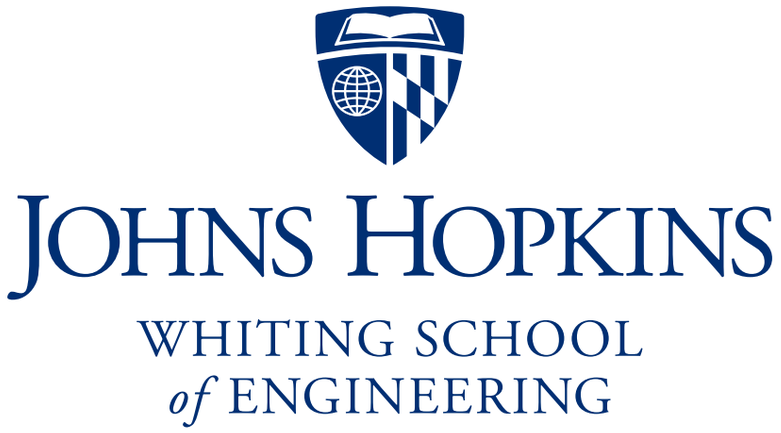
# Tries and suffix tries

Ben Langmead



# Tries

A trie (pronounced “try”) is a tree representing a collection of strings with one node per common prefix

Smallest tree such that:

Each edge is labeled with a character $c \in \Sigma$

A node has at most one outgoing edge labeled $c$, for $c \in \Sigma$

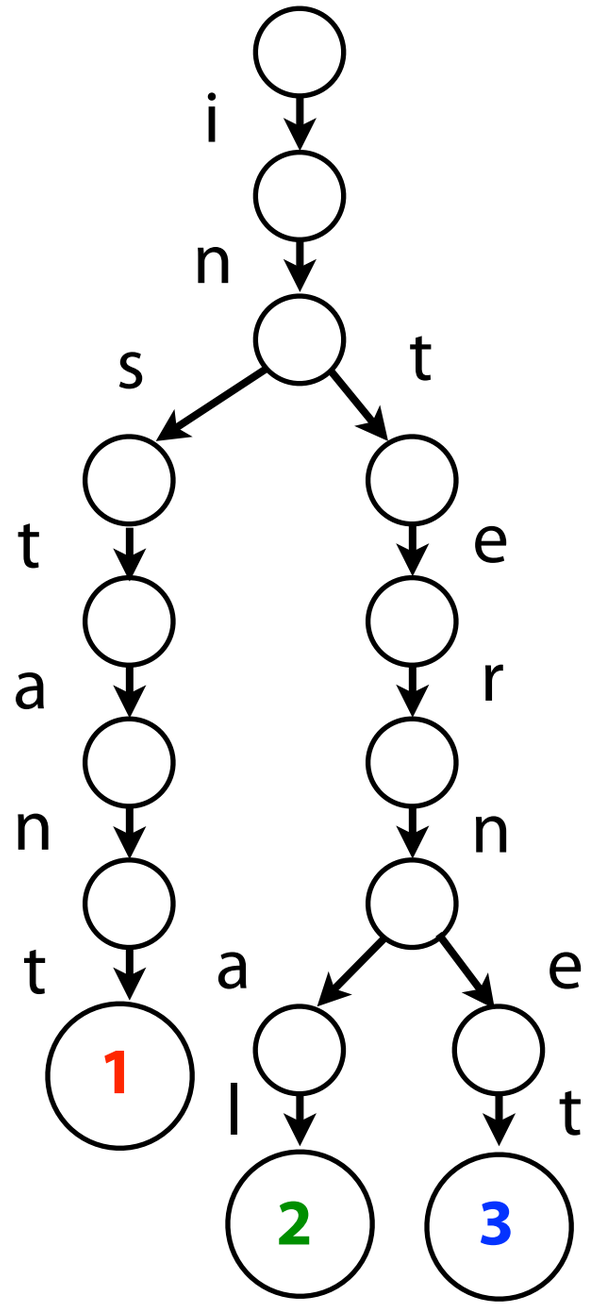
Each key is “spelled out” along some path starting at the root

Natural way to represent either a *set* or a *map* where keys are strings

# Tries: example

Represent this map with a trie:

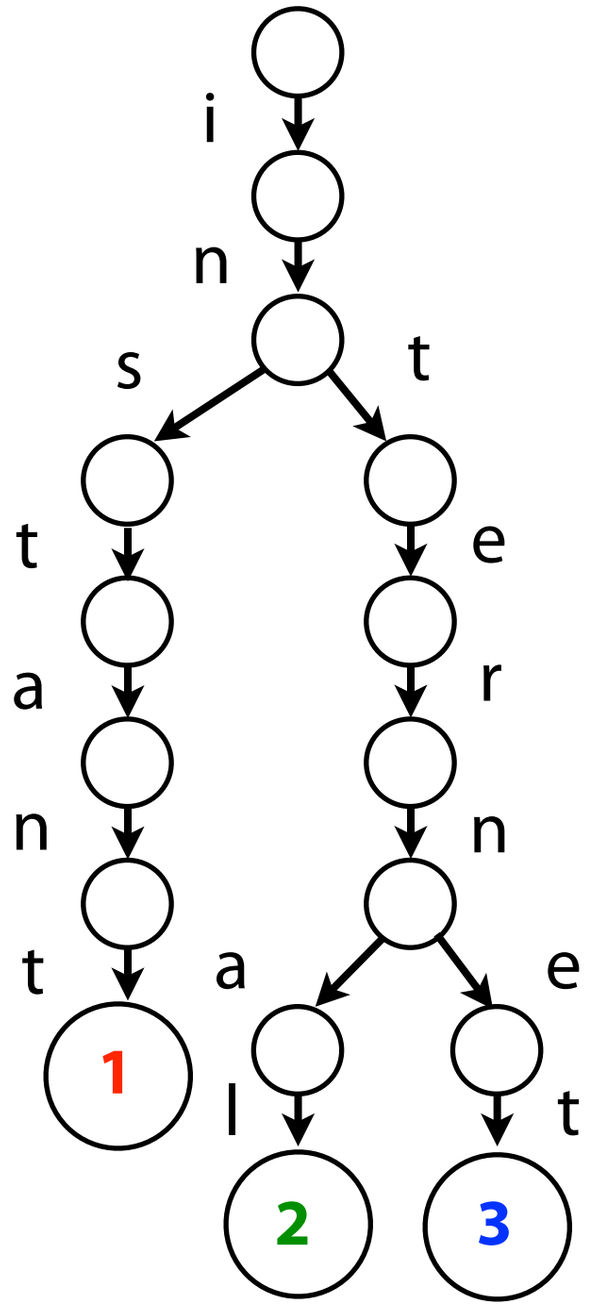
| Key | Value |
|---|---|
| instant | 1 |
| internal | 2 |
| internet | 3 |




The smallest tree such that:

- Each edge is labeled with a character $c \in \Sigma$
- A node has at most one outgoing edge labeled $c$, for $c \in \Sigma$
- Each key is "spelled out" along some path starting at the root

