

# Suffix Trees

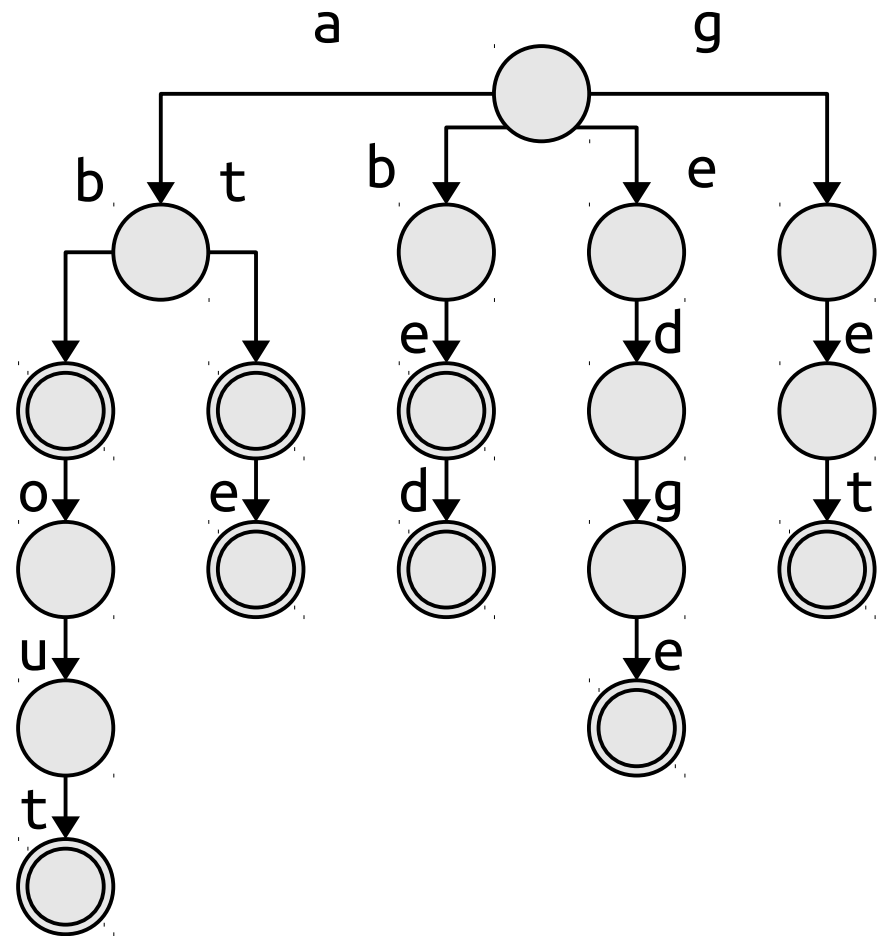
# Outline for Today

- **Review from Last Time**
  - A quick refresher on tries.
- **Suffix Tries**
  - A simple data structure for string searching.
- **Suffix Trees**
  - A compact, powerful, and flexible data structure for string algorithms.
- **Generalized Suffix Trees**
  - An even more flexible data structure.

Review from Last Time

# Tries

- A **trie** is a tree that stores a collection of strings over some alphabet  $\Sigma$ .
- Each node corresponds to a prefix of some string in the set.
- Tries are sometimes called **prefix trees**, since each node in a trie corresponds to a prefix of one of the words in the trie.



# Aho-Corasick String Matching

- The ***Aho-Corasick string matching algorithm*** is an algorithm for finding all occurrences of a set of strings  $P_1, \dots, P_k$  inside a string  $T$ .
- Runtime is  $\langle O(n), O(m + z) \rangle$ , where
  - $m = |T|$ ,
  - $n = |P_1| + \dots + |P_k|$ , and
  - $z$  is the number of matches.
- Great for the case where the patterns are fixed and the text to search changes.

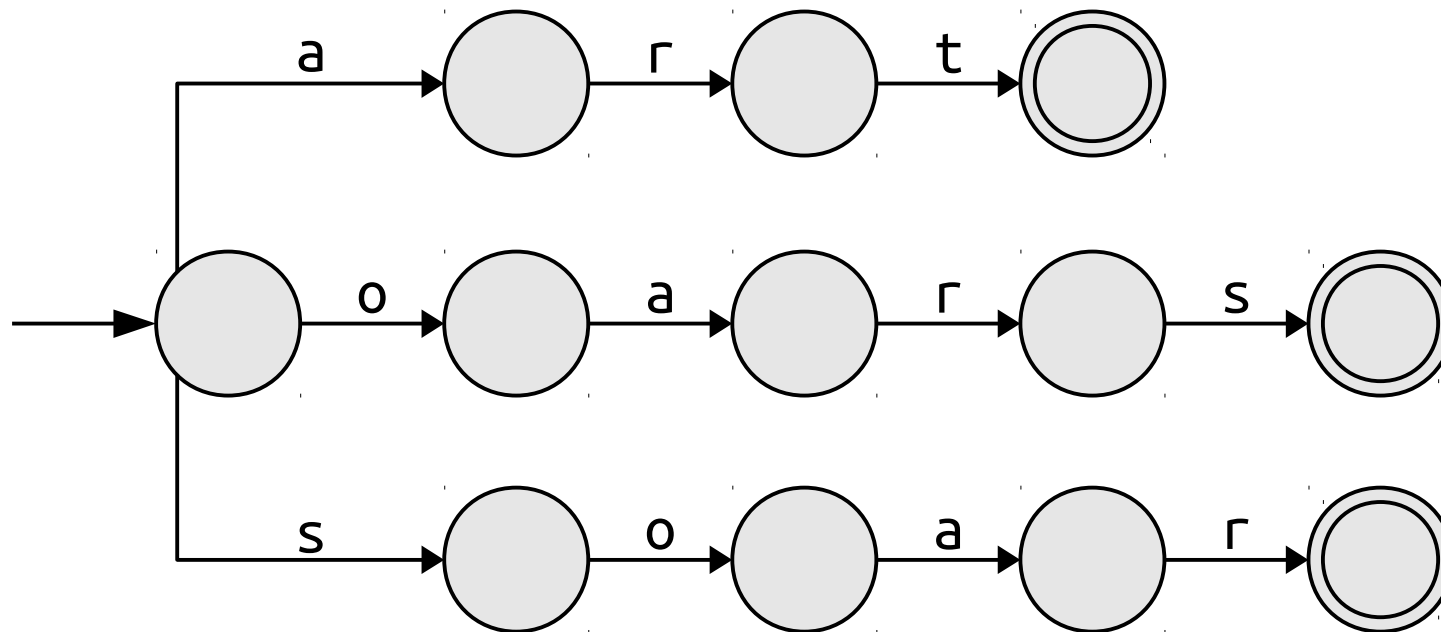
# Genomics Databases

- Many string algorithms these days are developed for or used extensively in computational genomics.
- Typically, we have a *huge* database with many very large strings (genomes) that we'll preprocess to speed up future operations.
- **Common problem:** given a fixed string  $T$  to search and changing patterns  $P_1, \dots, P_k$ , find all matches of those patterns in  $T$ .
- **Question:** Can we instead preprocess  $T$  to make it easy to search for variable patterns?

# Suffix Tries

# Substrings, Prefixes, and Suffixes

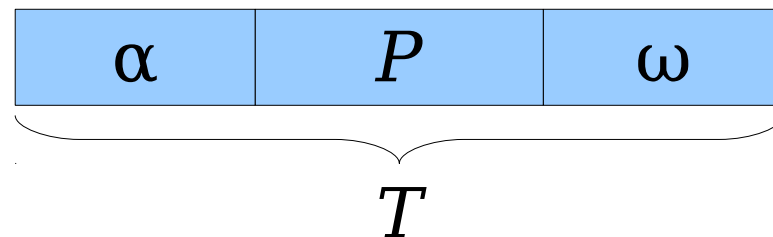
- **Useful Fact 1:** Given a trie storing a set of strings  $S_1, S_2, \dots, S_k$ , it's possible to determine, in time  $O(|Q|)$ , whether a query string  $Q$  is a prefix of any  $S_i$ .





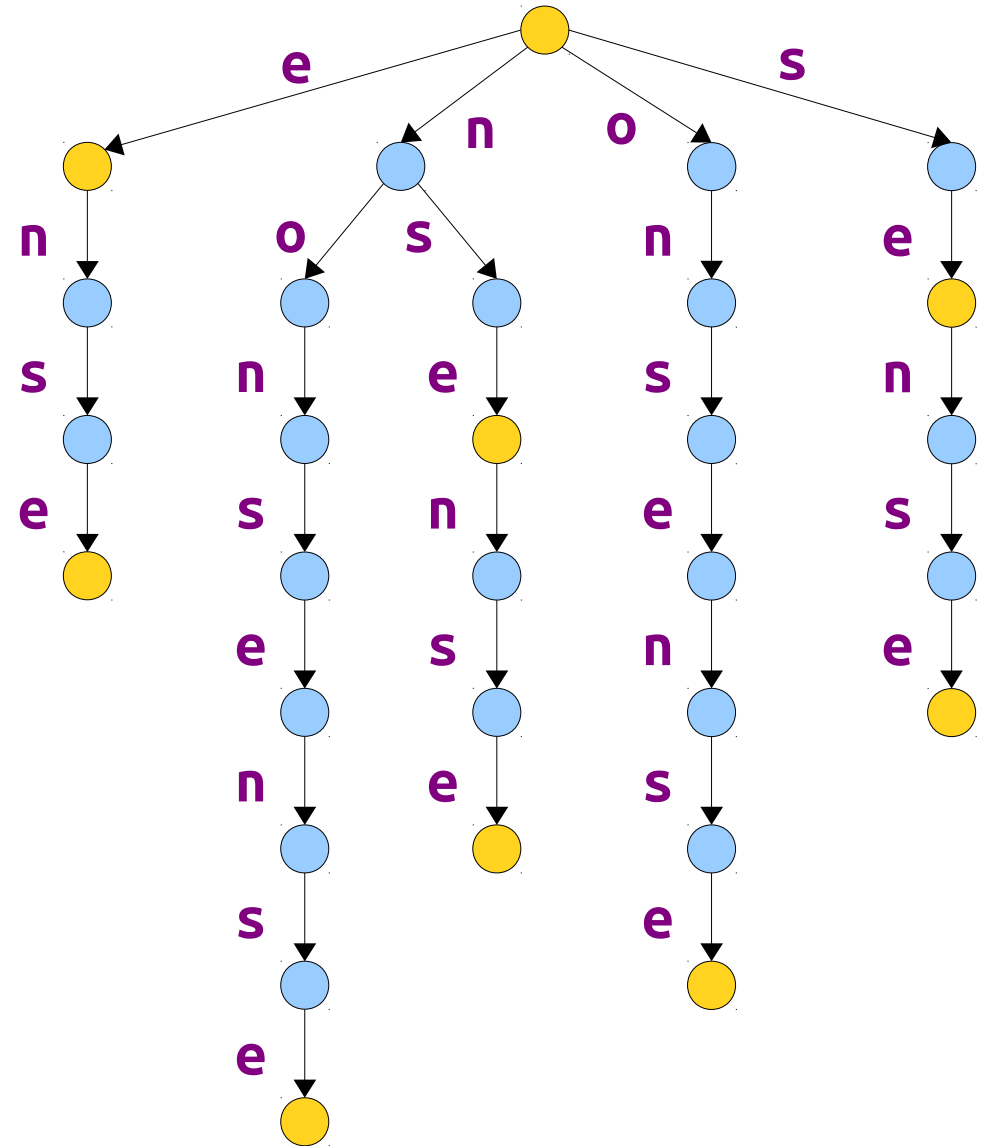
# Substrings, Prefixes, and Suffixes

- **Useful Fact 1:** Given a trie storing a set of strings  $S_1, S_2, \dots, S_k$ , it's possible to determine, in time  $O(|Q|)$ , whether a query string  $Q$  is a prefix of any  $S_i$ .
- **Useful Fact 2:** A string  $P$  is a substring of a string  $T$  if and only if  $P$  is a prefix of some suffix of  $T$ .
  - Specifically, write  $T = \alpha P \omega$ ; then  $T$  is a prefix of the suffix  $P\omega$  of  $T$ .



# Suffix Tries

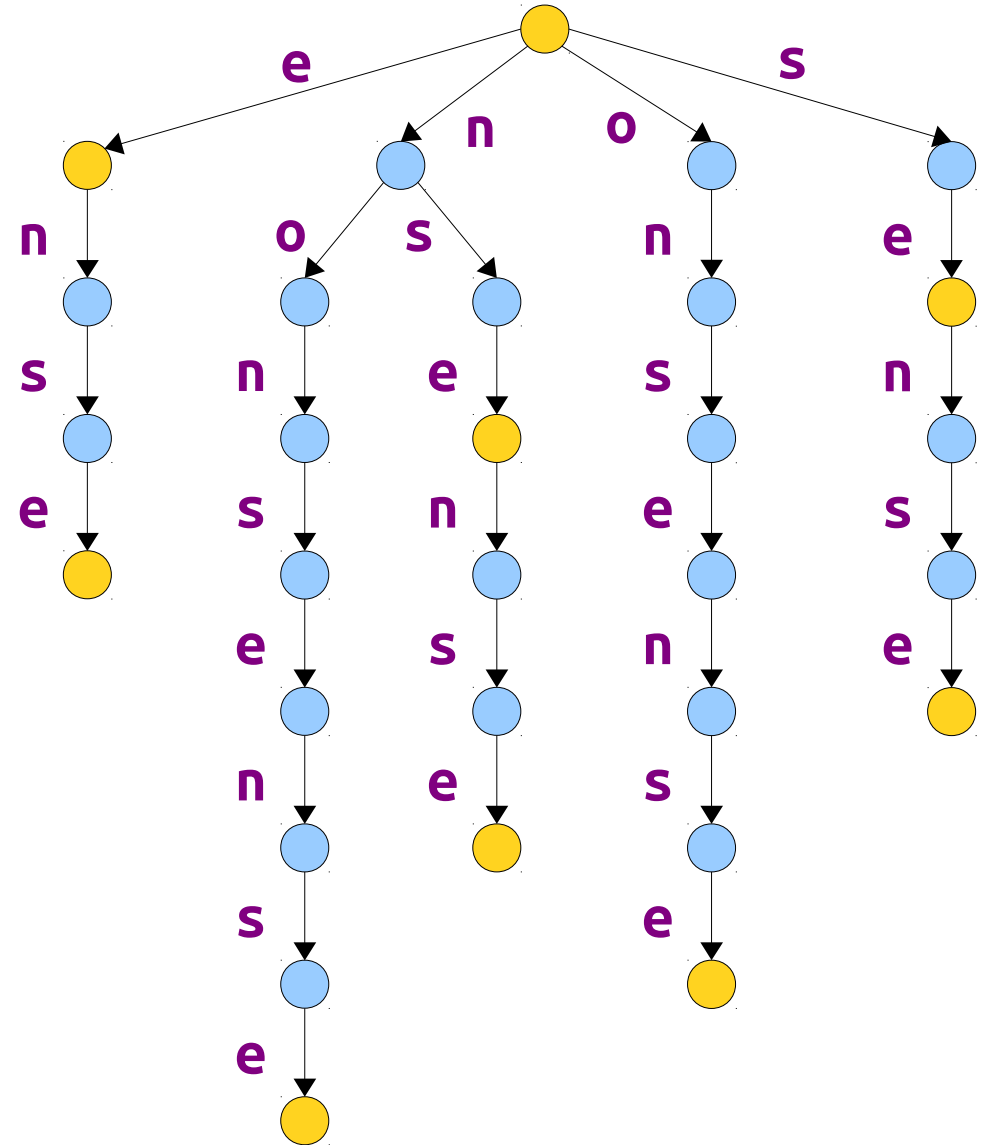
- A **suffix trie** of  $T$  is a trie of all the suffixes of  $T$ .
- Given any pattern string  $P$ , we can check in time  $O(|P|)$  whether  $P$  is a substring of  $T$  by seeing whether  $P$  is a prefix in  $T$ 's suffix trie.
  - (Because that means that  $P$  is a prefix of a suffix of  $T$ .)



nonsense

# Suffix Tries

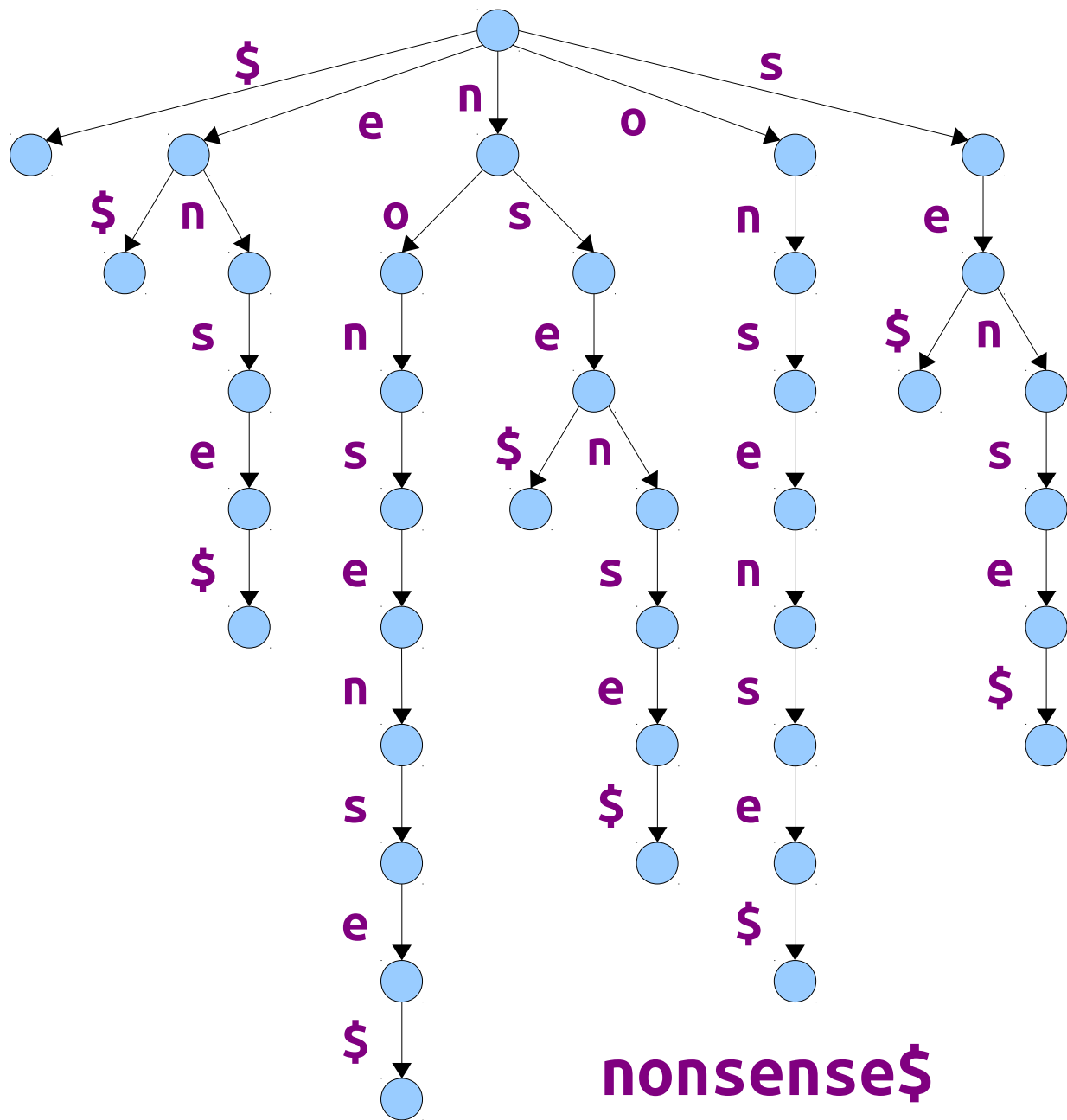
- A **suffix trie** of  $T$  is a trie of all the suffixes of  $T$ .
- More generally, given any nonempty patterns  $P_1, \dots, P_k$  of total length  $n$ , we can detect how many of those patterns are substrings of  $T$  in time  $O(n)$ .
- (Finding all matches is a bit trickier; more on that later.)



