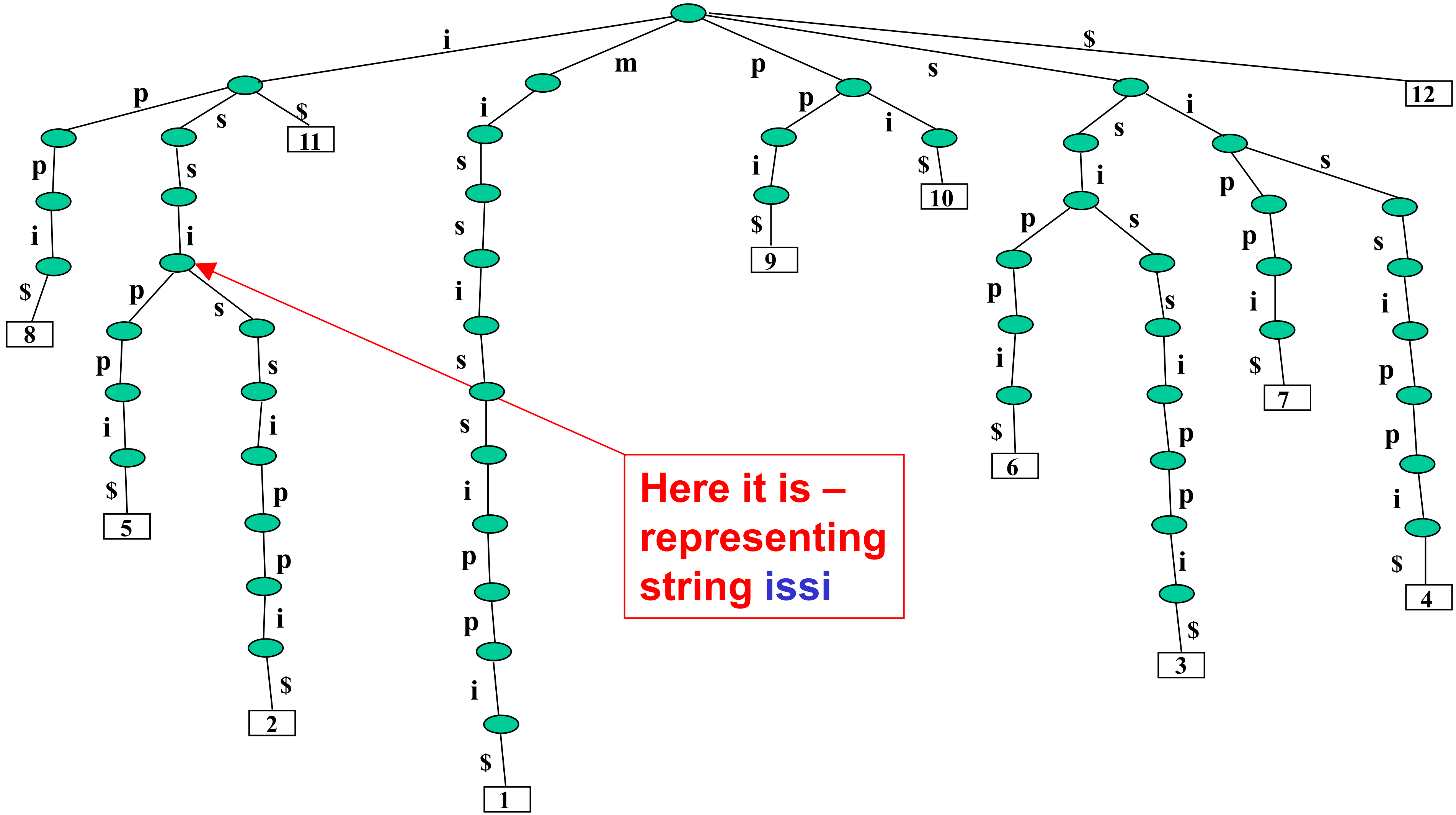


Each leaf node contains the number of its suffix

Suffix trie for `S=mississippi`
(append `$` - `mississippi$`)
`123456789012`



Search for occurrences of `ssi`



Search for longest repeat - traverse the tree to find a node with at least two children and maximum distance from root

Building a suffix trie

- Inserting the i^{th} suffix takes $O(n-i)$ steps
  - so requires $O(n^2)$ time overall

Is a suffix trie useful in practice?

- How many nodes?
  - for a text of length n, this is typically proportional to n^2
  - so the space needed is quadratic in n
- Searching for a string of length m is $O(m)$ after $O(n^2)$ preprocessing time
- Finding longest repeat is $O(n^2)$
- In both cases $O(n^2)$ space is needed
- Can we improve substantially on this?
  - Yes: with a *suffix tree*