# Tries: example




Checking for presence of a key $P$, where $n = |P|$, is $\boldsymbol{O(n)}$ time

If total length of all keys is $N$, trie has $\boldsymbol{O(N)}$ nodes

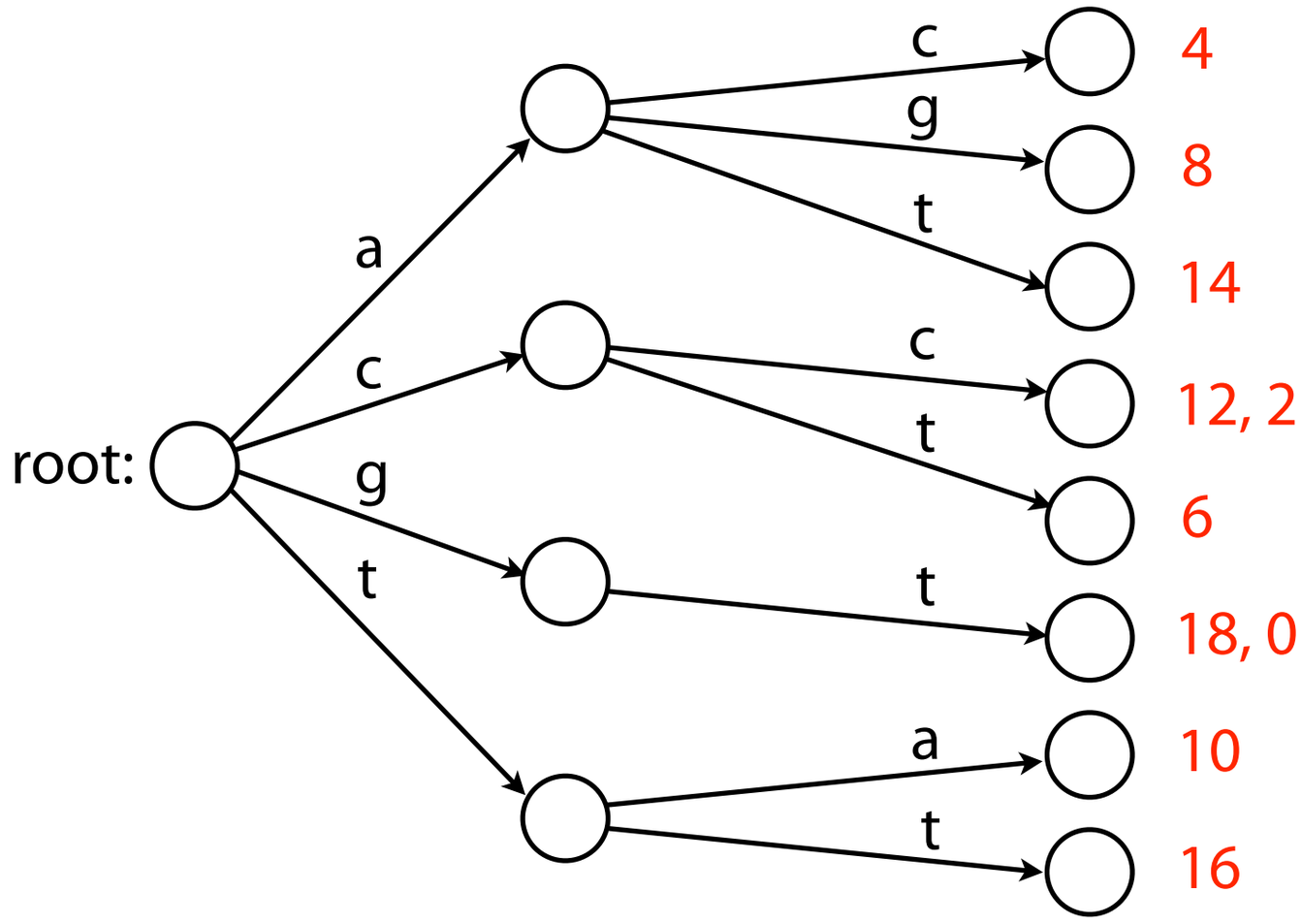
What about $|\Sigma|$?

Depends how we represent outgoing edges. If we don't assume $|\Sigma|$ is a small constant, it shows up in one or both bounds.

# Tries: another example

We can index *T* with a trie. The trie maps substrings to offsets where they occur

| | |
|---|---|
| ac | 4 |
| ag | 8 |
| at | 14 |
| cc | 12 |
| cc | 2 |
| ct | 6 |
| gt | 18 |
| gt | 0 |
| ta | 10 |
| tt | 16 |



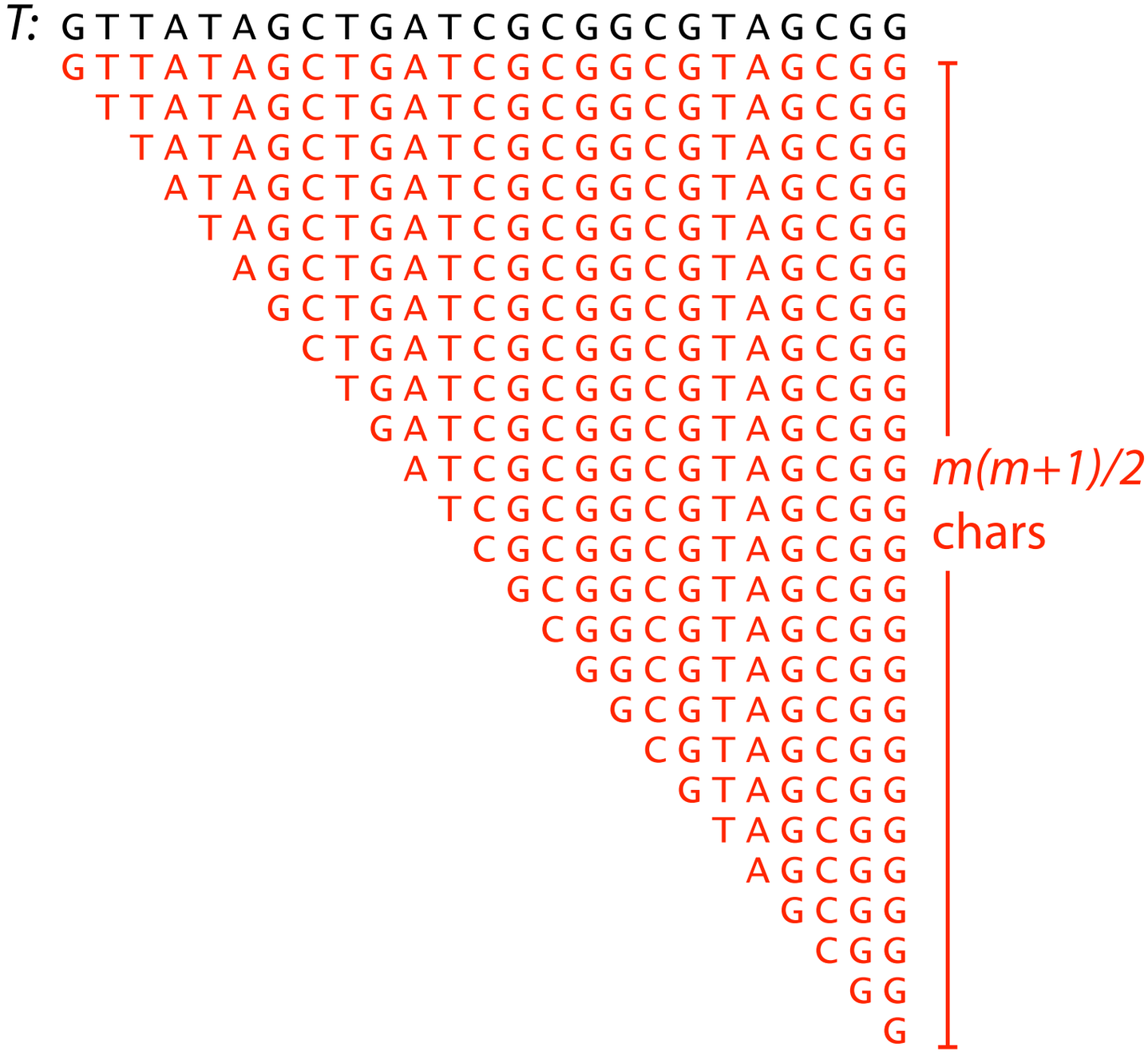
# Tries: implementation

```
class TrieMap(object):
    """ Trie implementation of a map.  Associating keys (strings or other
        sequence type) with values.  Values can be any type. """

    def __init__(self, kvs):
        self.root = {}
        # For each key (string)/value pair
        for (k, v) in kvs: self.add(k, v)

    def add(self, k, v):
        """ Add a key-value pair """
        cur = self.root
        for c in k: # for each character in the string
            if c not in cur:
                cur[c] = {} # if not there, make new edge on character c
            cur = cur[c]
        cur['value'] = v # at the end of the path, add the value

    def query(self, k):
        """ Given key, return associated value or None """
        cur = self.root
        for c in k:
            if c not in cur:
                return None # key wasn't in the trie
            cur = cur[c]
        # get value, or None if there's no value associated with this node
        return cur.get('value')
```

Python example:
http://nbviewer.ipython.org/6603619

# Suffix trie

Build a **trie** containing all **suffixes** of a text *T*



# Suffix trie

First add special *terminal character* **$** to the end of *T*

**$** is a character that does not appear elsewhere in *T*, and we define it to be less than other characters (for DNA: **$** < **A** < **C** < **G** < **T**)

**$** enforces a rule we're all used to using: e.g. "as" comes before "ash" in the dictionary. **$** also guarantees no suffix is a prefix of any other suffix.