**nonsense**

# A Typical Transform

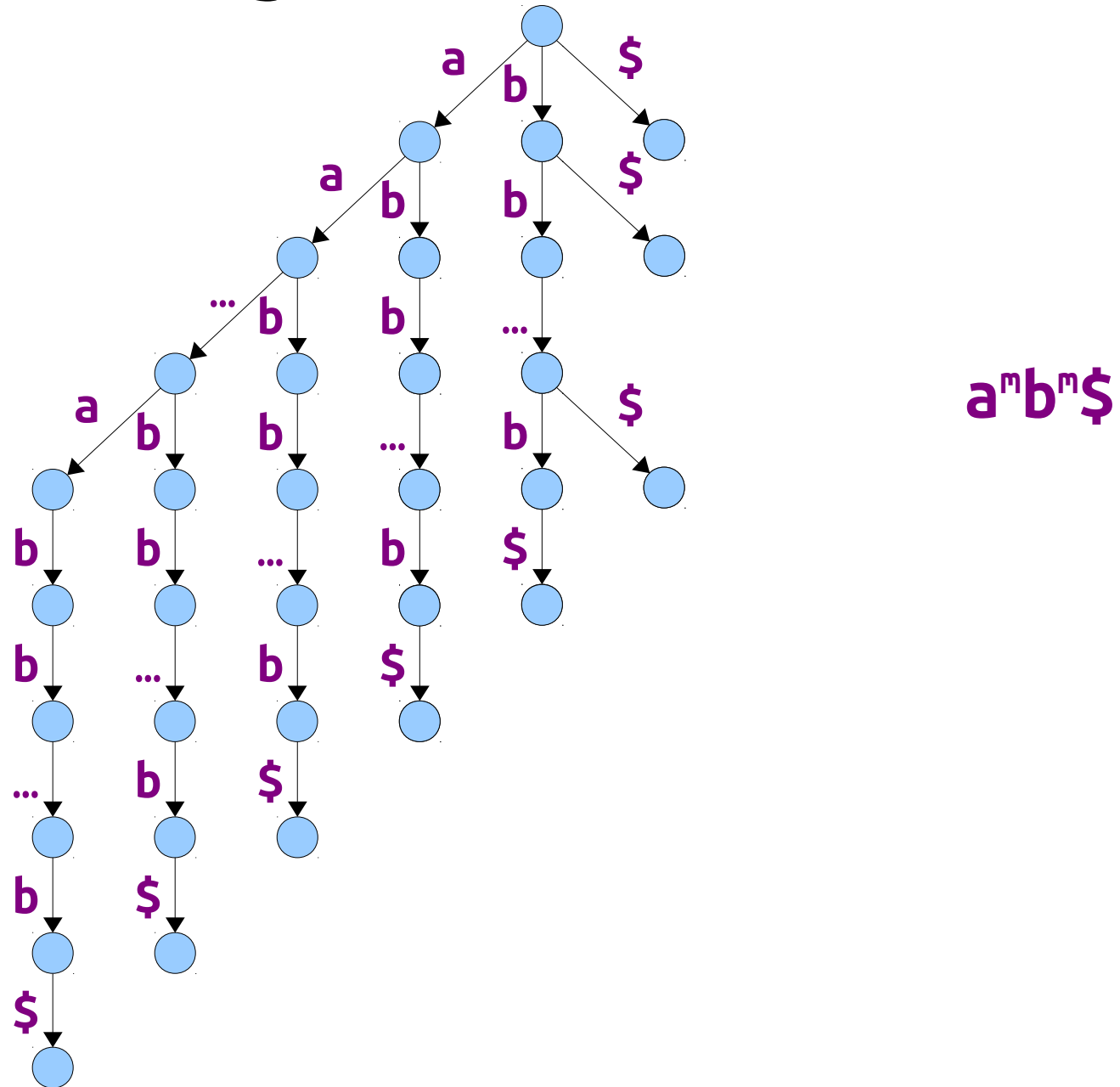
- Typically, we append some new character  $\$ \notin \Sigma$  to the end of  $T$ , then construct the trie for  $T\$$ .
- Leaf nodes correspond to suffixes.
- Internal nodes correspond to prefixes of those suffixes.



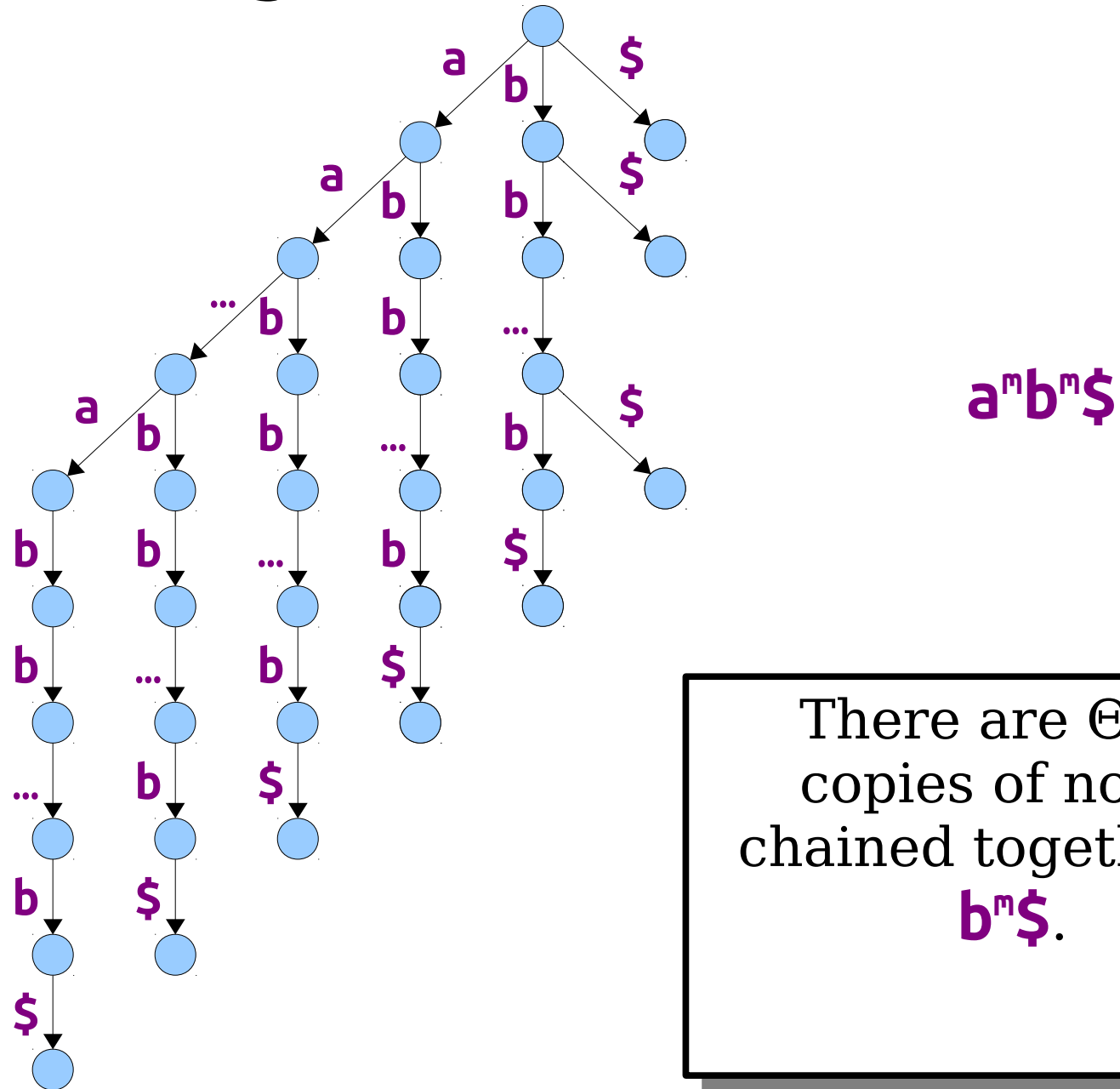
# Constructing Suffix Tries

- Once we build a single suffix trie for string  $T$ , we can efficiently detect whether patterns match in time  $O(n)$ .
- **Question:** How long does it take to construct a suffix trie?
- **Problem:** There's an  $\Omega(m^2)$  lower bound on the worst-case complexity of *any* algorithm for building suffix tries.

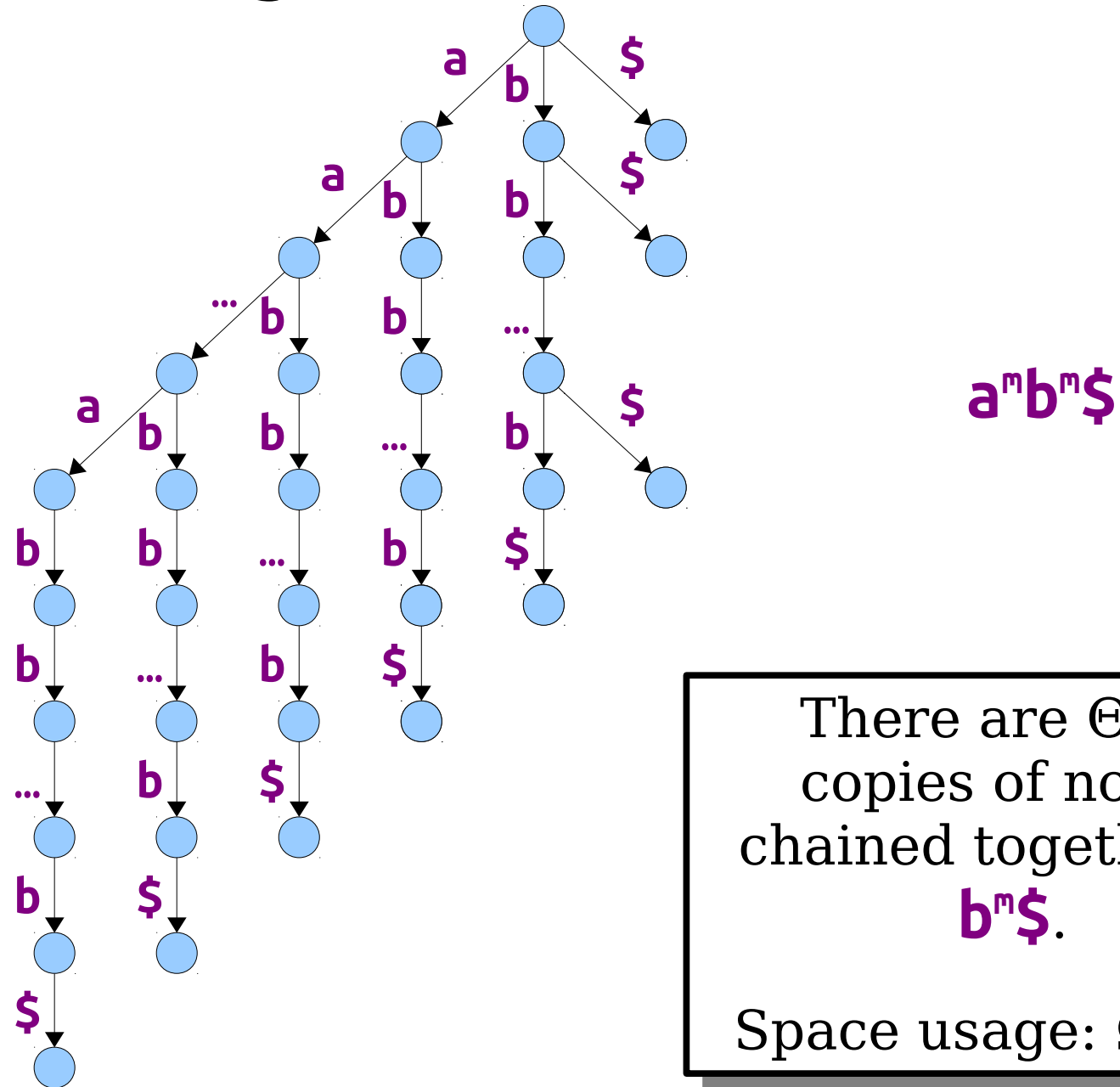
# A Degenerate Case



# A Degenerate Case



# A Degenerate Case



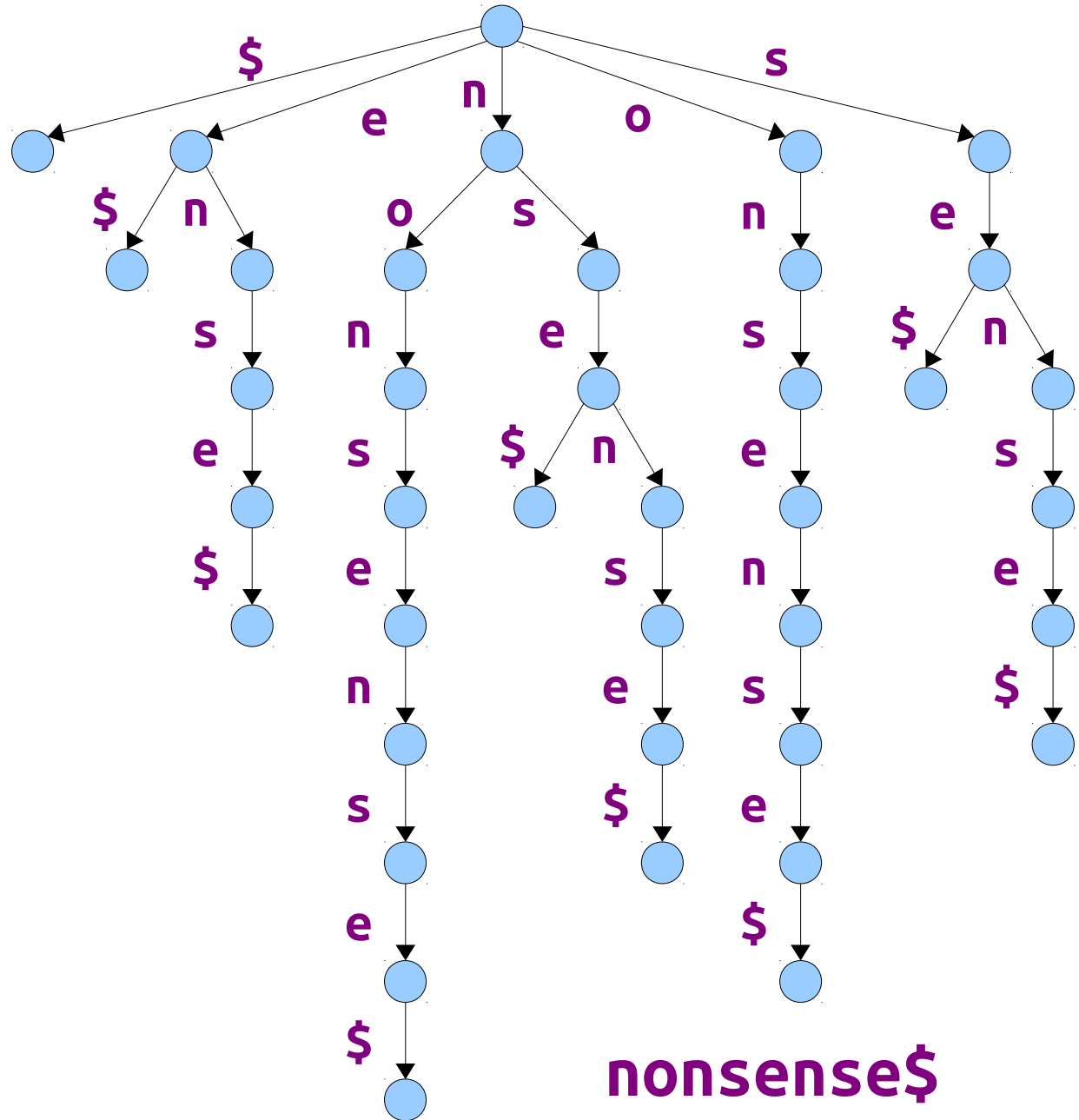


# Correcting the Problem

- Because suffix tries may have  $\Omega(m^2)$  nodes, all suffix trie algorithms must run in time  $\Omega(m^2)$  in the worst-case.
- Can we reduce the number of nodes in the trie?

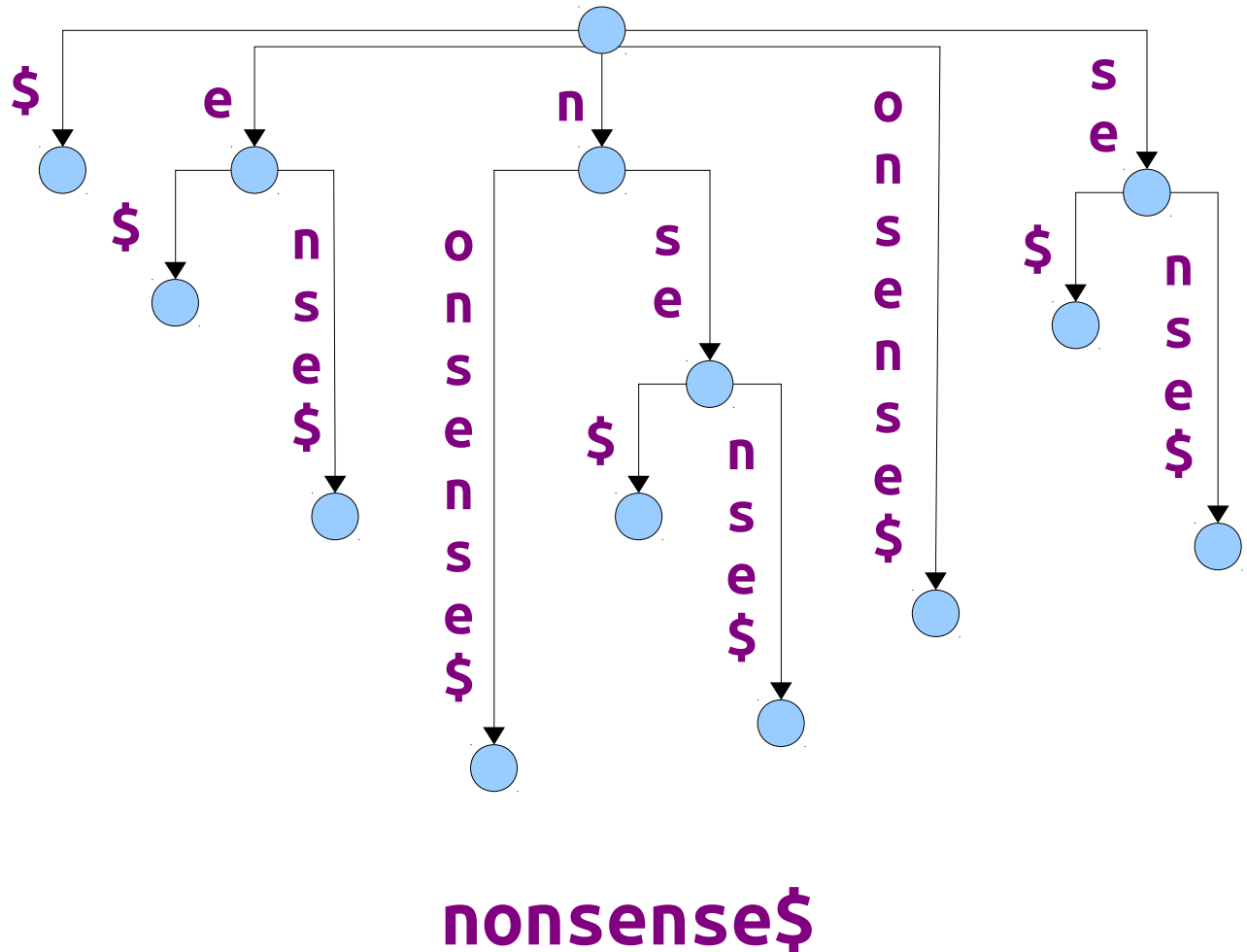
# Patricia Tries

- A “silly” node in a trie is a node that has exactly one child.
- A ***Patricia trie*** (or ***radix trie***) is a trie where all “silly” nodes are merged with their parents.



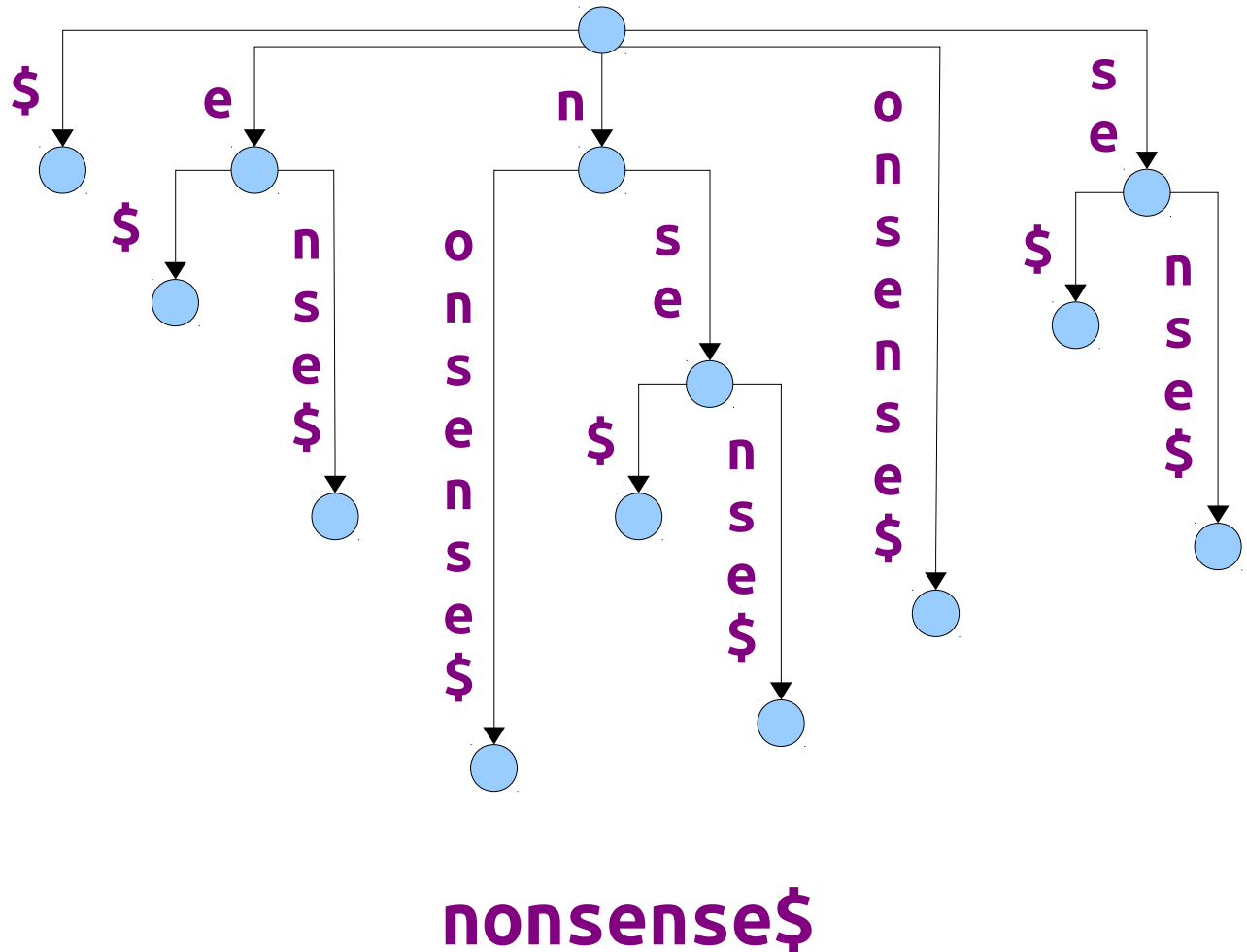
# Patricia Tries

- A “silly” node in a trie is a node that has exactly one child.
- A *Patricia trie* (or *radix trie*) is a trie where all “silly” nodes are merged with their parents.



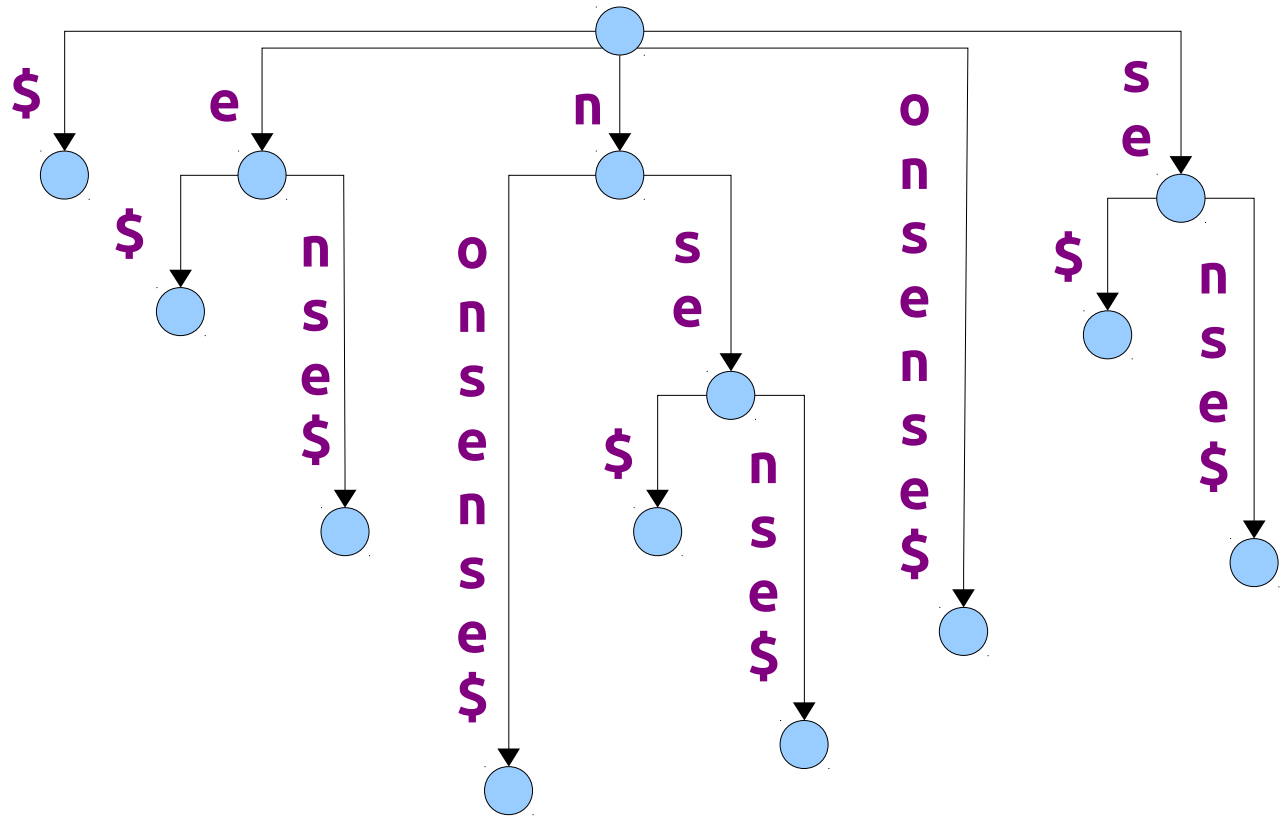
# Suffix Trees

- A **suffix tree** for a string  $T$  is an Patricia trie of  $T\$$  where each leaf is labeled with the index where the corresponding suffix starts in  $T\$$ .



# Suffix Trees

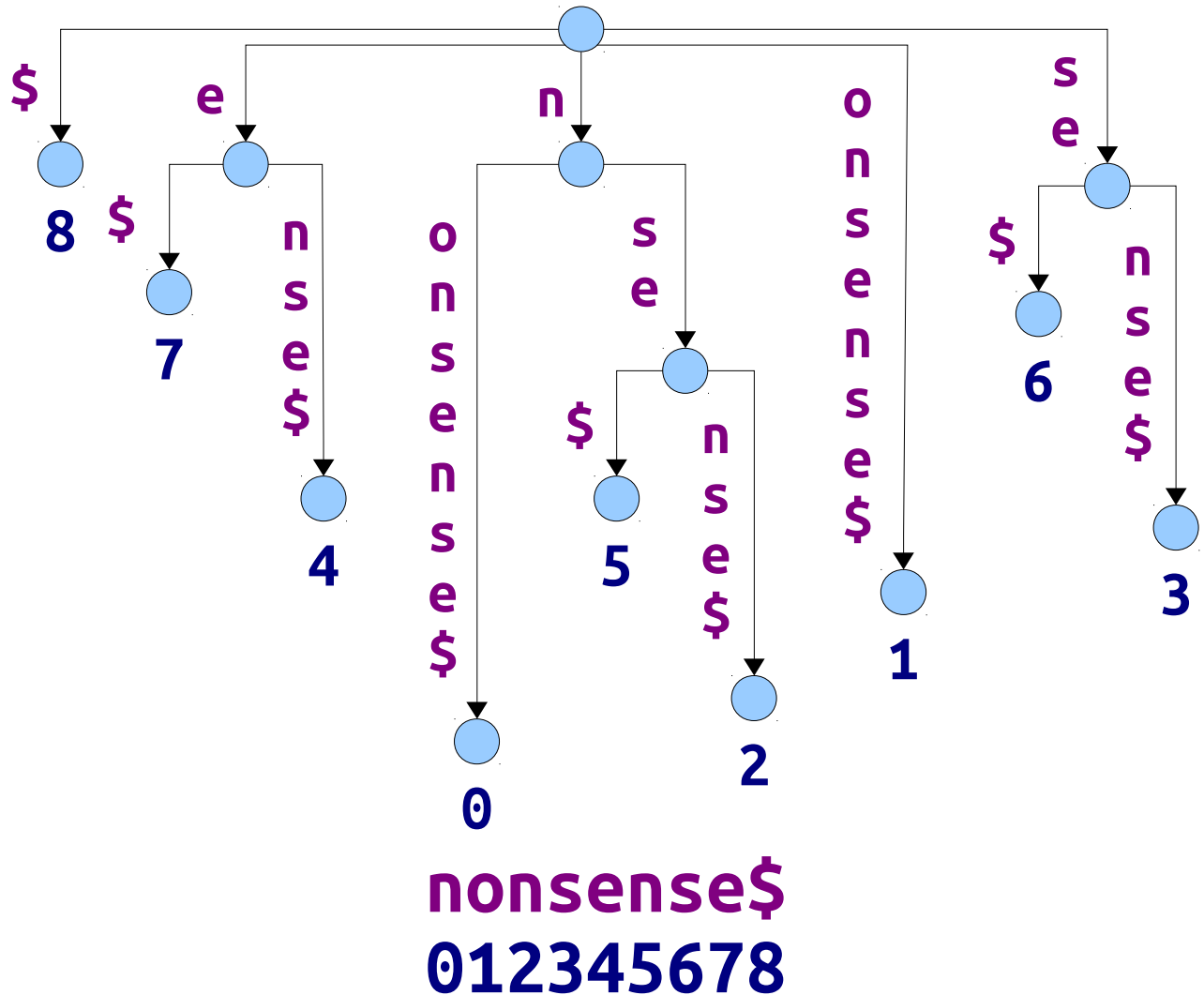
- A **suffix tree** for a string  $T$  is an Patricia trie of  $T\$$  where each leaf is labeled with the index where the corresponding suffix starts in  $T\$$ .



nonsense\$  
012345678

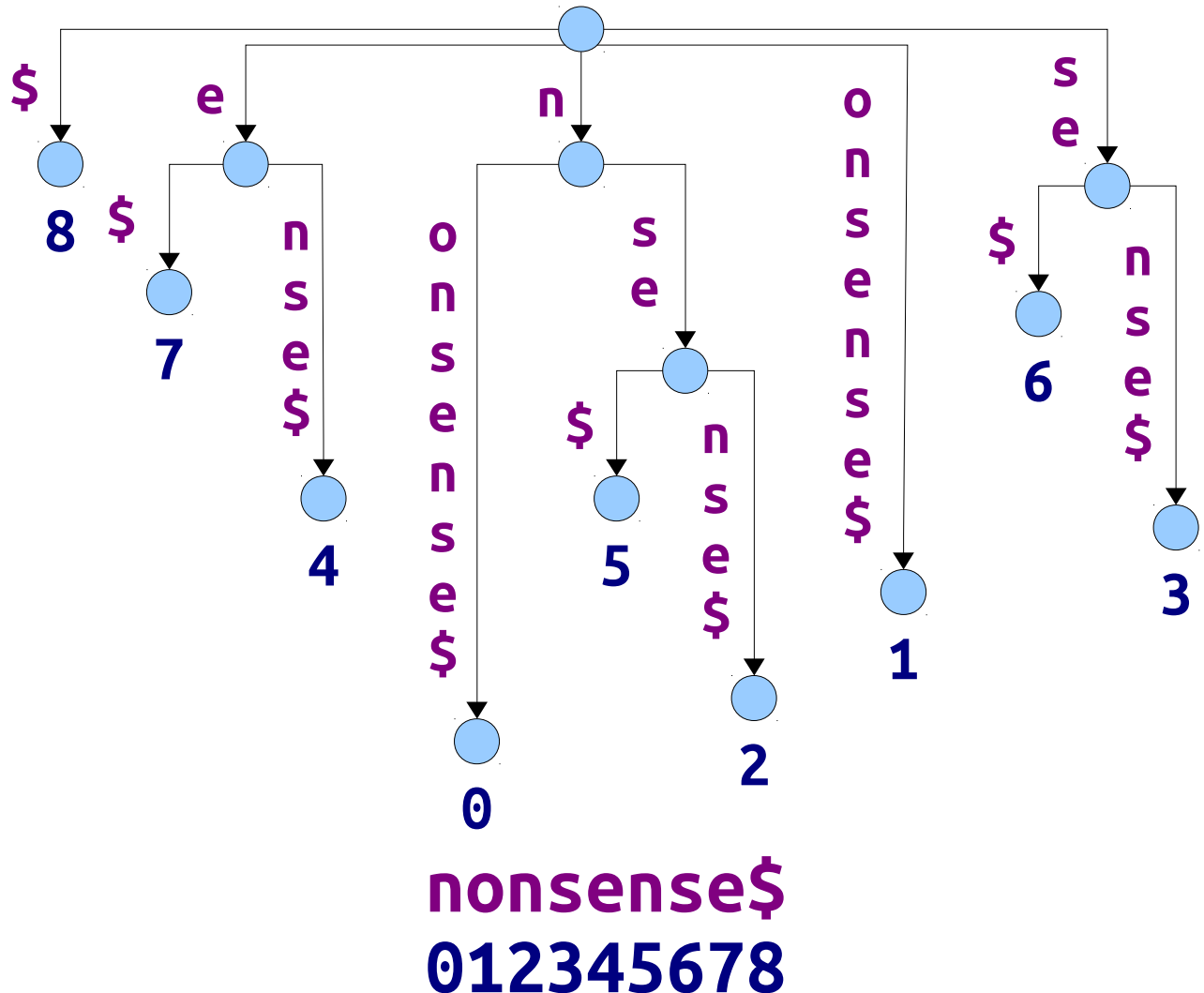
# Suffix Trees

- A **suffix tree** for a string  $T$  is an Patricia trie of  $T\$$  where each leaf is labeled with the index where the corresponding suffix starts in  $T\$$ .



# Properties of Suffix Trees

- If  $|T| = m$ , the suffix tree has exactly  $m + 1$  leaf nodes.
- For any  $T \neq \varepsilon$ , all internal nodes in the suffix tree have at least two children.
- Number of nodes in a suffix tree is  $\Theta(m)$ .