*T:* G T T A T A G C T G A T C G C G G C G T A G C G G $

G T T A T A G C T G A T C G C G G C G T A G C G G $
T T A T A G C T G A T C G C G G C G T A G C G G $
T A T A G C T G A T C G C G G C G T A G C G G $
A T A G C T G A T C G C G G C G T A G C G G $
T A G C T G A T C G C G G C G T A G C G G $
A G C T G A T C G C G G C G T A G C G G $
G C T G A T C G C G G C G T A G C G G $
C T G A T C G C G G C G T A G C G G $
T G A T C G C G G C G T A G C G G $
G A T C G C G G C G T A G C G G $
A T C G C G G C G T A G C G G $
T C G C G G C G T A G C G G $
C G C G G C G T A G C G G $
G C G G C G T A G C G G $
C G G C G T A G C G G $
G G C G T A G C G G $
G C G T A G C G G $

# Tries

Smallest tree such that:

Each edge is labeled with a character from $\Sigma$

A node has at most one outgoing edge labeled with $c$, for any $c \in \Sigma$

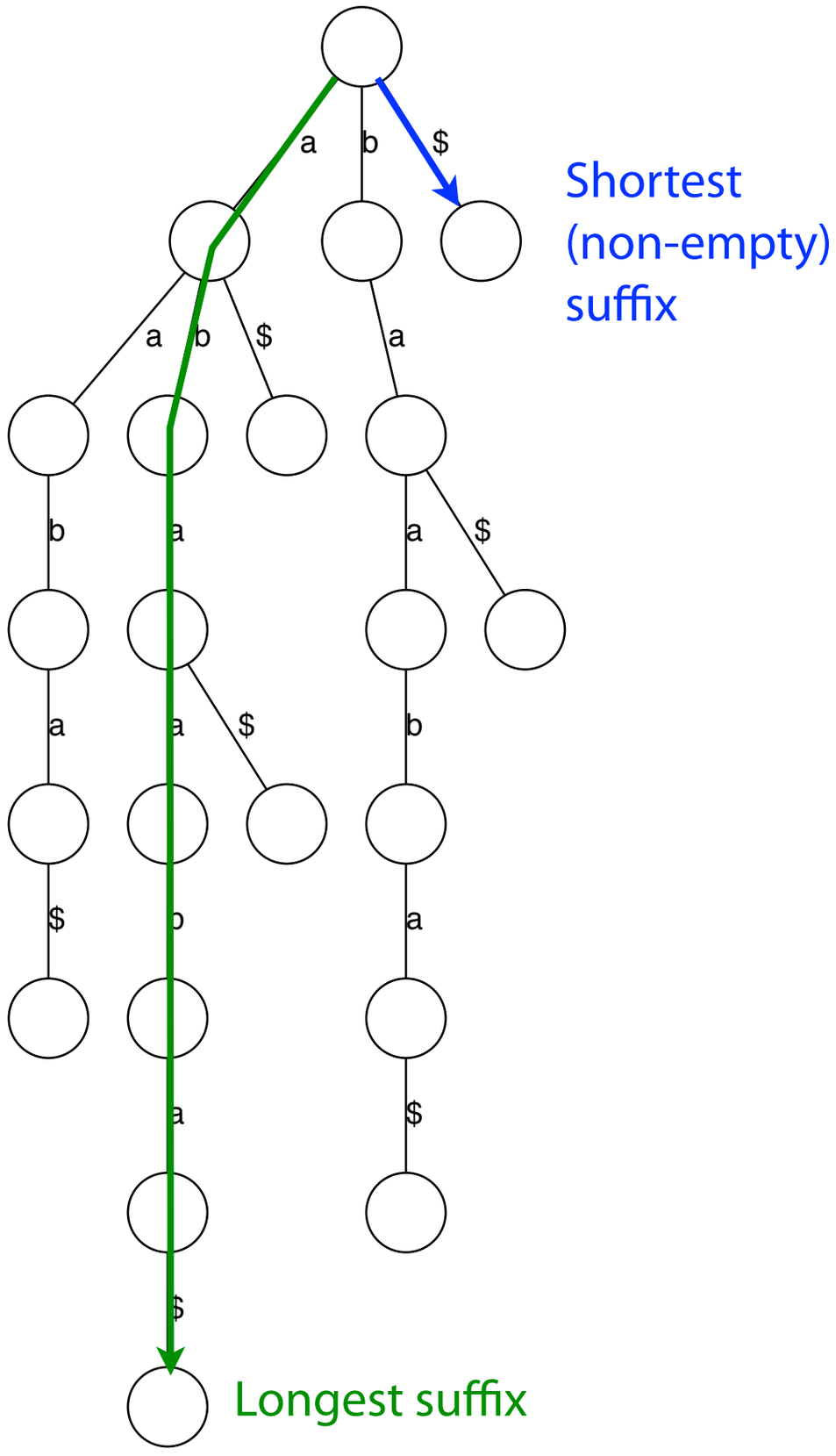
Each key is "spelled out" along some path starting at the root

# Suffix trie

*T:* abaaba *T***$***:* abaaba**$**

Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**?

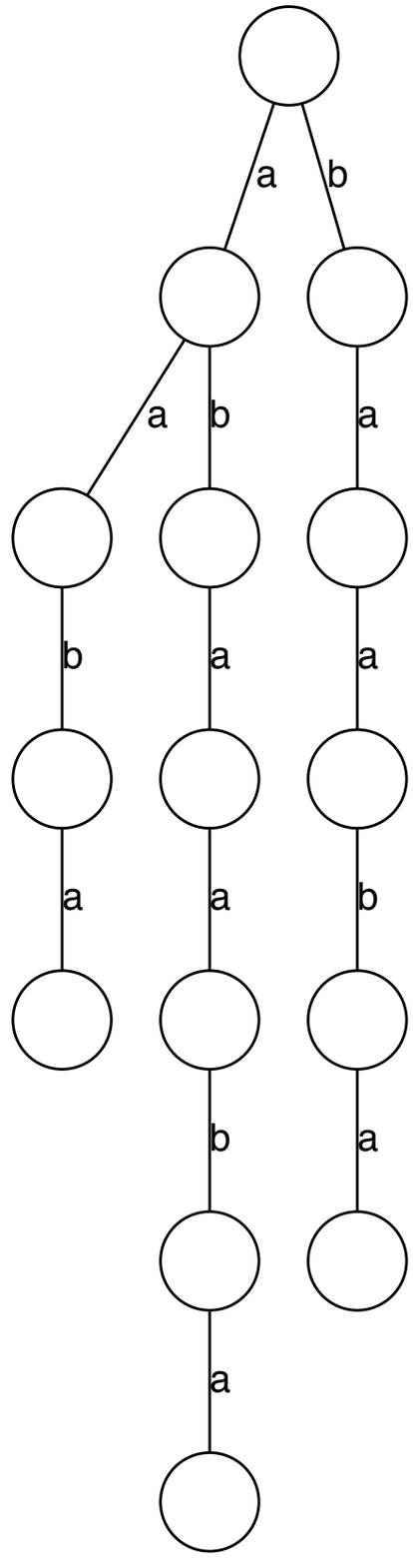


# Suffix trie

*T:* abaaba

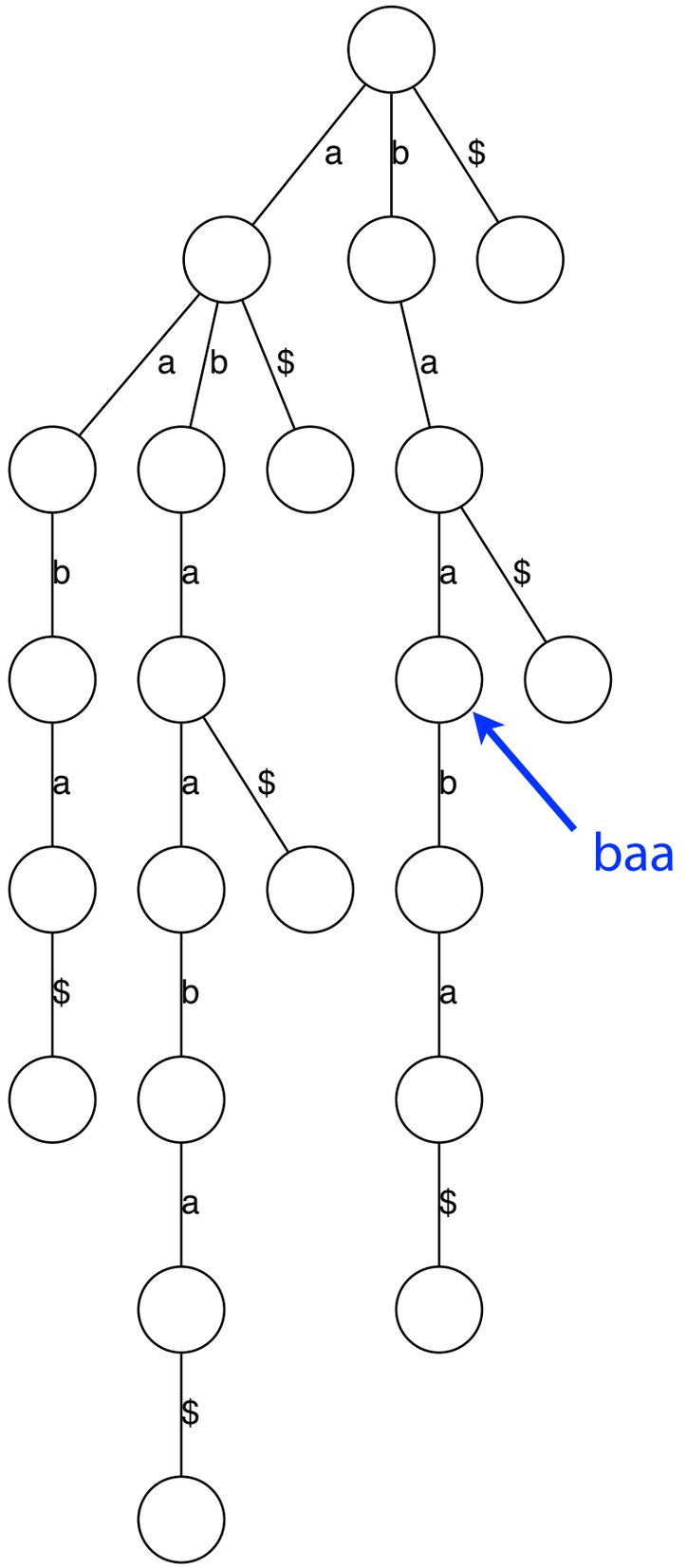
Each path from root to leaf represents a suffix; each suffix is represented by some path from root to leaf

Would this still be the case if we hadn't added **$**? **No**



# Suffix trie

We can think of nodes as having **labels**, where the label spells out characters on the path from the root to the node



# Suffix trie

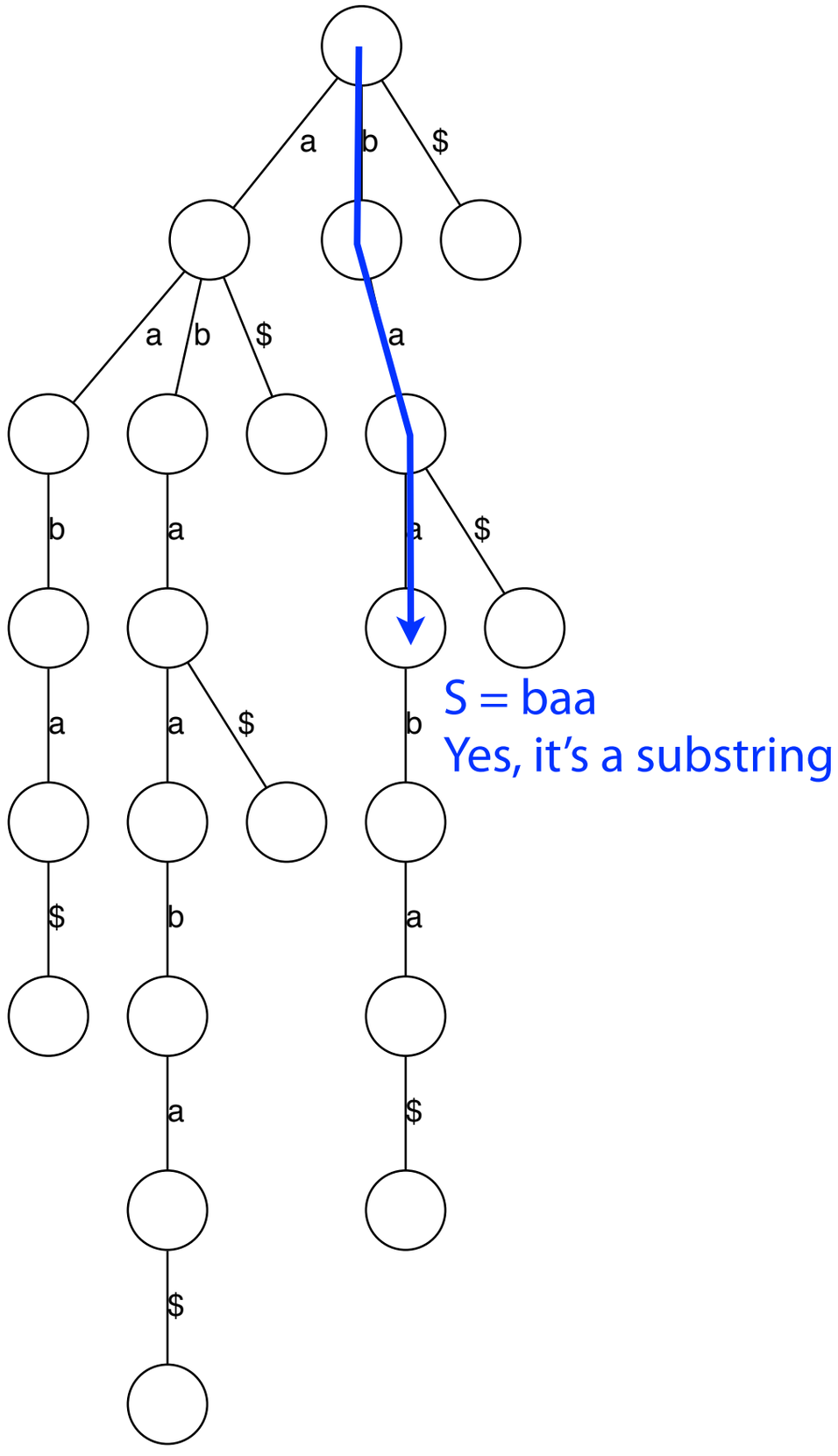
How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

If we exhaust *S* without falling off, *S* is a substring of *T*



# Suffix trie

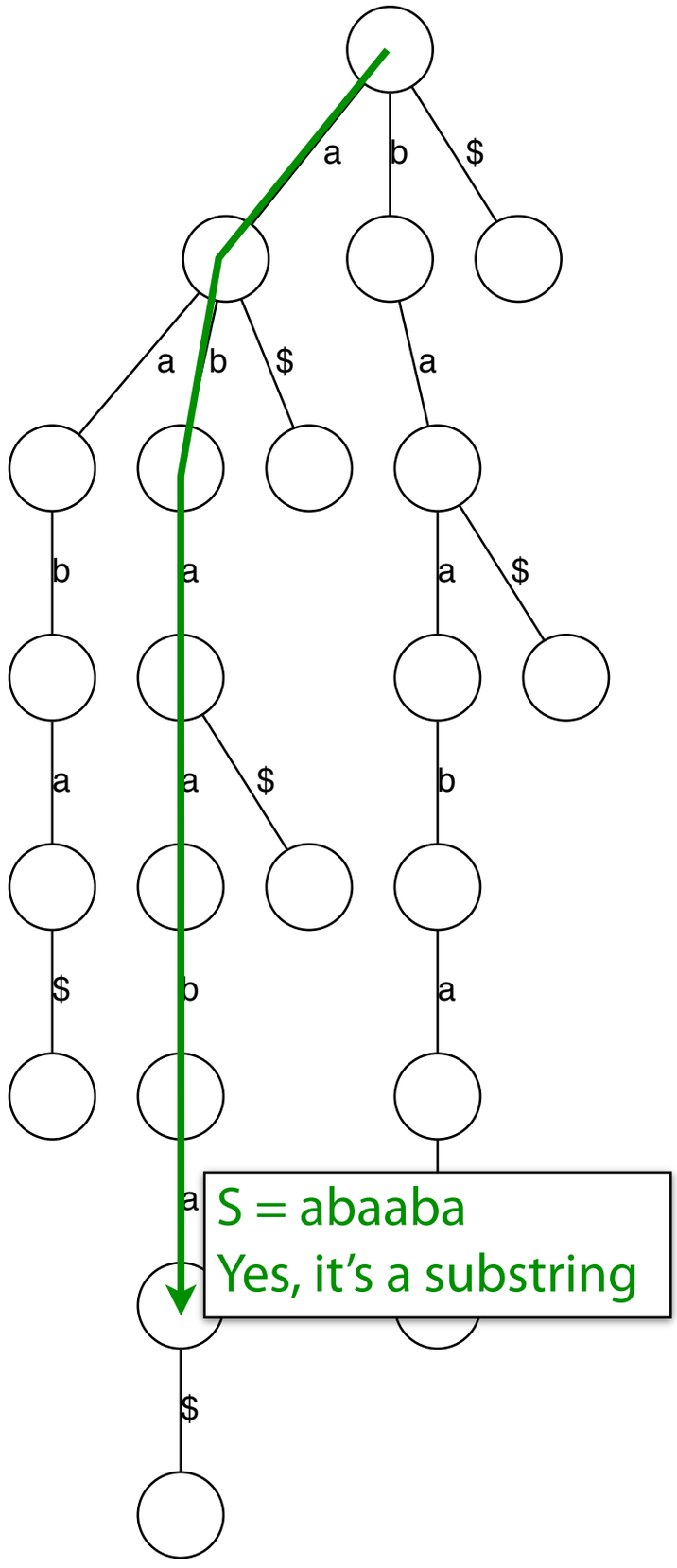
How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