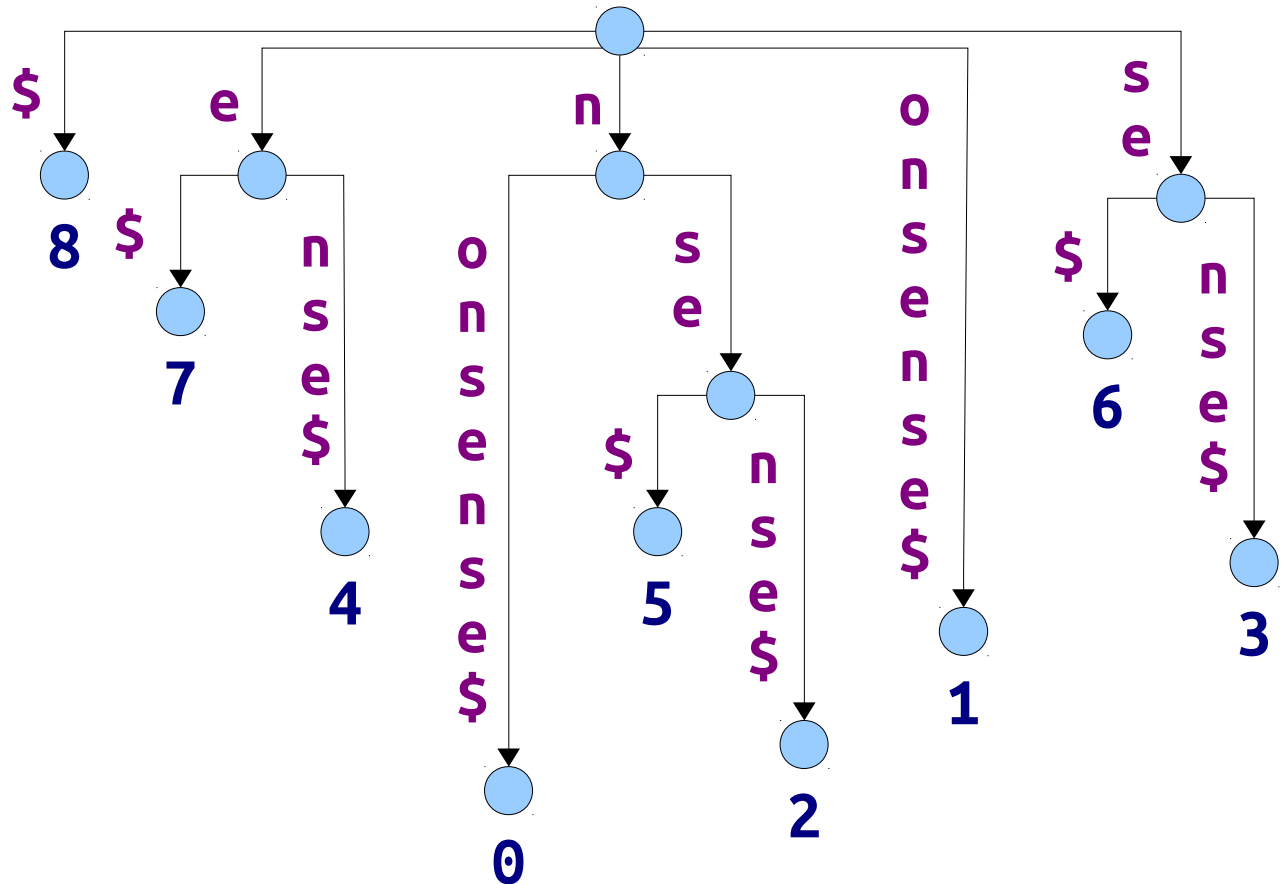


# Suffix Tree Representations

- Suffix trees may have  $\Theta(m)$  nodes, but the labels on the edges can have size  $\omega(1)$ .
- This means that a naïve representation of a suffix tree may take  $\omega(m)$  space.
- **Useful fact:** Each edge in a suffix tree is labeled with a consecutive range of characters from  $w$ .
- **Trick:** Represent each edge labeled with a string  $\alpha$  as a pair of integers [start, end] representing where in the string  $\alpha$  appears.

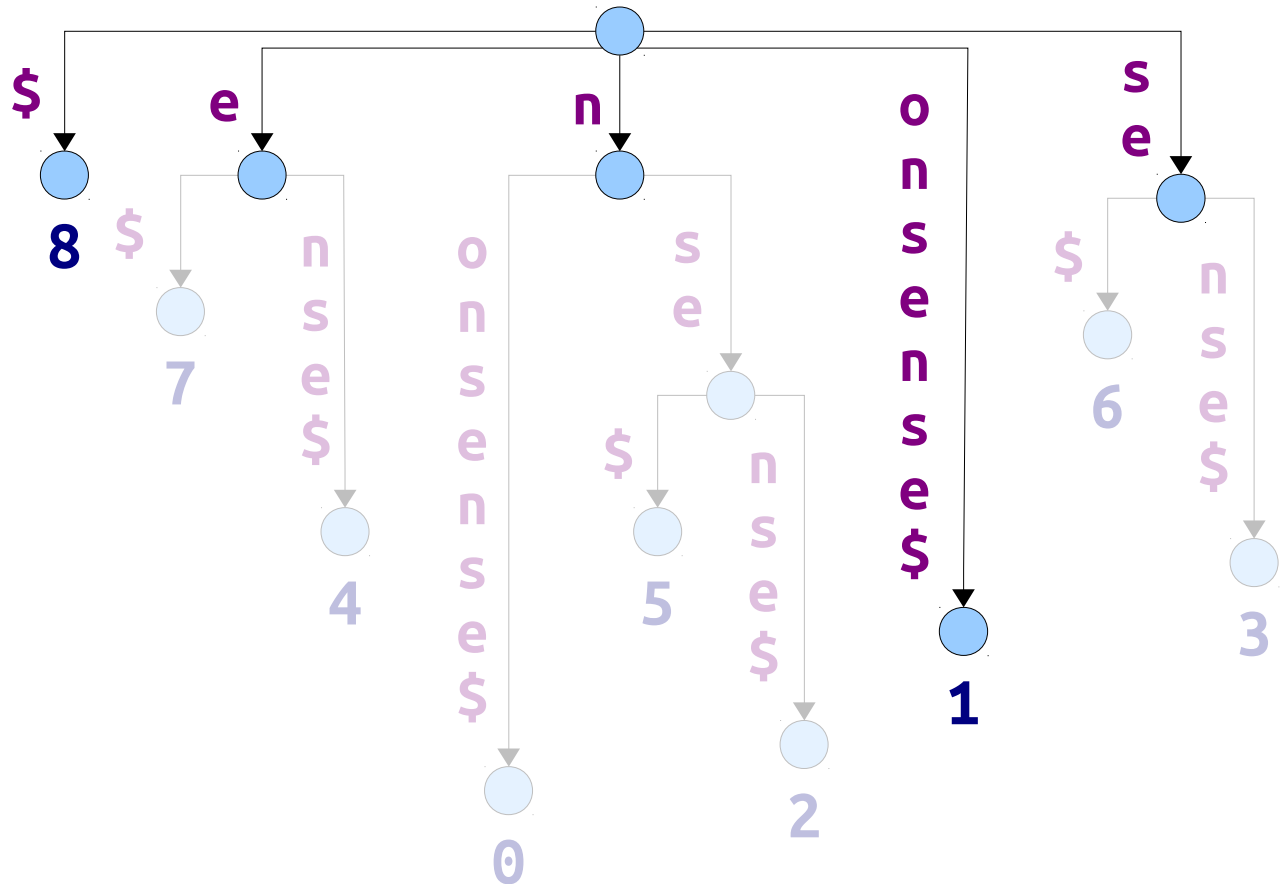


# Suffix Tree Representations



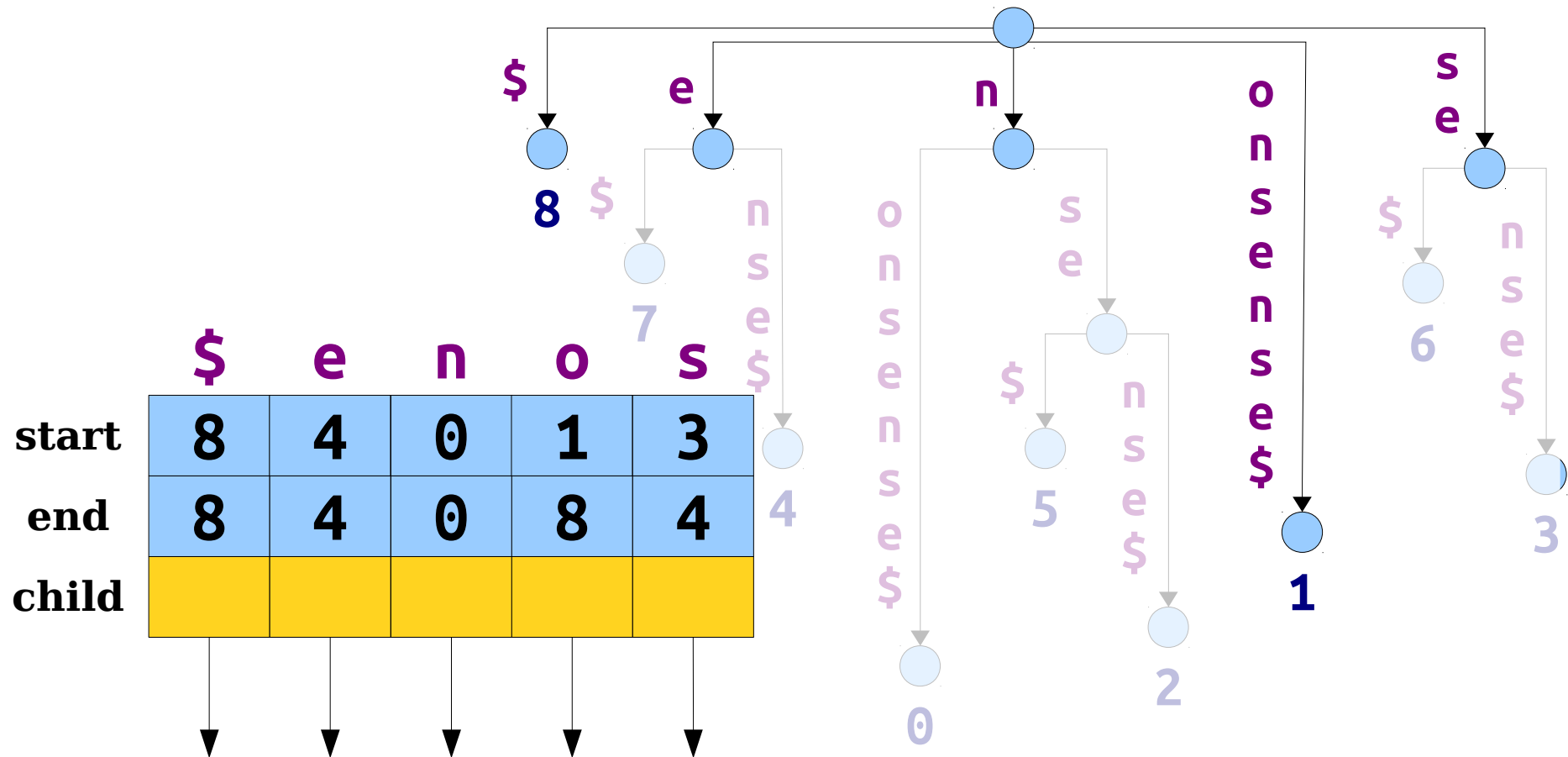
nonsense\$  
012345678

# Suffix Tree Representations



nonsense\$  
012345678

# Suffix Tree Representations



nonsense\$  
012345678

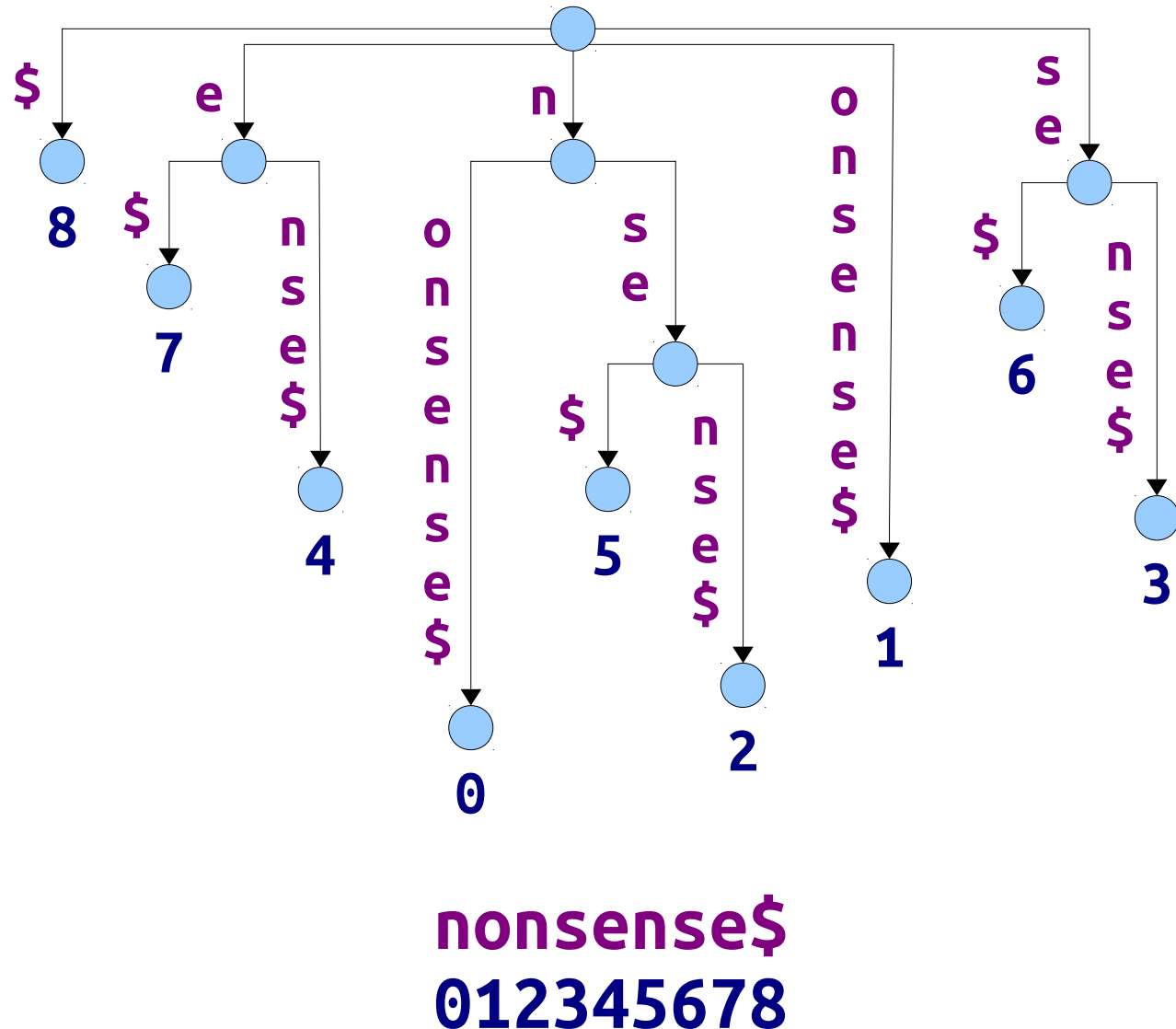
# Building Suffix Trees

- Using this representation, suffix trees can be constructed using space  $\Theta(m)$ .
- ***Claim:*** There are  $\Theta(m)$ -time algorithms for building suffix trees.
- *These algorithms are not trivial!* We'll discuss one of them next time.

***Application:*** Multi-String Matching

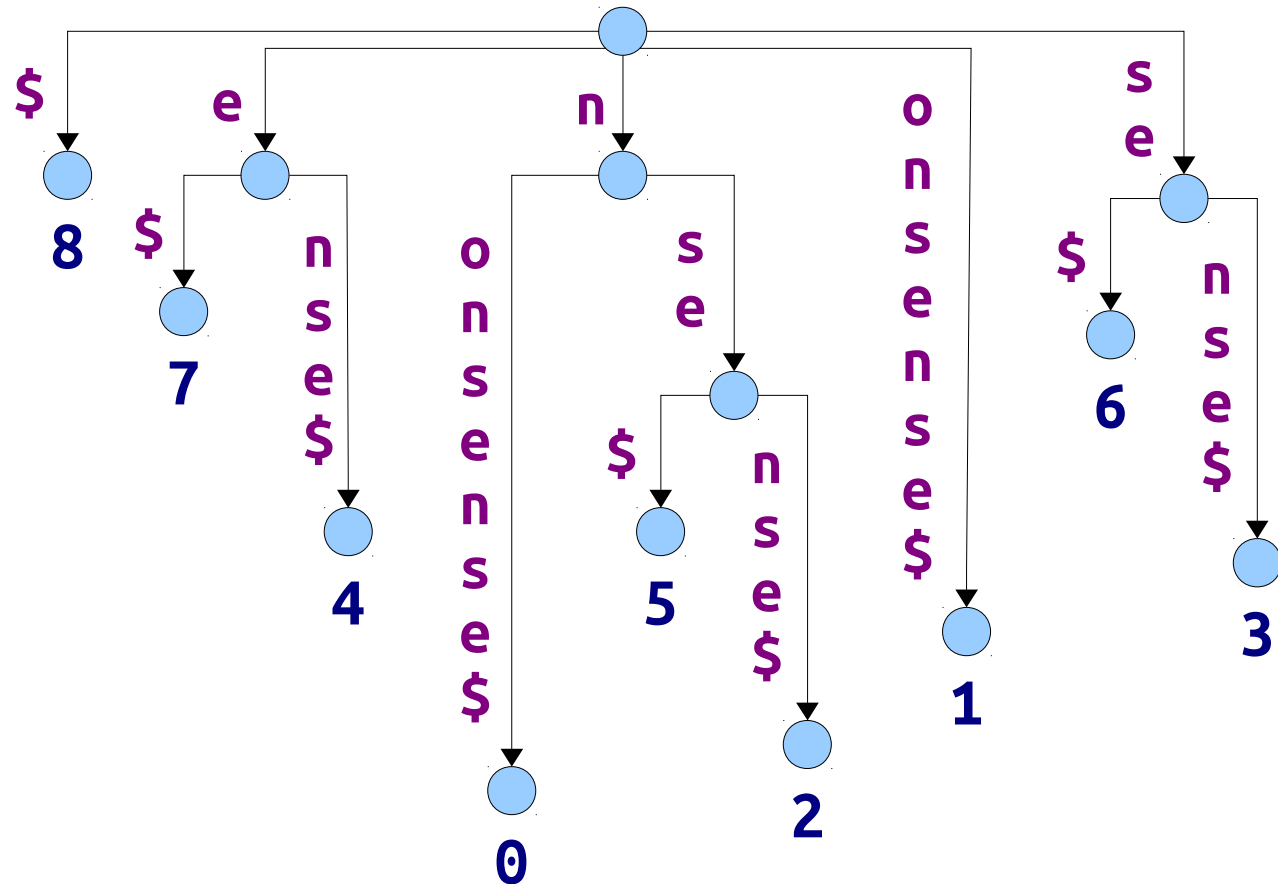
# String Matching

- Suppose we preprocess a string  $T$  by building a suffix tree for it.
- Given any pattern string  $P$  of length  $n$ , we can determine, in time  $O(n)$ , whether  $n$  is a substring of  $P$  by looking it up in the suffix tree.