If we exhaust *S* without falling off, *S* is a substring of *T*



# Suffix trie

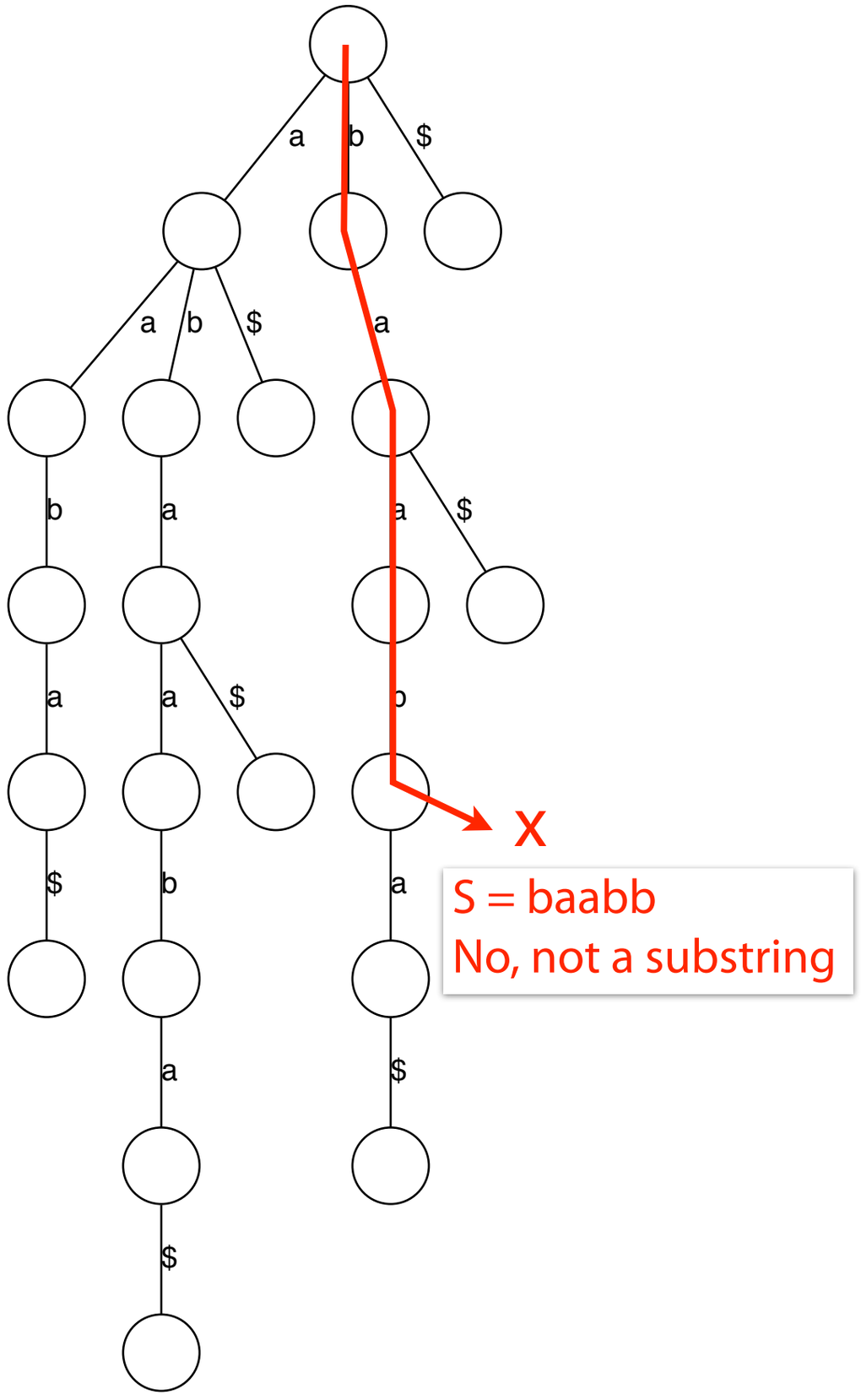
How do we check whether a string *S* is a substring of *T*?

Note: Each of *T*'s substrings is spelled out along a path from the root. I.e., every *substring* is a *prefix* of some *suffix* of T.

Start at the root and follow the edges labeled with the characters of *S*

If we "fall off" the trie -- i.e. there is no outgoing edge for next character of *S*, then *S* is not a substring of *T*

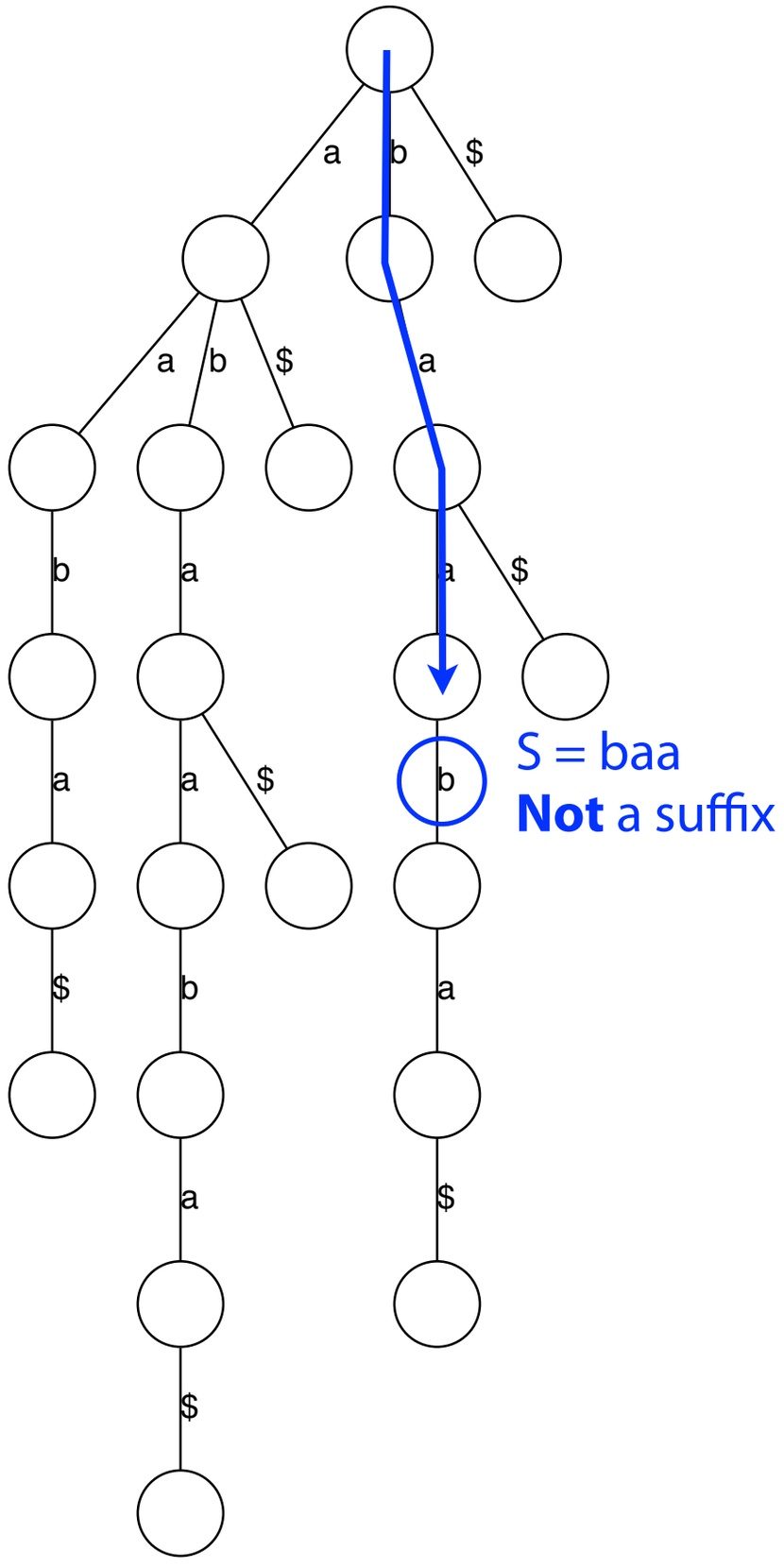
If we exhaust *S* without falling off, *S* is a substring of *T*



# Suffix trie

How do we check whether a string *S* is a **suffix** of *T*?