# String Matching

- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.

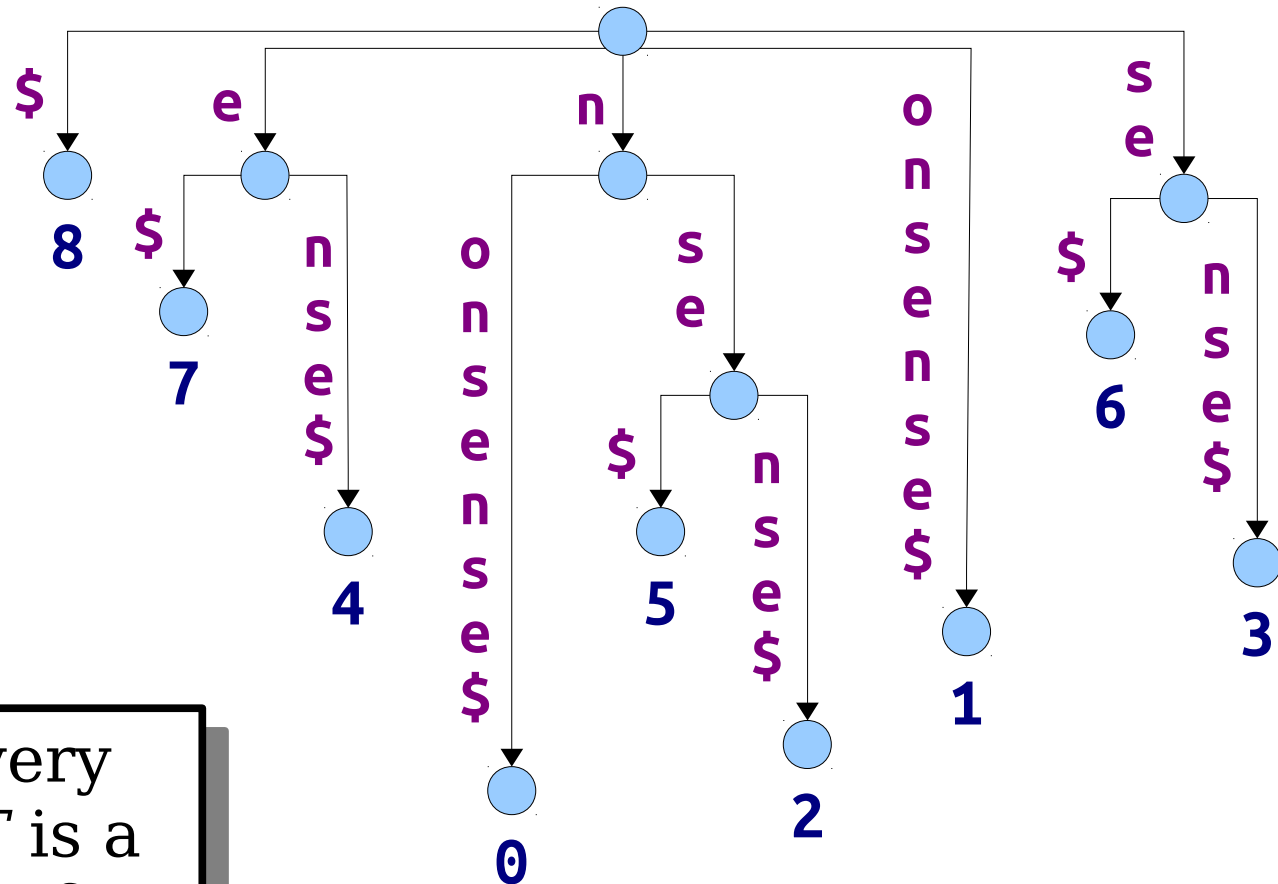


nonsense\$  
012345678

# String Matching

- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.

**Observation 1:** Every occurrence of  $P$  in  $T$  is a prefix of some suffix of  $T$ .

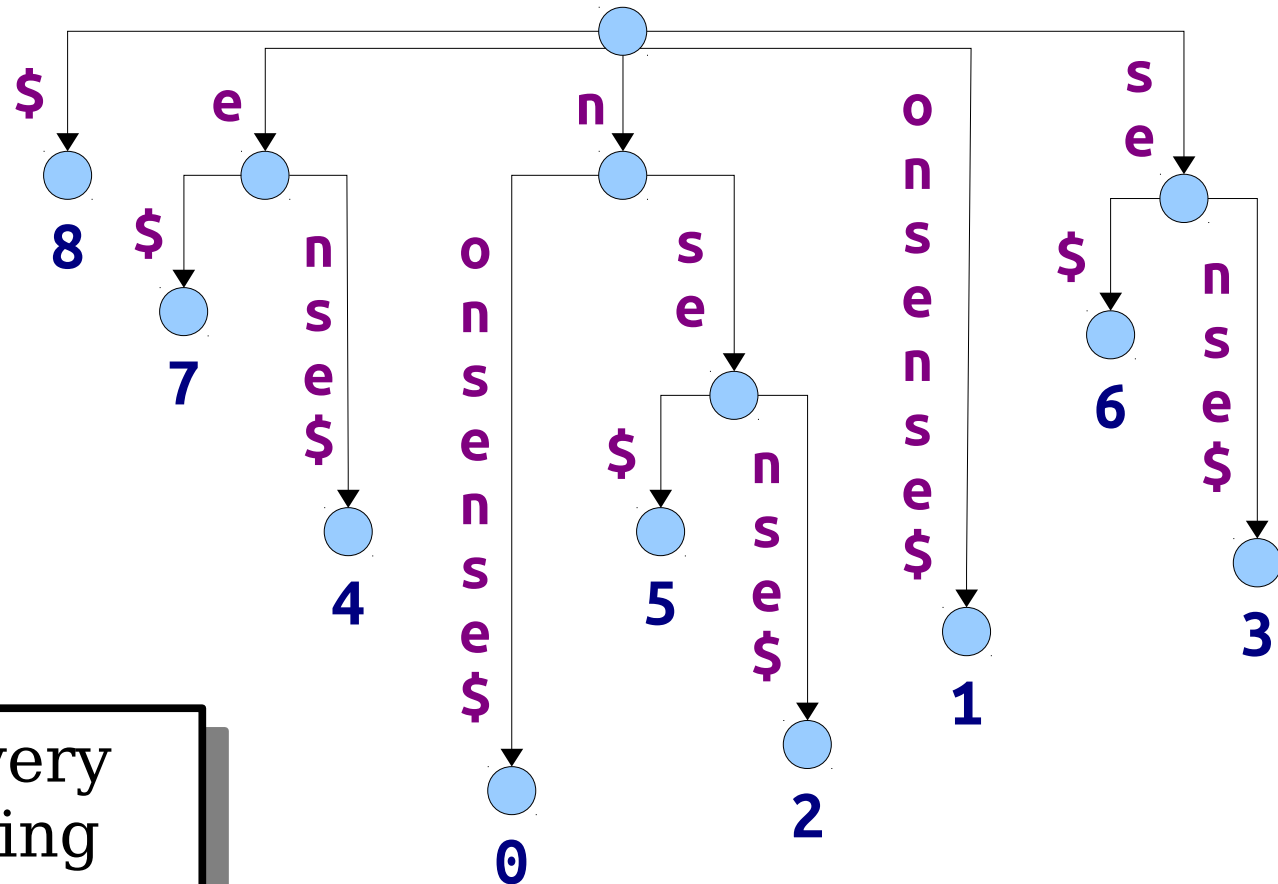


nonsense\$  
012345678



# String Matching

- **Claim:** After spending  $O(m)$  time preprocessing  $T\$$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.

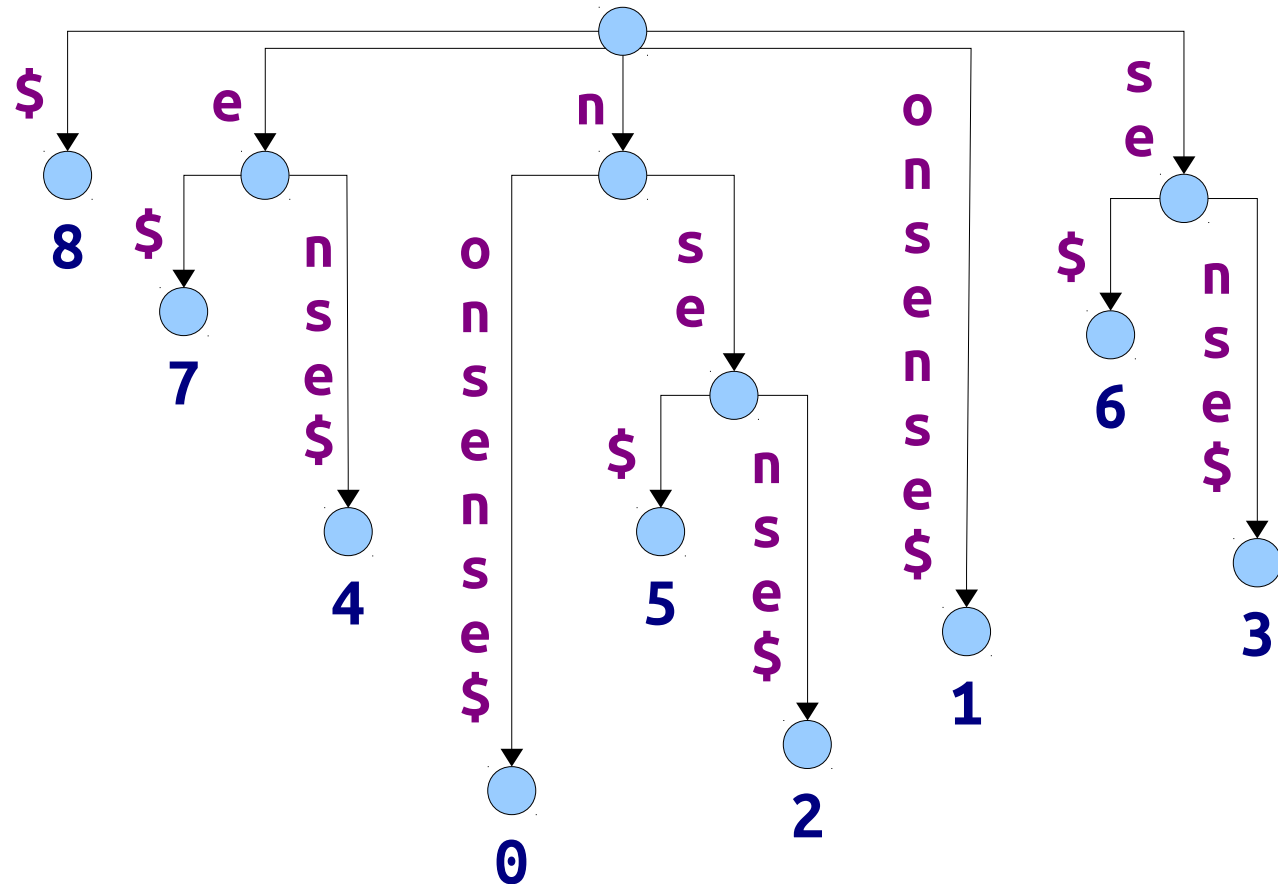


**Observation 2:** Every suffix of  $T\$$  beginning with some pattern  $P$  appears in the subtree found by searching for  $P$ .

nonsense\$  
012345678

# String Matching

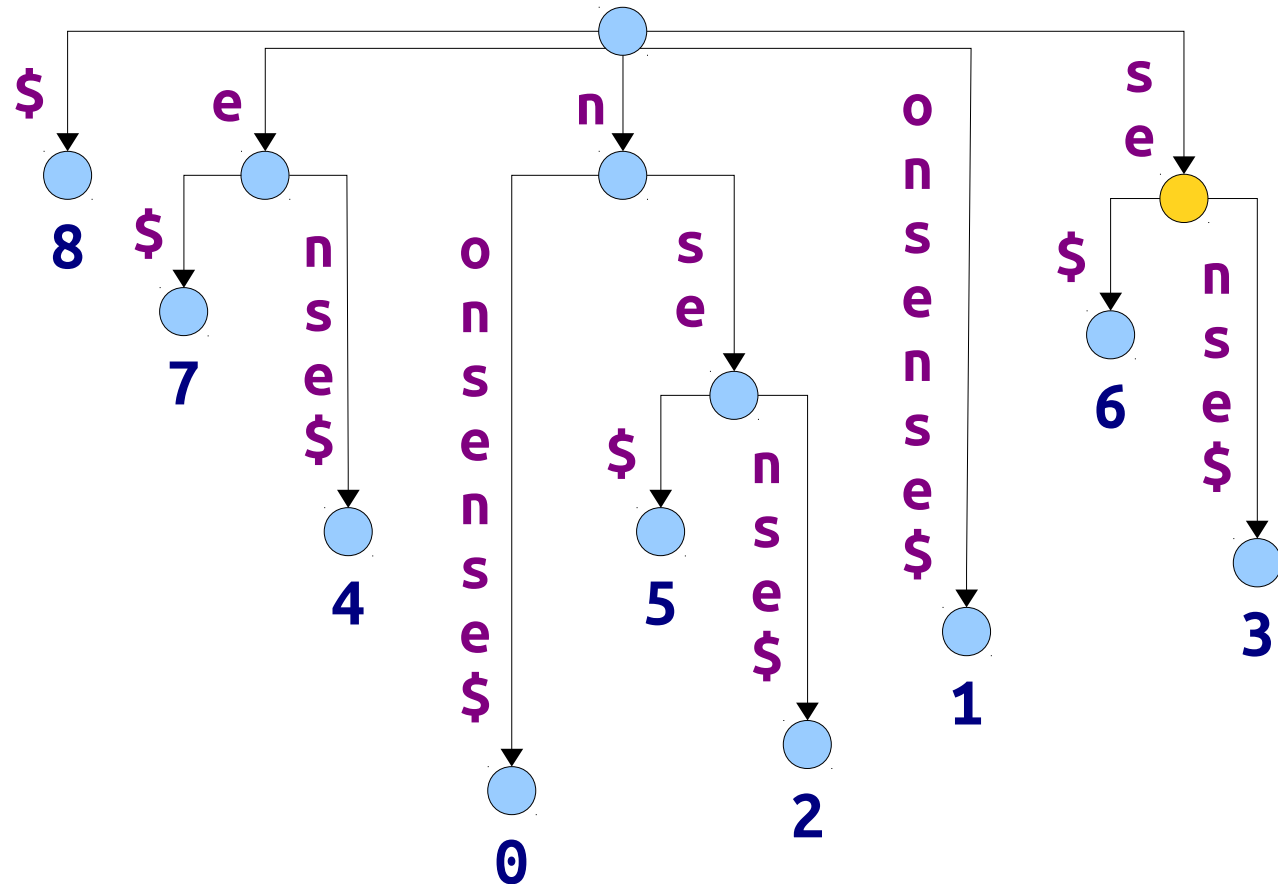
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

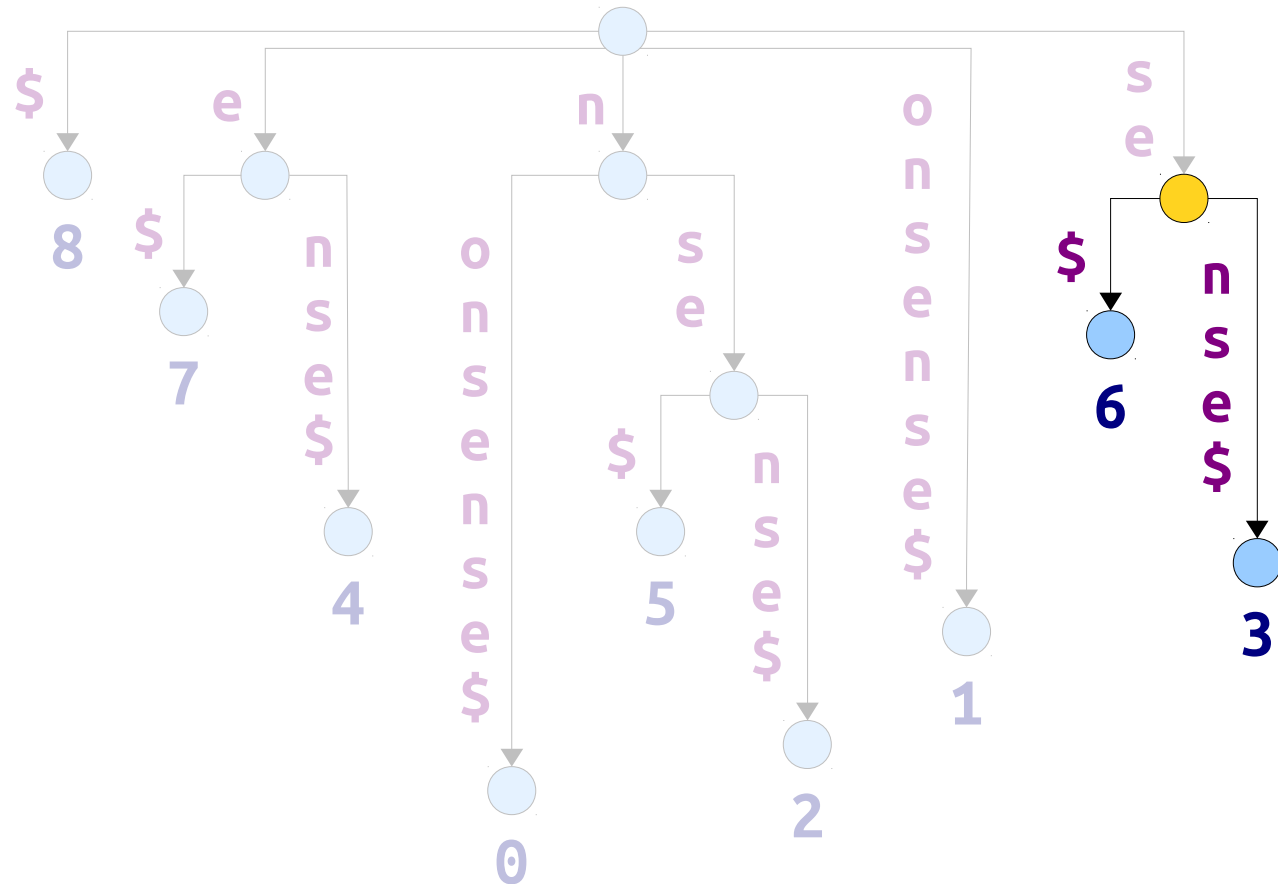
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

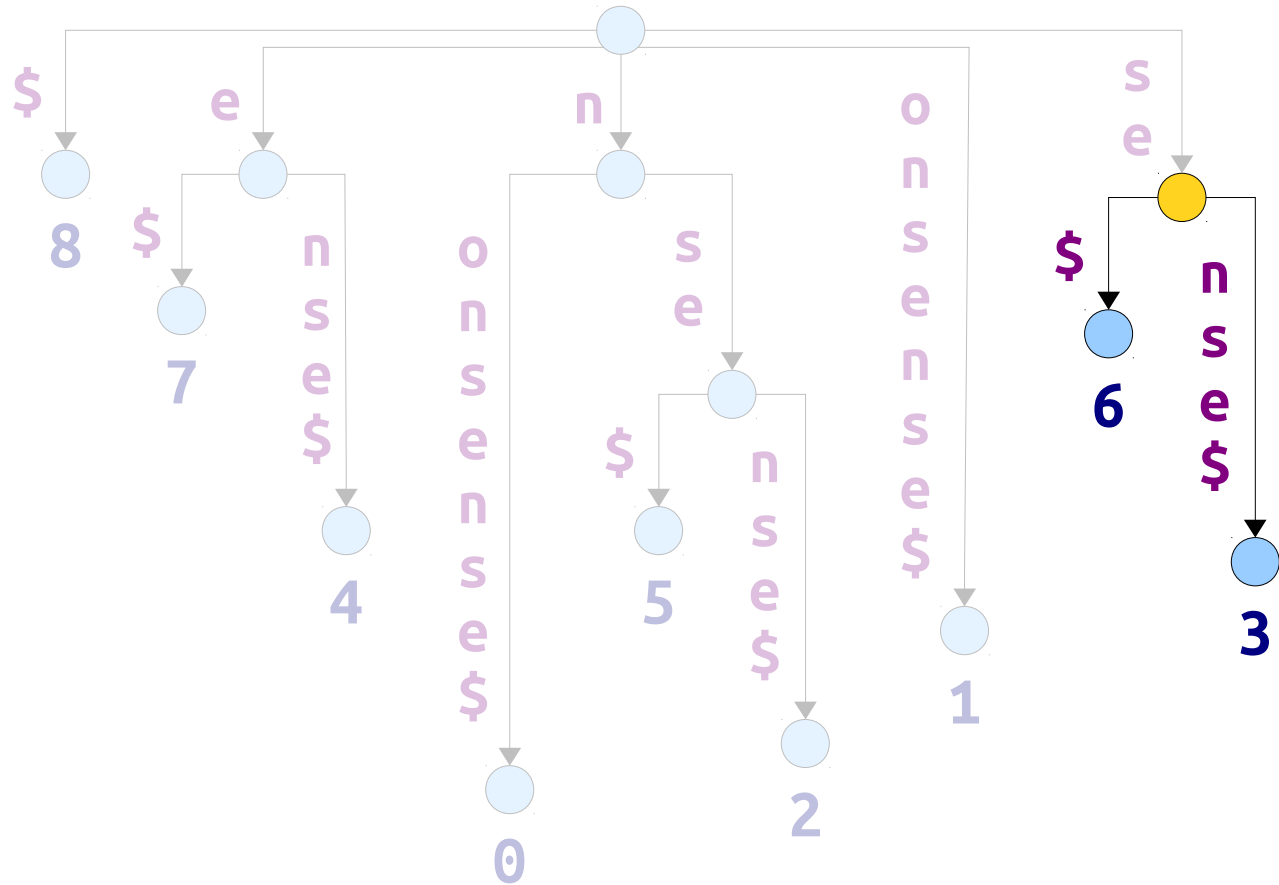
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

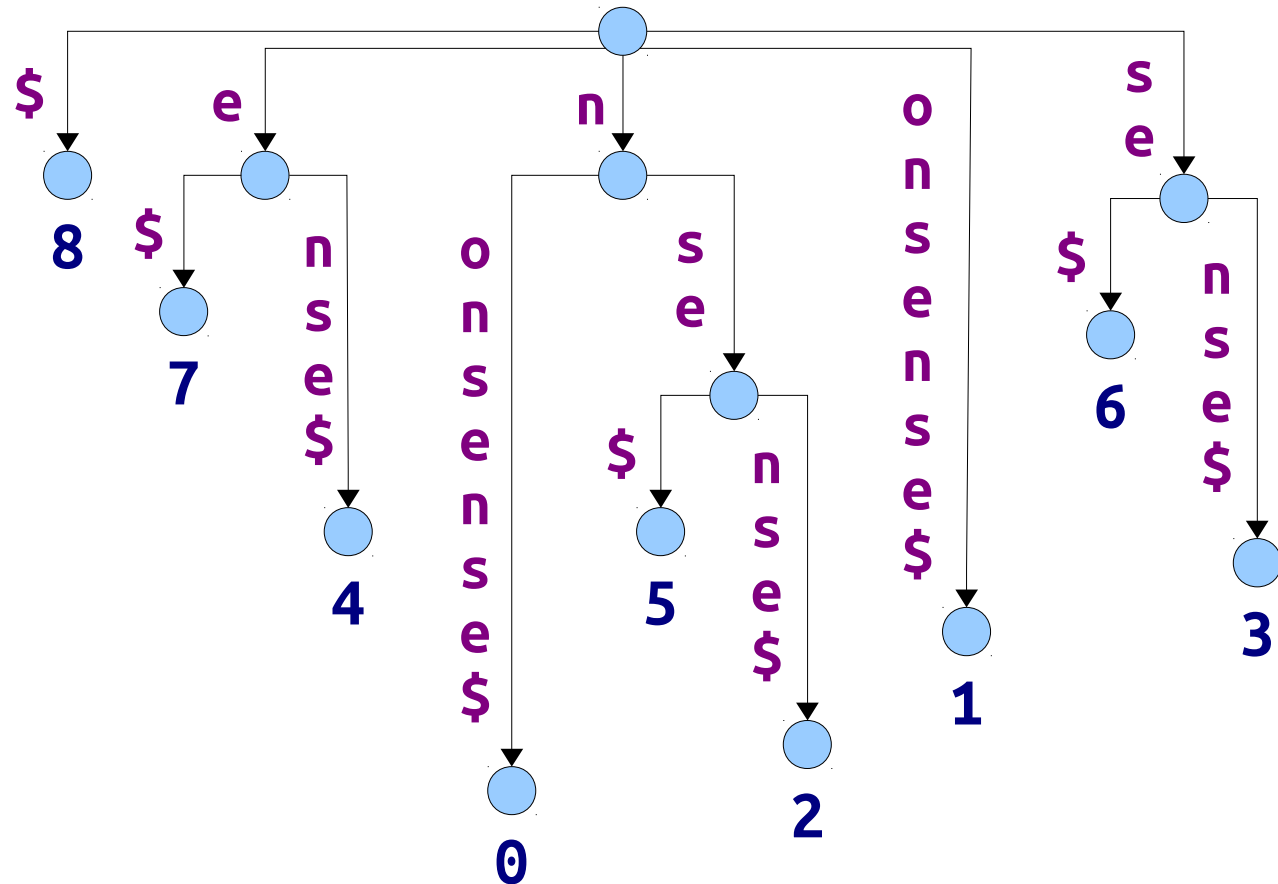
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

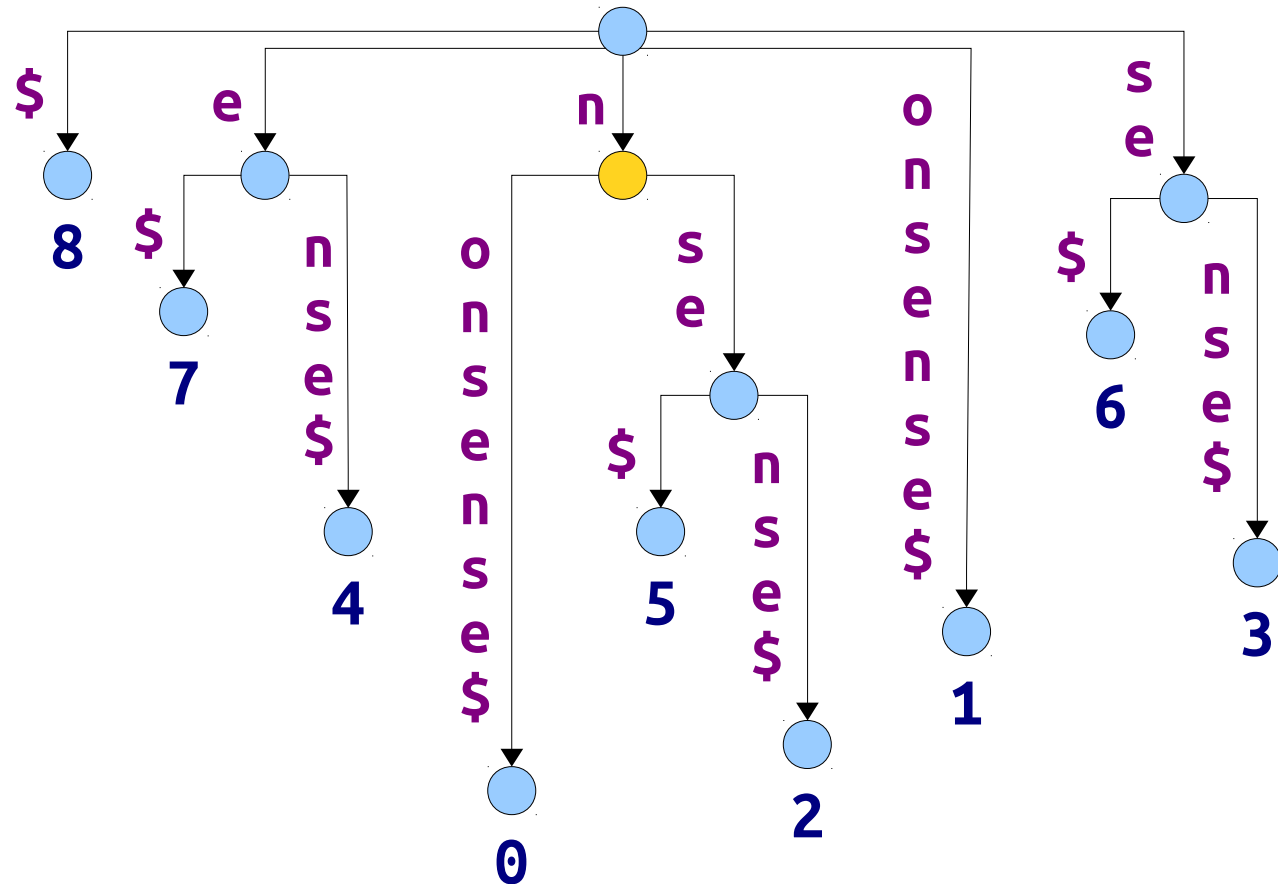
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

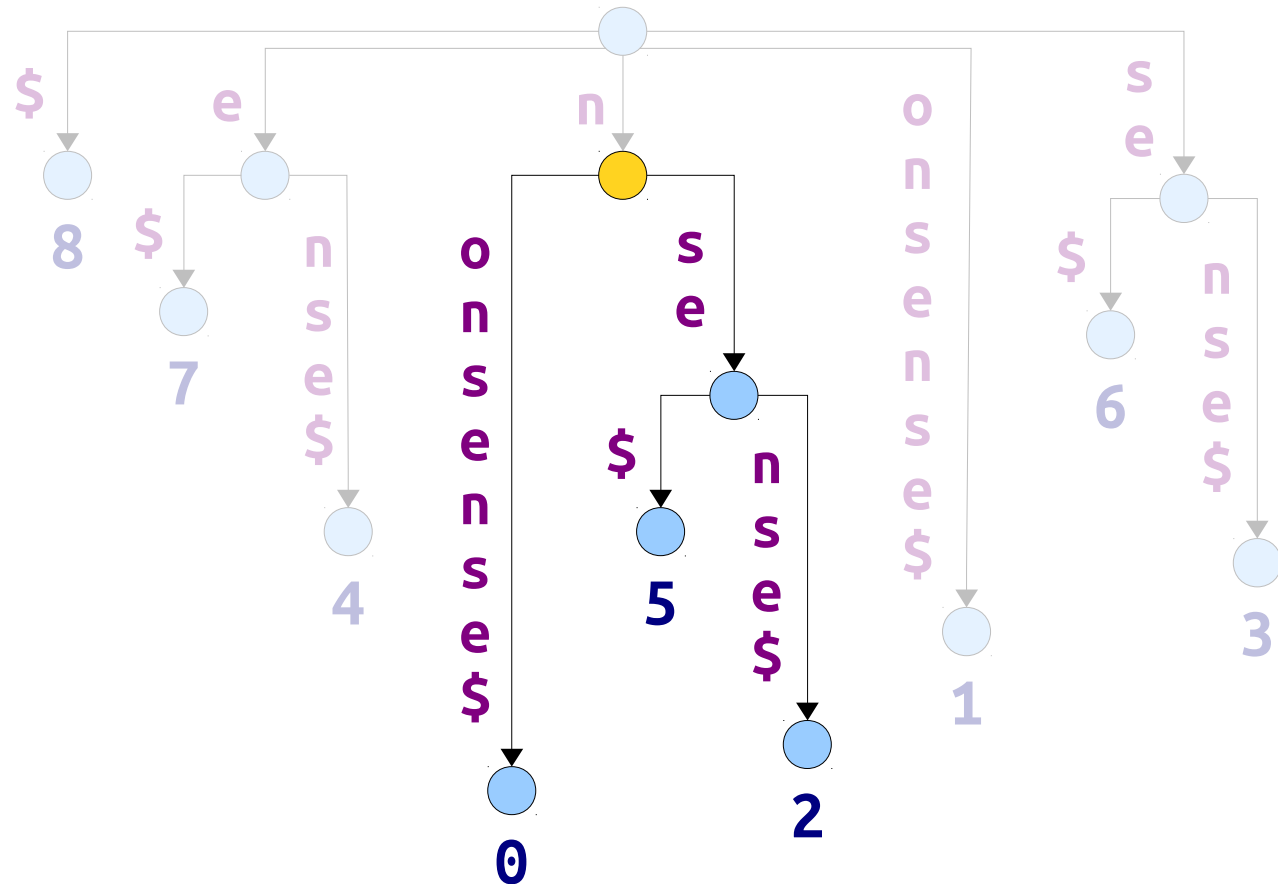
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.