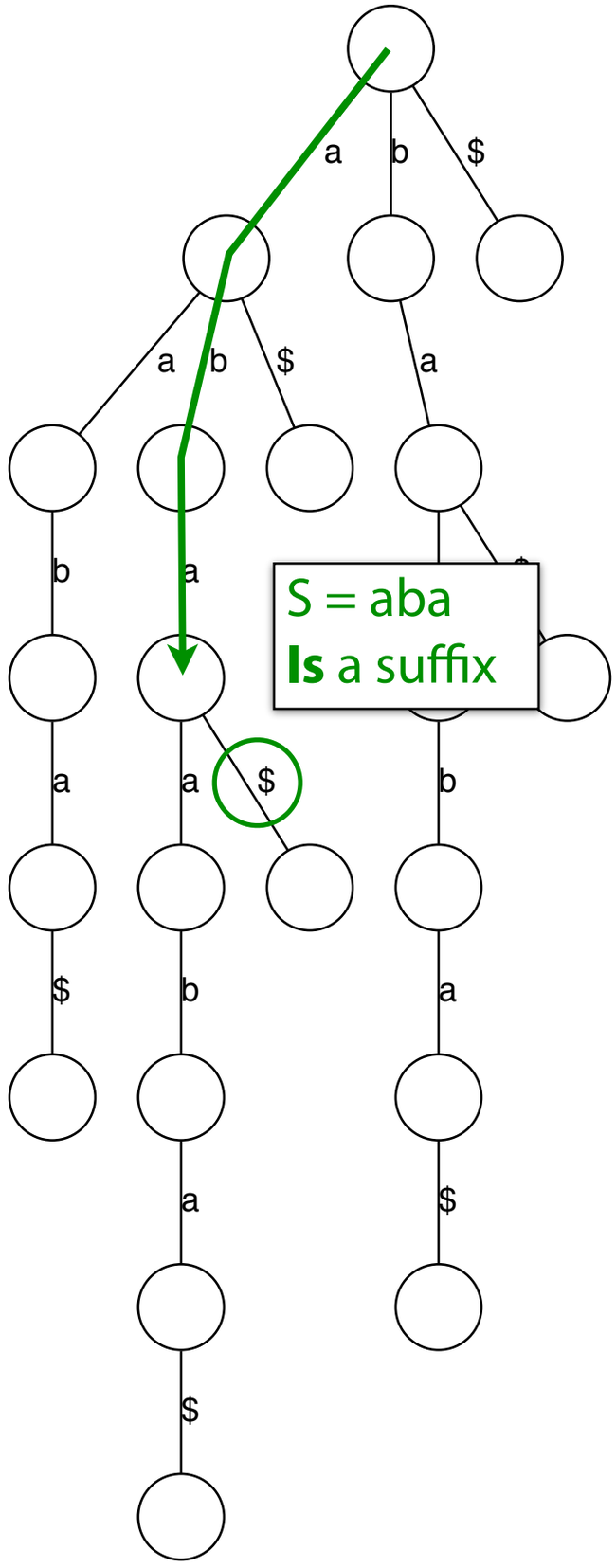
Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled **$**



# Suffix trie

How do we check whether a string *S* is a **suffix** of *T*?

Same procedure as for substring, but additionally check whether the final node in the walk has an outgoing edge labeled **$**

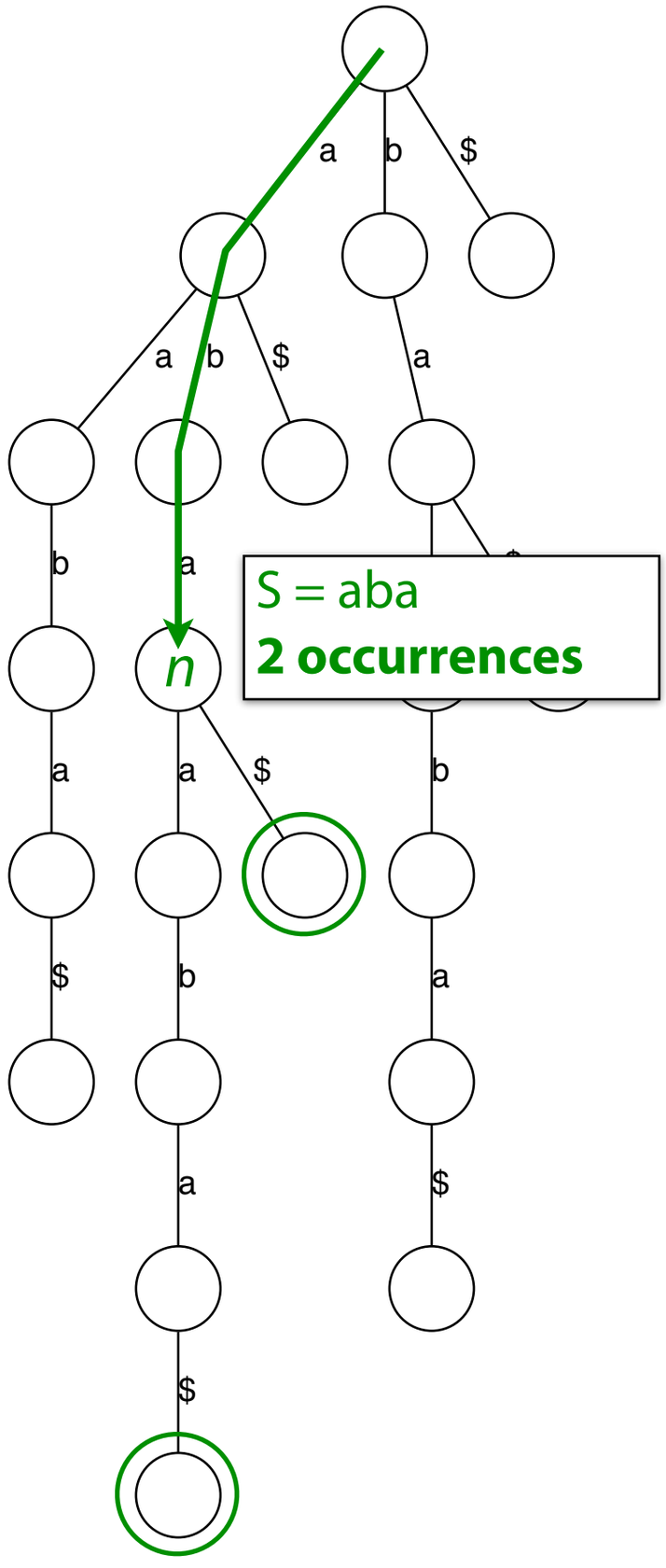


# Suffix trie

How do we count the **number of times** a string *S* occurs as a substring of *T*?

Follow path corresponding to *S*. Either we fall off, in which case answer is 0, or we end up at node *n* and the answer = # of leaf nodes in the subtree rooted at *n*.

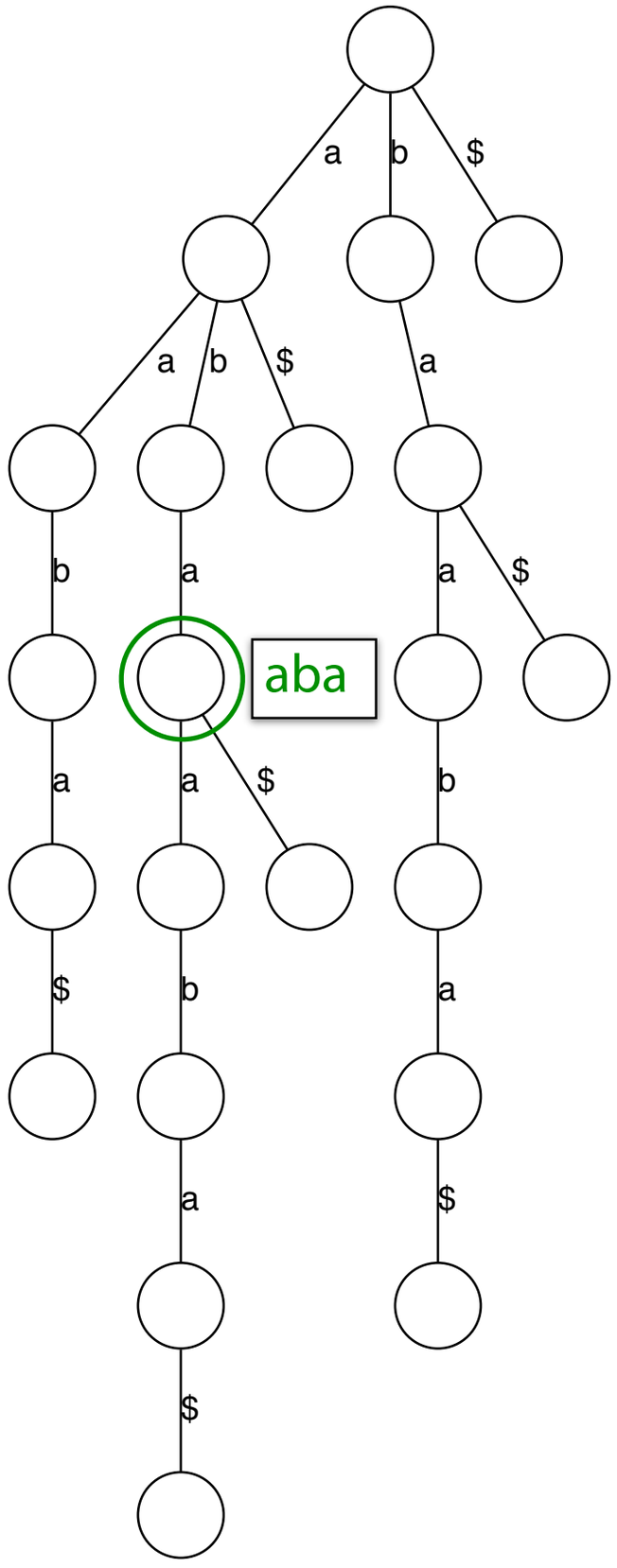
Leaves can be counted with depth-first traversal.