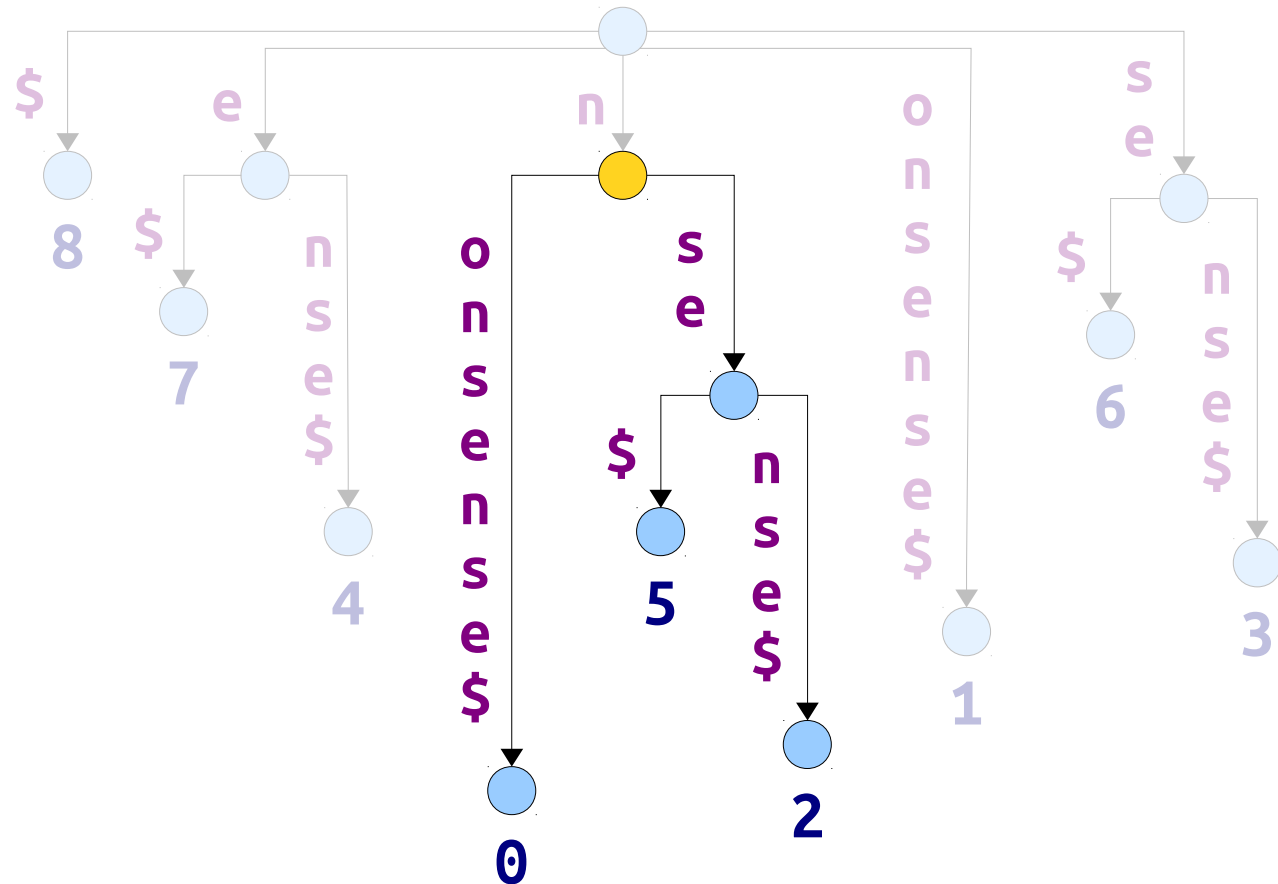


nonsense\$  
012345678



# String Matching

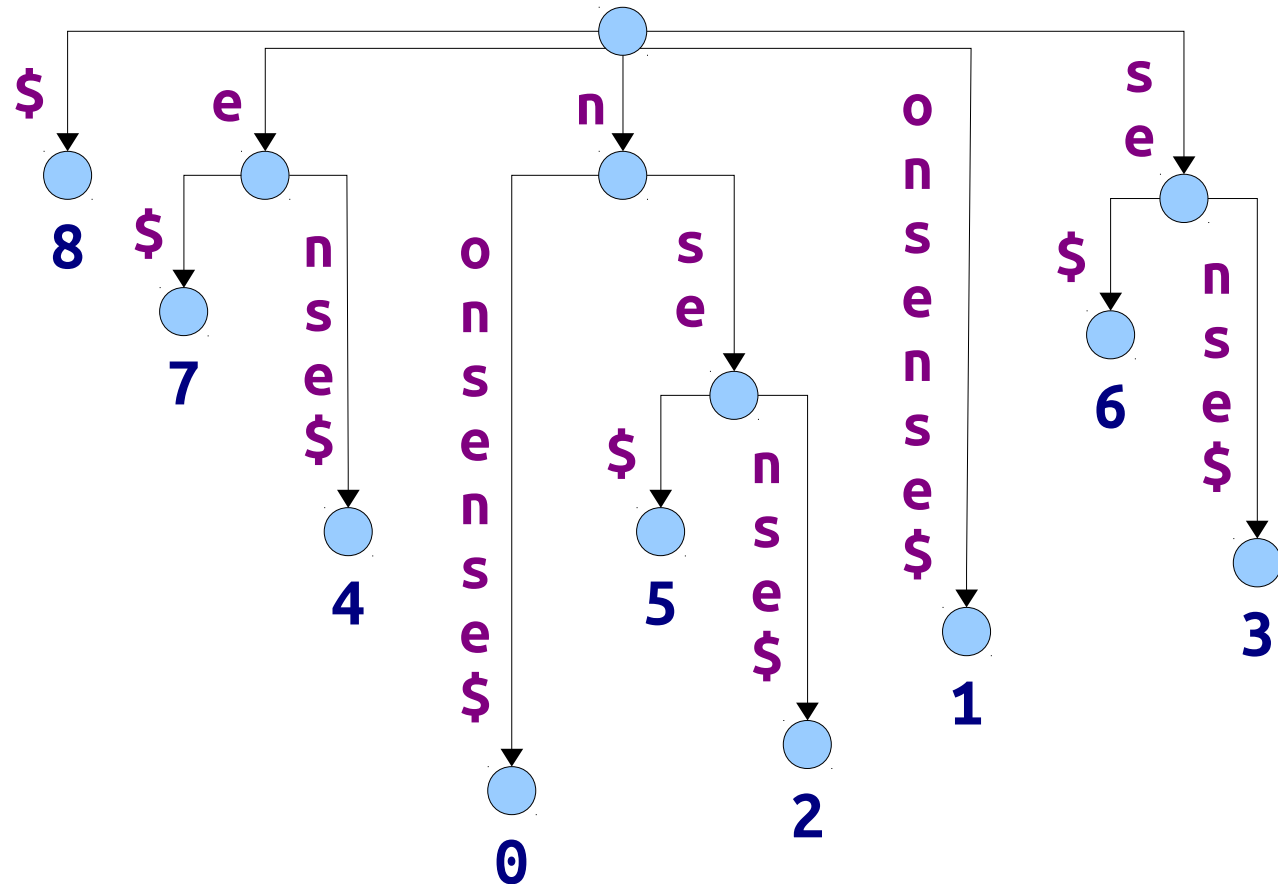
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

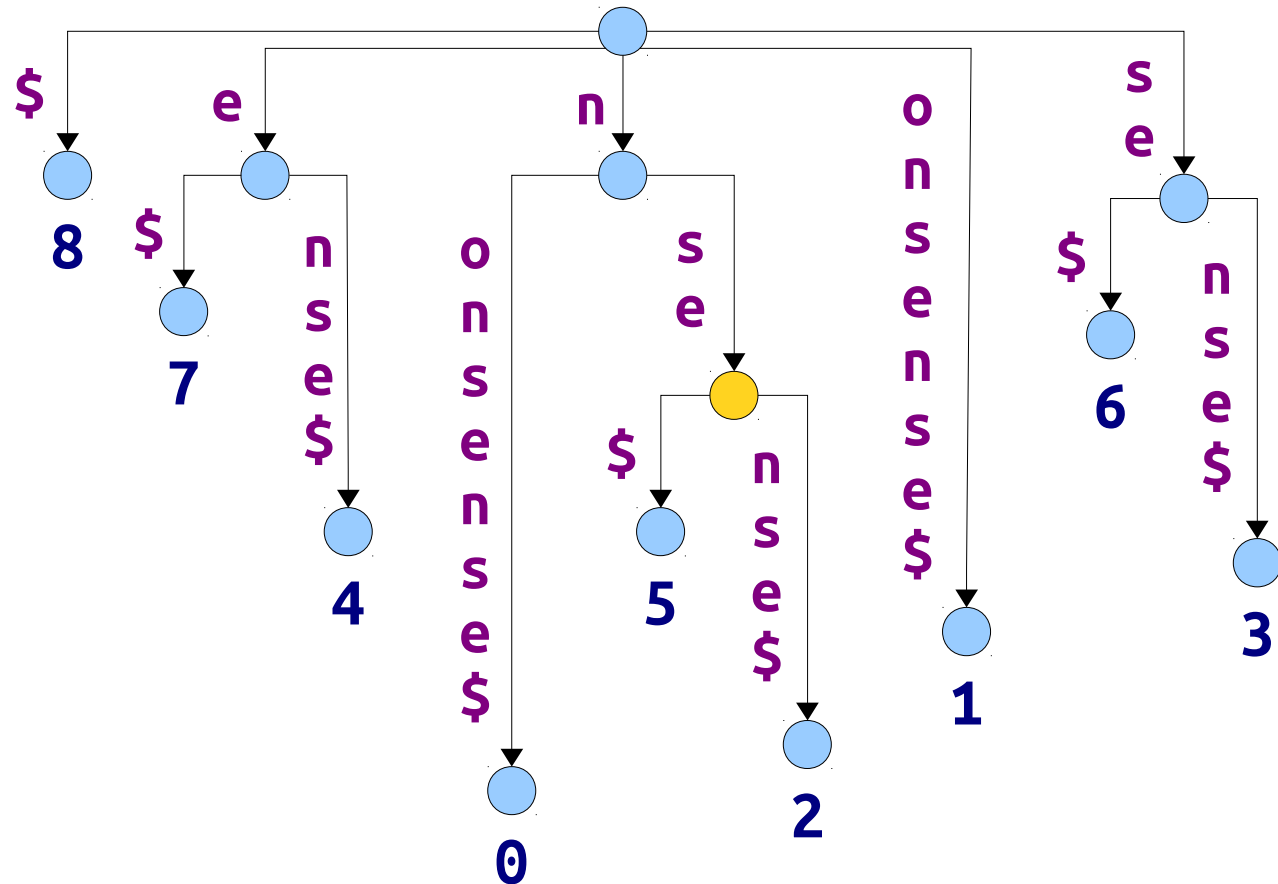
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

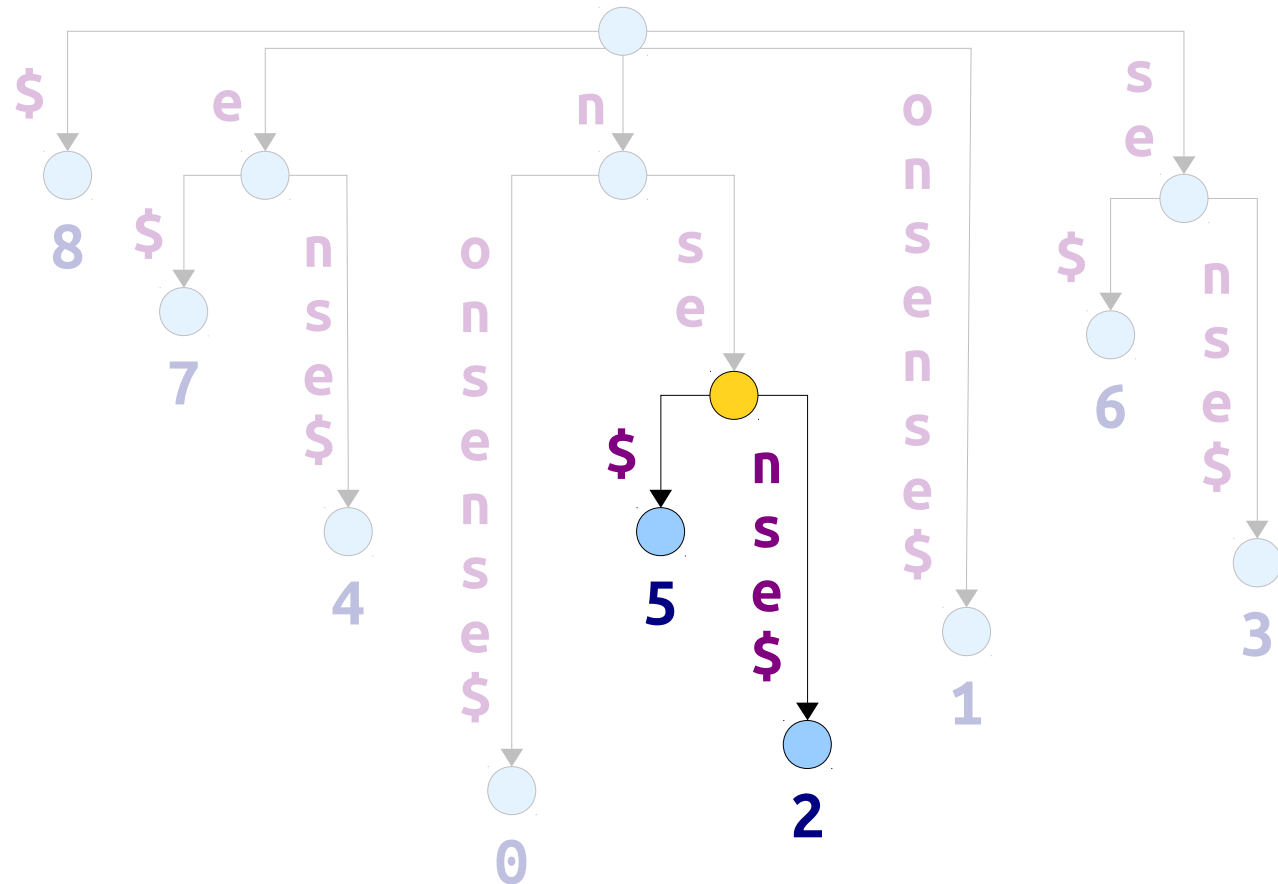
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

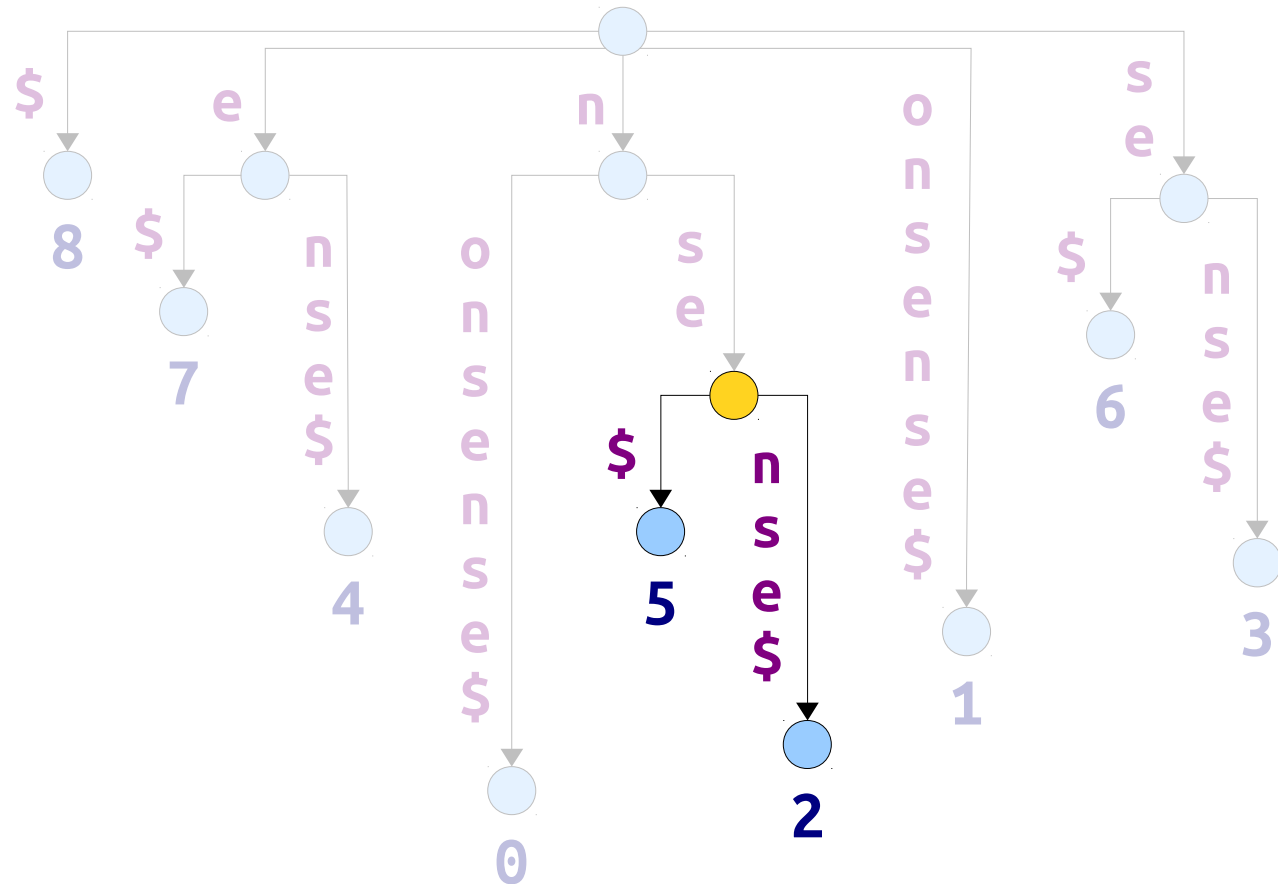
- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# String Matching

- **Claim:** After spending  $O(m)$  time preprocessing  $T$ , can find all matches of a string  $P$  in time  $O(n + z)$ , where  $z$  is the number of matches.



nonsense\$  
012345678

# Finding All Matches

- To find all matches of string  $P$ , start by searching the tree for  $P$ .
- If the search falls off the tree, report no matches.
- Otherwise, let  $v$  be the node at which the search stops, or the endpoint of the edge where it stops if it ends in the middle of an edge.
- Do a DFS and report the numbers of all the leaves found in this subtree. The indices reported this way give back all positions at which  $P$  occurs.

# Finding All Matches

To find all matches of string  $P$ , start by searching the tree for  $P$ .

If the search falls off the tree, report no matches.

Otherwise, let  $v$  be the node at which the search stops, or the endpoint of the edge where it stops if it ends in the middle of an edge.

- Do a DFS and report the numbers of all the leaves found in this subtree. The indices reported this way give back all positions at which  $P$  occurs.

# Finding All Matches

To find all matches of string  $P$ , start by searching the tree for  $P$ .

If the search falls off the tree, report no matches.

Otherwise, let  $v$  be the node at which the search stops, or the endpoint of the edge where it stops if it ends in the middle of an edge.

- Do a DFS and report the numbers of all the leaves found in this subtree. The indices reported this way give back all positions at which  $P$  occurs.

How fast is this step?



**Claim:** The DFS to find all leaves in the subtree corresponding to prefix  $P$  takes time  $O(z)$ , where  $z$  is the number of matches.

**Proof:** If the DFS reports  $z$  matches, it must have visited  $z$  different leaf nodes.

Since each internal node of a suffix tree has at least two children, the total number of internal nodes visited during the DFS is at most  $z - 1$ .

During the DFS, we don't need to actually match the characters on the edges. We just follow the edges, which takes time  $O(1)$ .

Therefore, the DFS visits at most  $O(z)$  nodes and edges and spends  $O(1)$  time per node or edge, so the total runtime is  $O(z)$ . ■

# Reverse Aho-Corasick

- Given patterns  $P_1, \dots, P_k$  of total length  $n$ , suffix trees can find all matches of those patterns in time  $O(m + n + z)$ .
  - Search for all matches of each  $P_i$ ; total time across all searches is  $O(n + z)$ .
- Acts as a “reverse” Aho-Corasick:
  - Aho-Corasick string matching runs in time  $\langle O(n), O(m+z) \rangle$
  - Suffix tree string matching runs in time  $\langle O(m), O(n+z) \rangle$

***Another Application:***  
Longest Repeated Substring

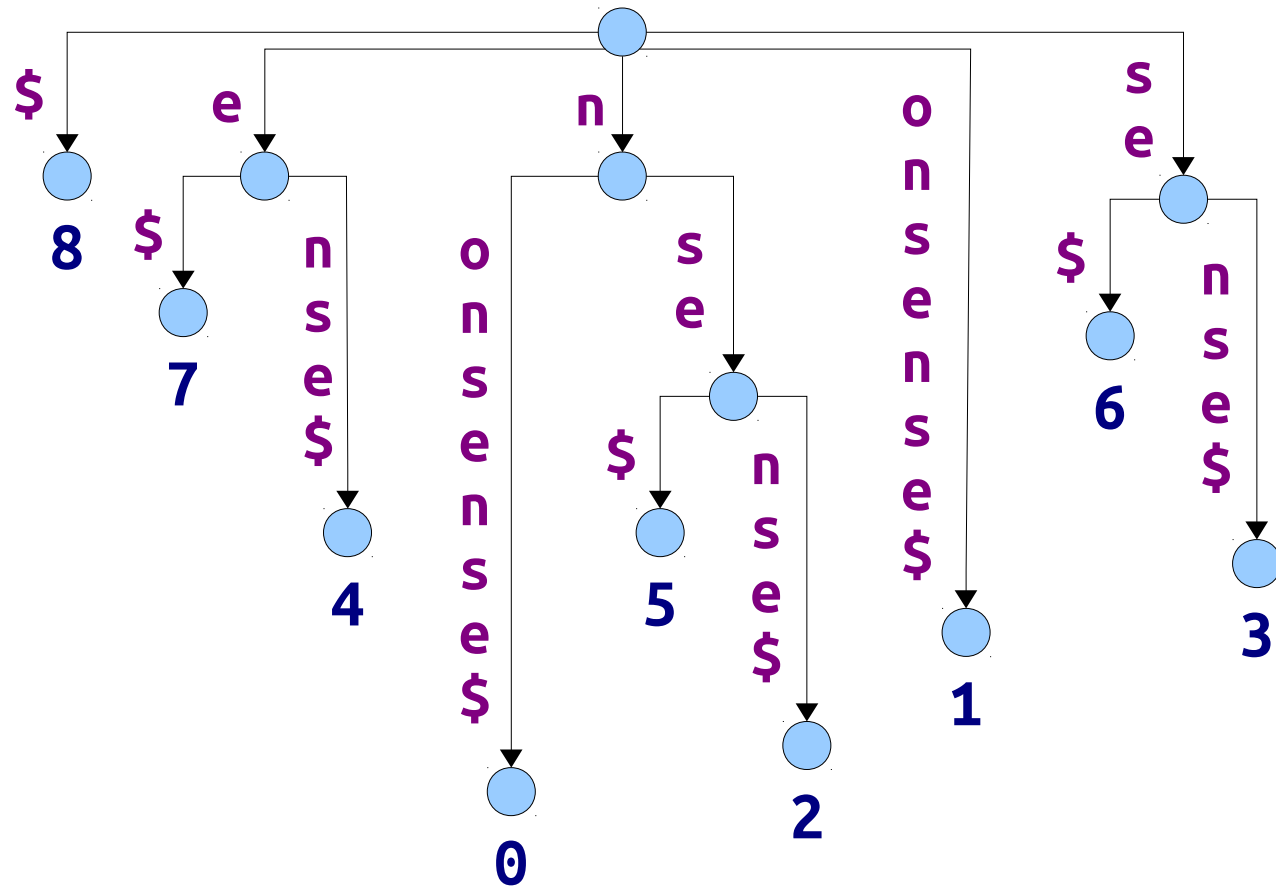
# Longest Repeated Substring

- Consider the following problem:

Given a string  $T$ , find the longest substring  $w$  of  $T$  that appears in at least two different positions.