# Suffix trie

How do we find the **longest repeated substring** of *T*?

Find the deepest node with more than one child



# Suffix trie: implementation

```
class SuffixTrie(object):

    def __init__(self, t):
        """ Make suffix trie from t """
        t += '$' # special terminator symbol
        self.root = {}
        for i in xrange(len(t)): # for each suffix
            cur = self.root
            for c in t[i:]: # for each character in i'th suffix
                if c not in cur:
                    cur[c] = {} # add outgoing edge if necessary
                cur = cur[c]
```

```
    def followPath(self, s):
        """ Follow path given by characters of s.  Return node at
            end of path, or None if we fall off. """
        cur = self.root
        for c in s:
            if c not in cur:
                return None
            cur = cur[c]
        return cur

    def hasSubstring(self, s):
        """ Return true iff s appears as a substring of t """
        return self.followPath(s) is not None

    def hasSuffix(self, s):
        """ Return true iff s is a suffix of t """
        node = self.followPath(s)
        return node is not None and '$' in node
```

Python example:
http://nbviewer.ipython.org/6603756

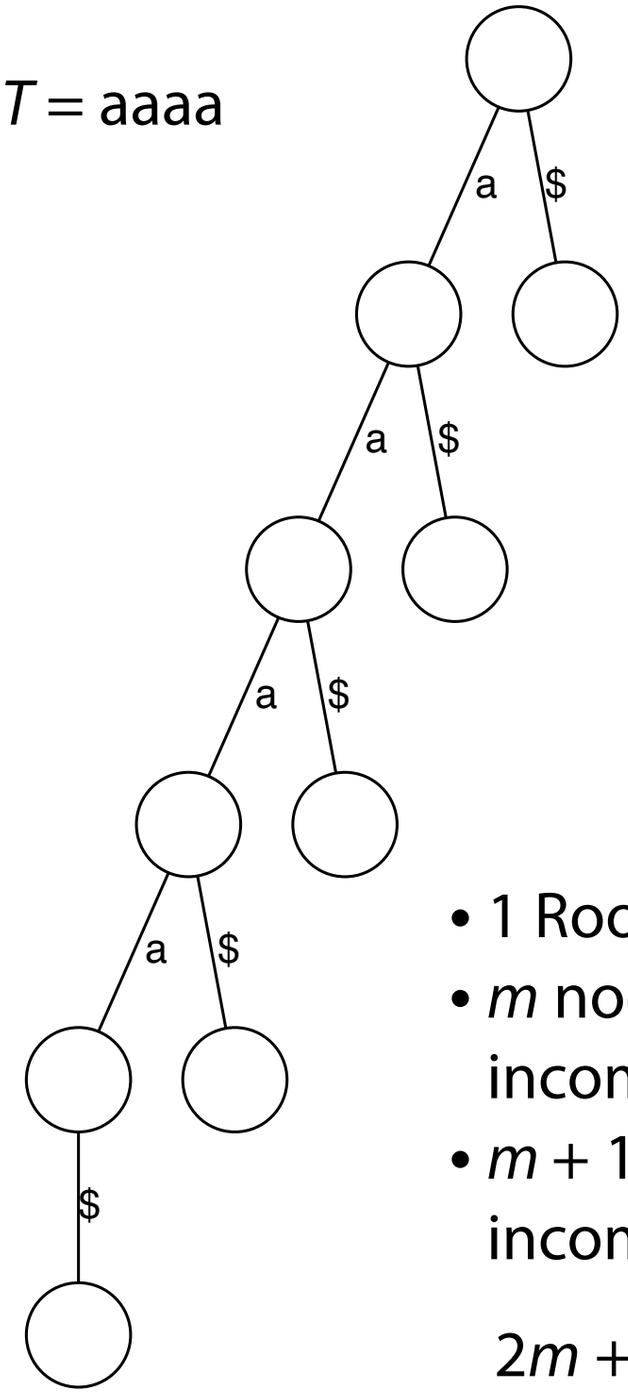
# Suffix trie

How many nodes does the suffix trie have?

Is there a class of string where the number of suffix trie nodes grows linearly with *m*?

Yes: e.g. a string of *m* a's in a row ($a^m$)

$T$ = aaaa




- 1 Root
- *m* nodes with incoming **a** edge
- *m* + 1 nodes with incoming **$** edge

2*m* + 2 nodes

# Suffix trie

Is there a class of string where the number of suffix trie nodes grows with $m^2$?

Yes: $a^n b^n$

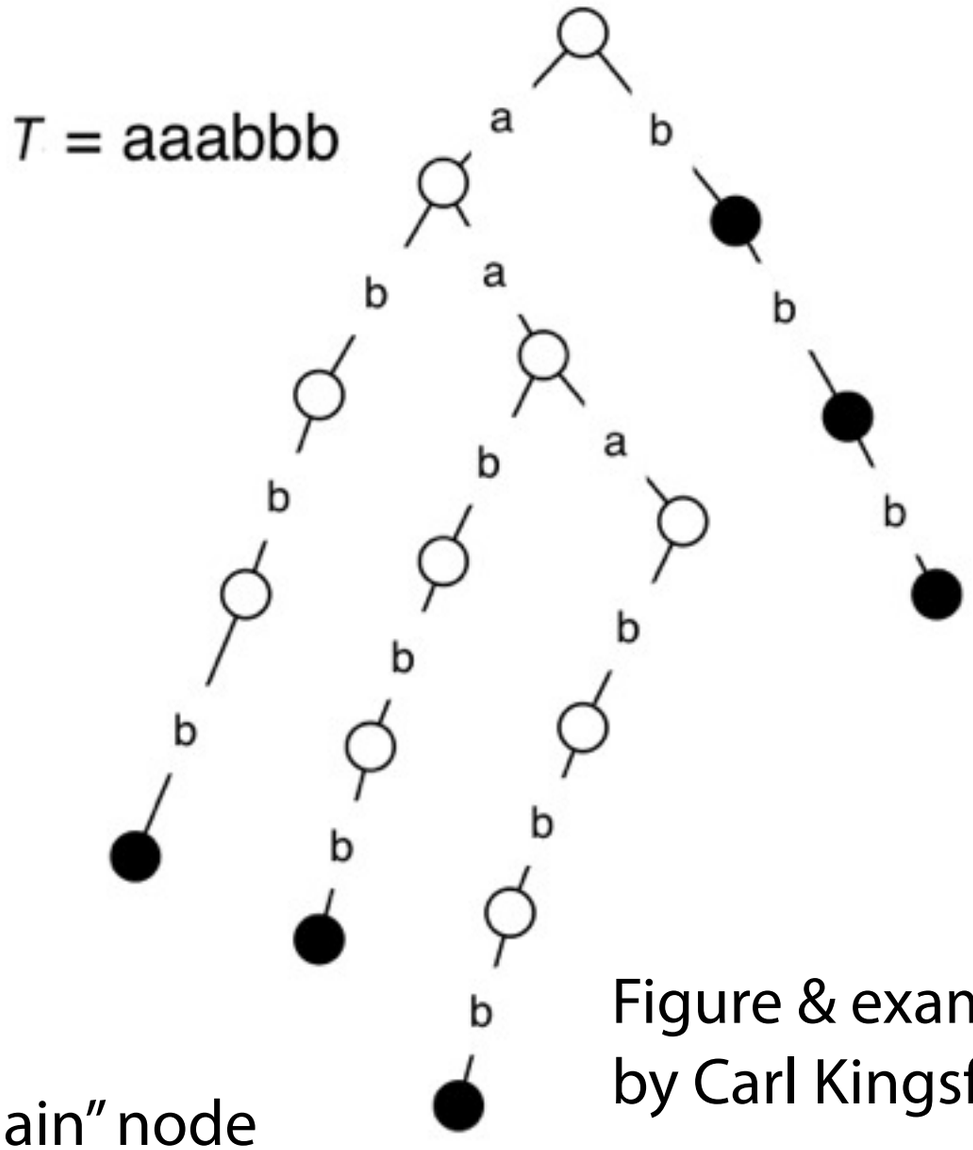


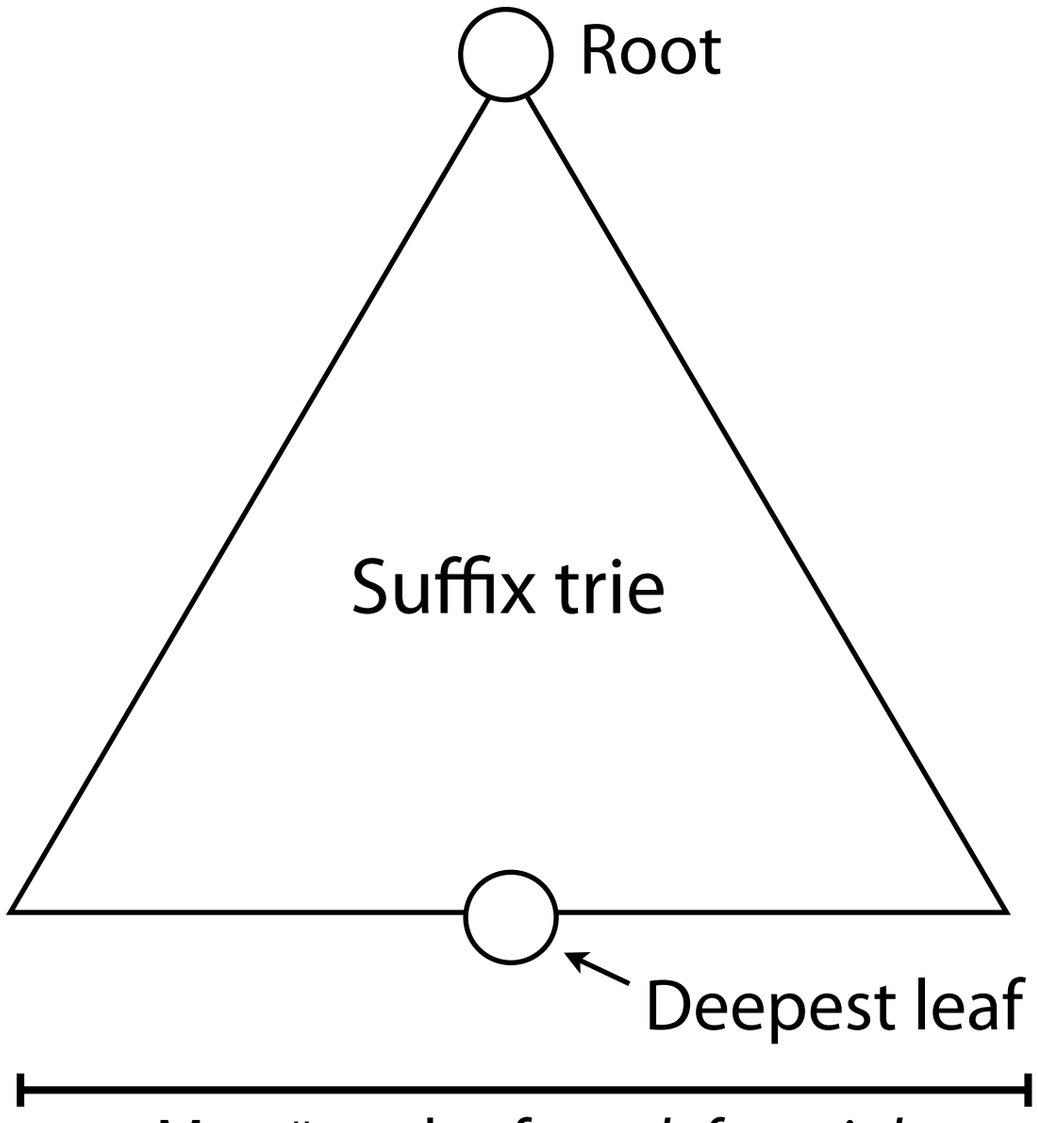

Figure & example by Carl Kingsford

- 1 root
- *n* nodes along "b chain," right
- *n* nodes along "a chain," middle
- *n* chains of *n* "b" nodes hanging off each"a chain" node
- $2n + 1$ **$** leaves (not shown)

$n^2 + 4n + 2$ nodes, where $m = 2n$

# Suffix trie: upper bound on size

Could worst-case # nodes be worse than O($m^2$)?




Max # nodes from *left to right*
= max # distinct substrings of any length
≤ m

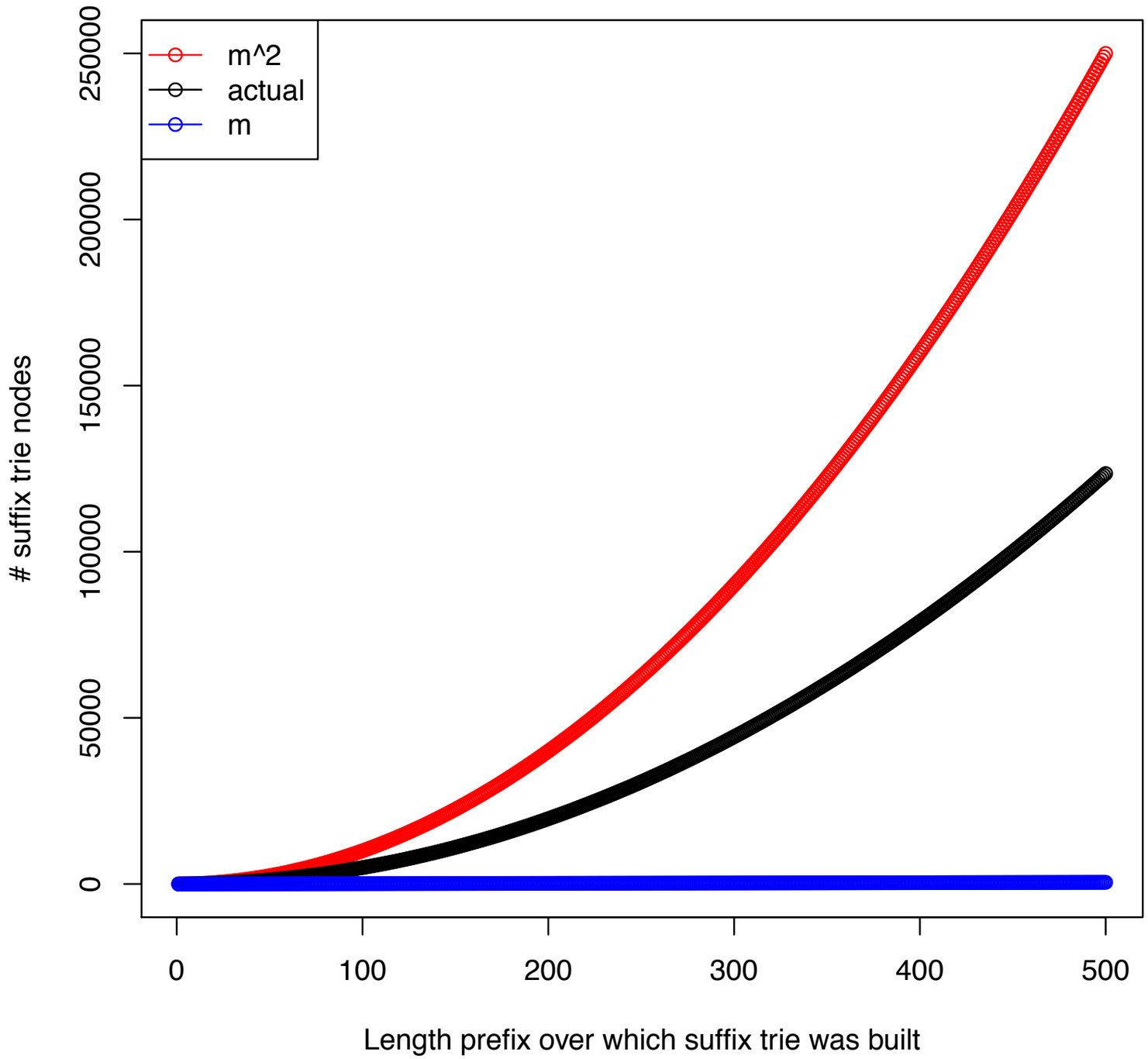
Max # nodes from *top to bottom*
= length of longest suffix + 1
= $m + 1$

$O(m^2)$ is worst case

# Suffix trie: actual growth

Built suffix tries for the first 500 prefixes of the lambda phage virus genome

Black curve shows how # nodes increases with prefix length



# Applications of Suffix Tries (1)

Check whether q is a **substring** of T:

Follow the path for q starting from the root.
If you exhaust the query string, then q is in T.

Check whether q is a **suffix** of T:

Follow the path for q starting from the root.
If you end at a leaf at the end of q, then q is a suffix of T

Count # of occurrences of q in T:

Follow the path for q starting from the root.
The number of leaves under the node you end up in is the number of occurrences of q.

Find the longest repeat in T:

Find the deepest node that has at least 2 leaves under it.

Find the lexicographically (alphabetically) first suffix:

Start at the root, and follow the edge labeled with the lexicographically (alphabetically) smallest letter.