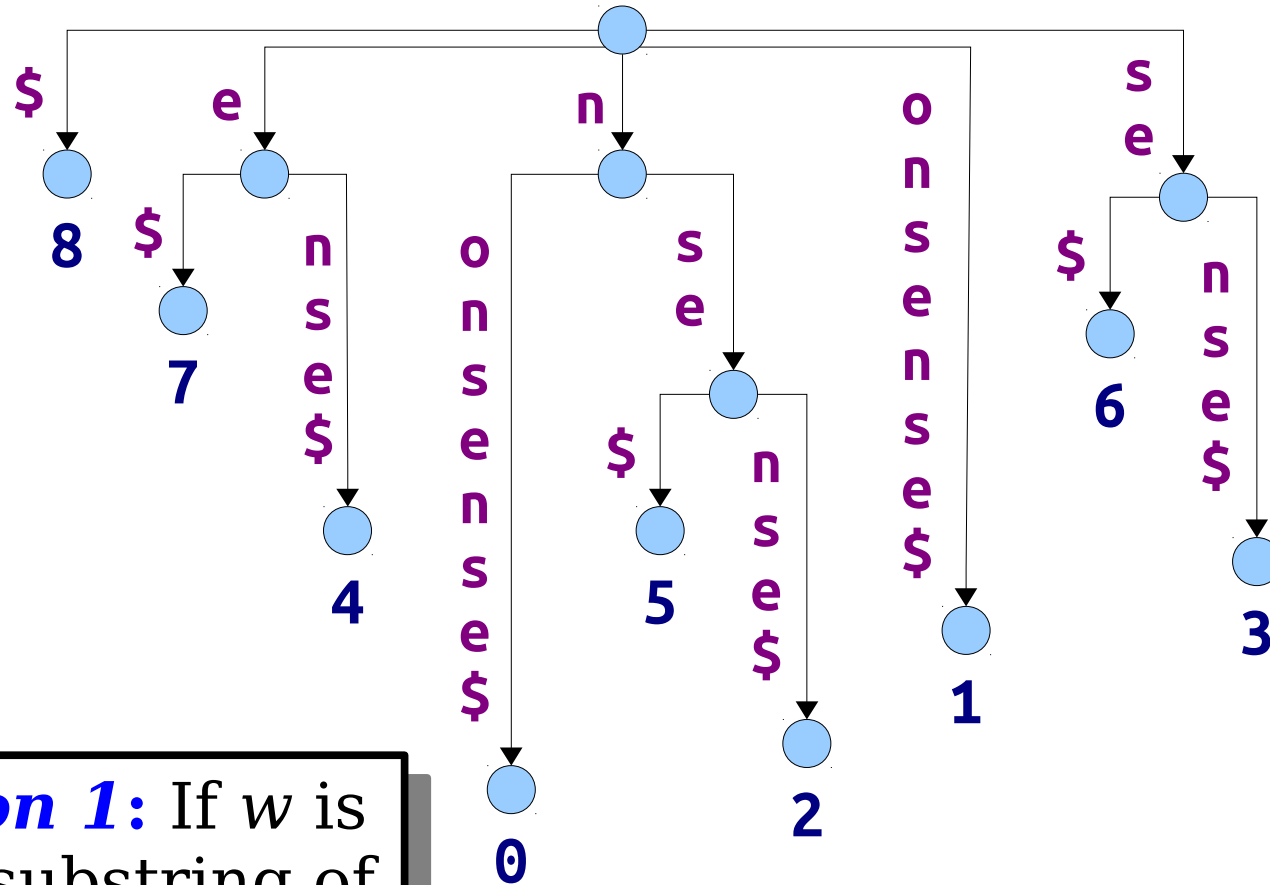
- Applications to computational biology: more than half of the human genome is formed from repeated DNA sequences!

# Longest Repeated Substring



nonsense\$  
012345678

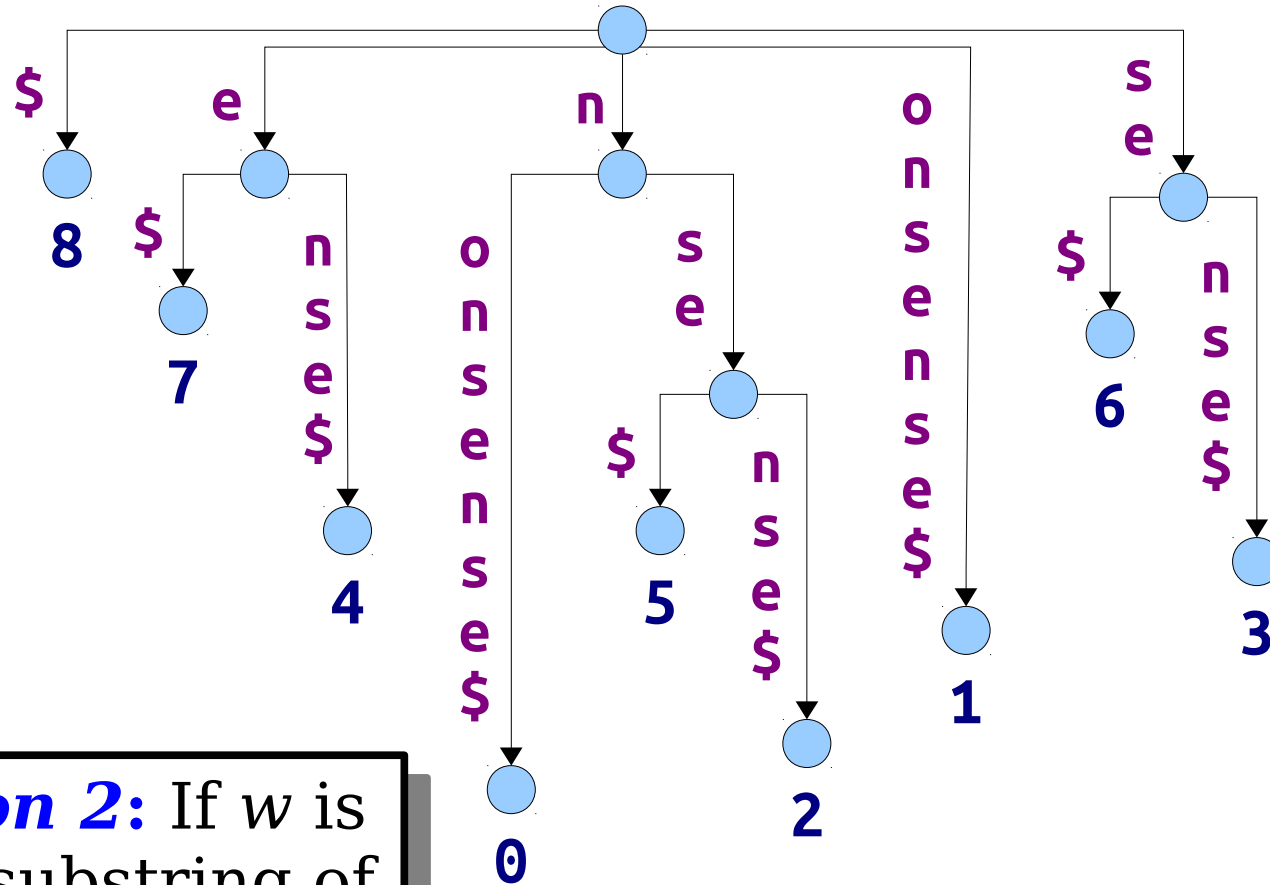
# Longest Repeated Substring



**Observation 1:** If  $w$  is a repeated substring of  $T$ , it must be a prefix of at least two different suffixes.

nonsense\$  
012345678

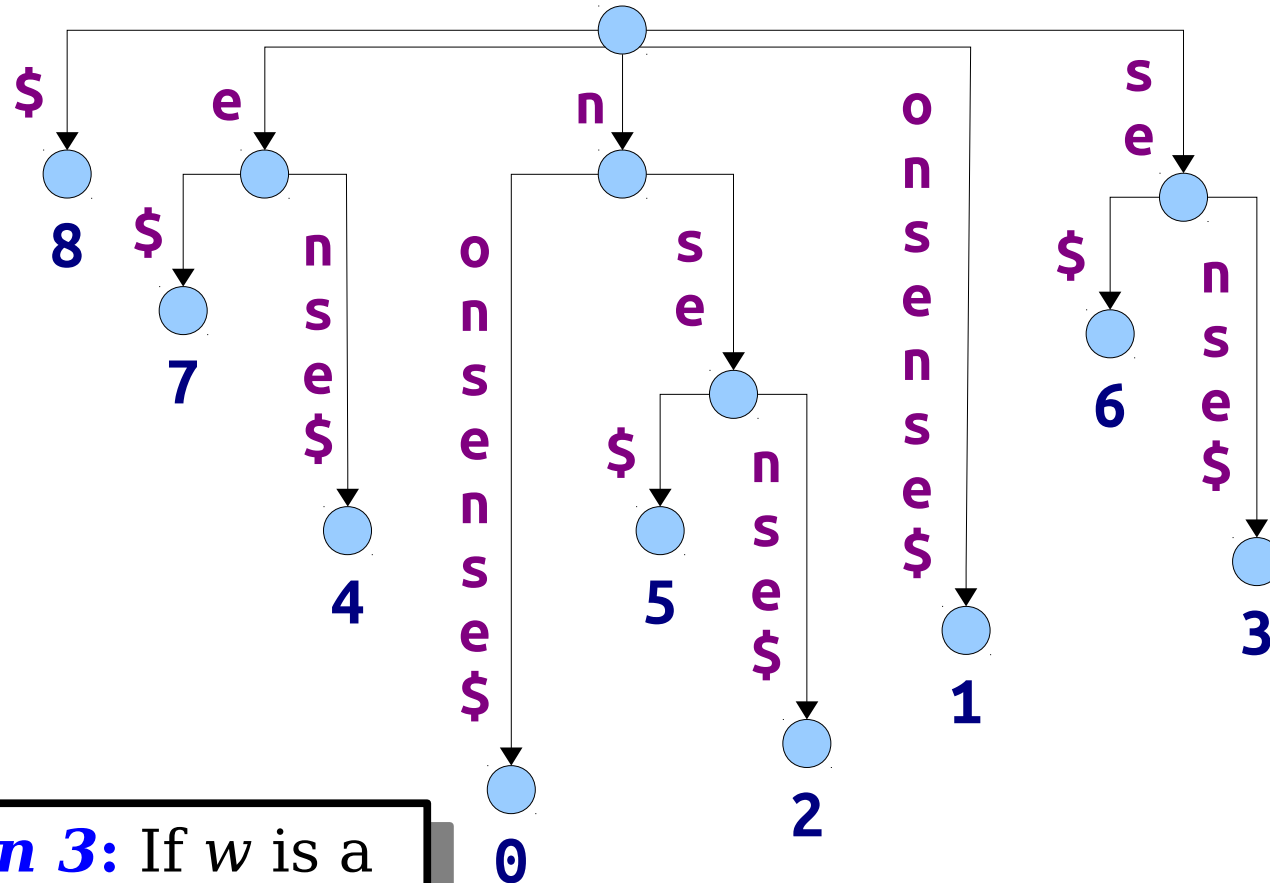
# Longest Repeated Substring



**Observation 2:** If  $w$  is a repeated substring of  $T$ , it must correspond to a prefix of a path to an internal node.

nonsense\$  
012345678

# Longest Repeated Substring

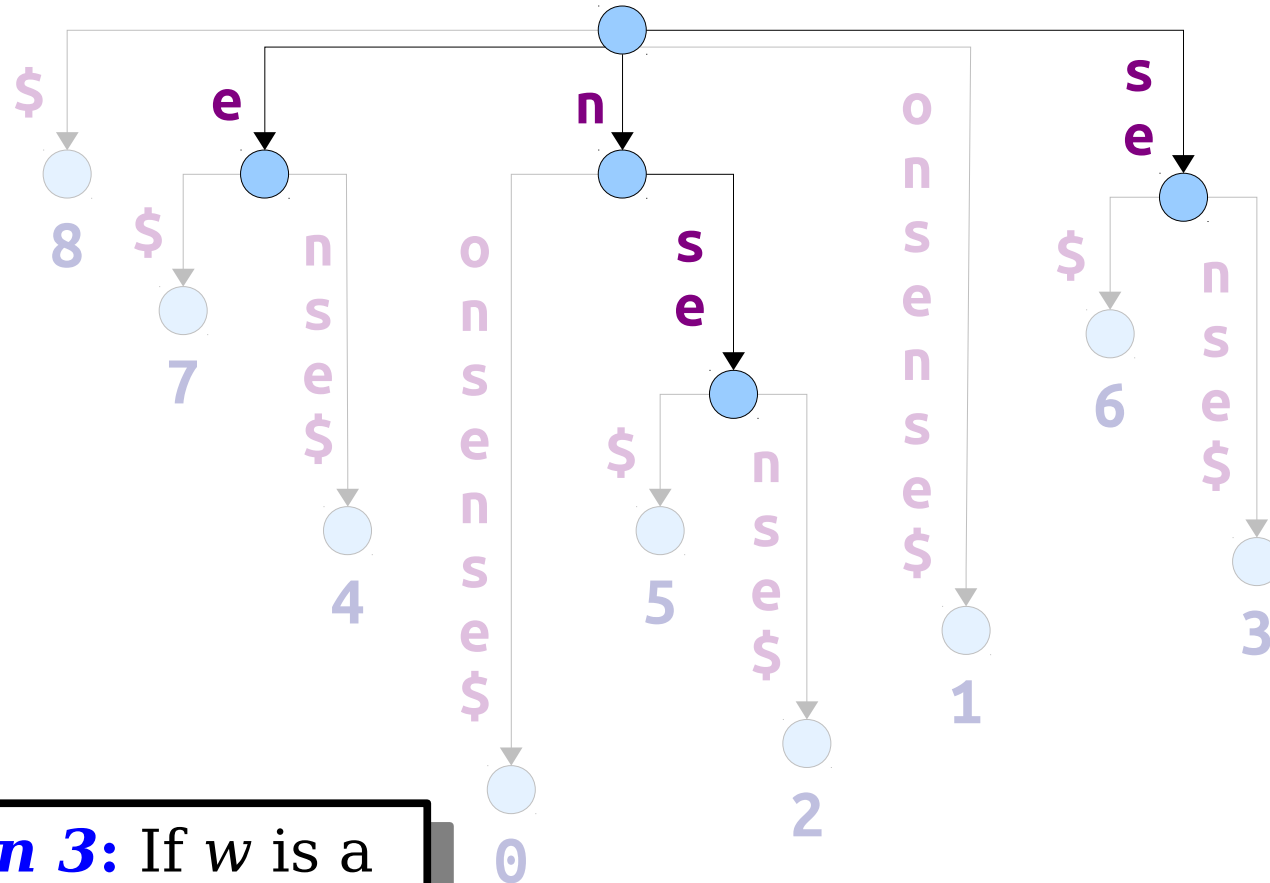


**Observation 3:** If  $w$  is a longest repeated substring, it corresponds to a full path to an internal node.

nonsense\$  
012345678



# Longest Repeated Substring



**Observation 3:** If  $w$  is a longest repeated substring, it corresponds to a full path to an internal node.

nonsense\$  
012345678

# Longest Repeated Substring

- For each node  $v$  in a suffix tree, let  $s(v)$  be the string that it corresponds to.
- The ***string depth*** of a node  $v$  is defined as  $|s(v)|$ , the length of the string  $v$  corresponds to.
- The longest repeated substring in  $T$  can be found by finding the internal node in  $T$  with the maximum string depth.

# Longest Repeated Substring

- Here's an  $O(m)$ -time algorithm for solving the longest repeated substring problem:
  - Build the suffix tree for  $T$  in time  $O(m)$ .
  - Run a DFS over  $T$ , tracking the string depth as you go, to find the internal node of maximum string depth.
  - Recover the string  $T$  corresponds to.
- ***Good exercise:*** How might you find the longest substring of  $T$  that repeats at least  $k$  times?

## ***Challenge Problem:***

Solve this problem in linear time without using suffix trees (or suffix arrays).

Time-Out for Announcements!

# Problem Set One

- Problem Set One was due today at 3:00PM.
  - Want to use your late days? Submit by Saturday at 3:00PM.
- Solutions will go out on Tuesday.
- Problem Set Two goes out on Tuesday – have a good weekend!

# Talk Today

- Jon Kleinberg (who authored *Algorithm Design* along with Eva Tardos) is giving a talk today at 4:15PM in the Mackenzie Boardroom.
- Focus is on algorithms for solving problems with agents who don't plan rationally.
- Sounds really fun – hopefully we'll finish with a little buffer time. ☺

Back to CS166!



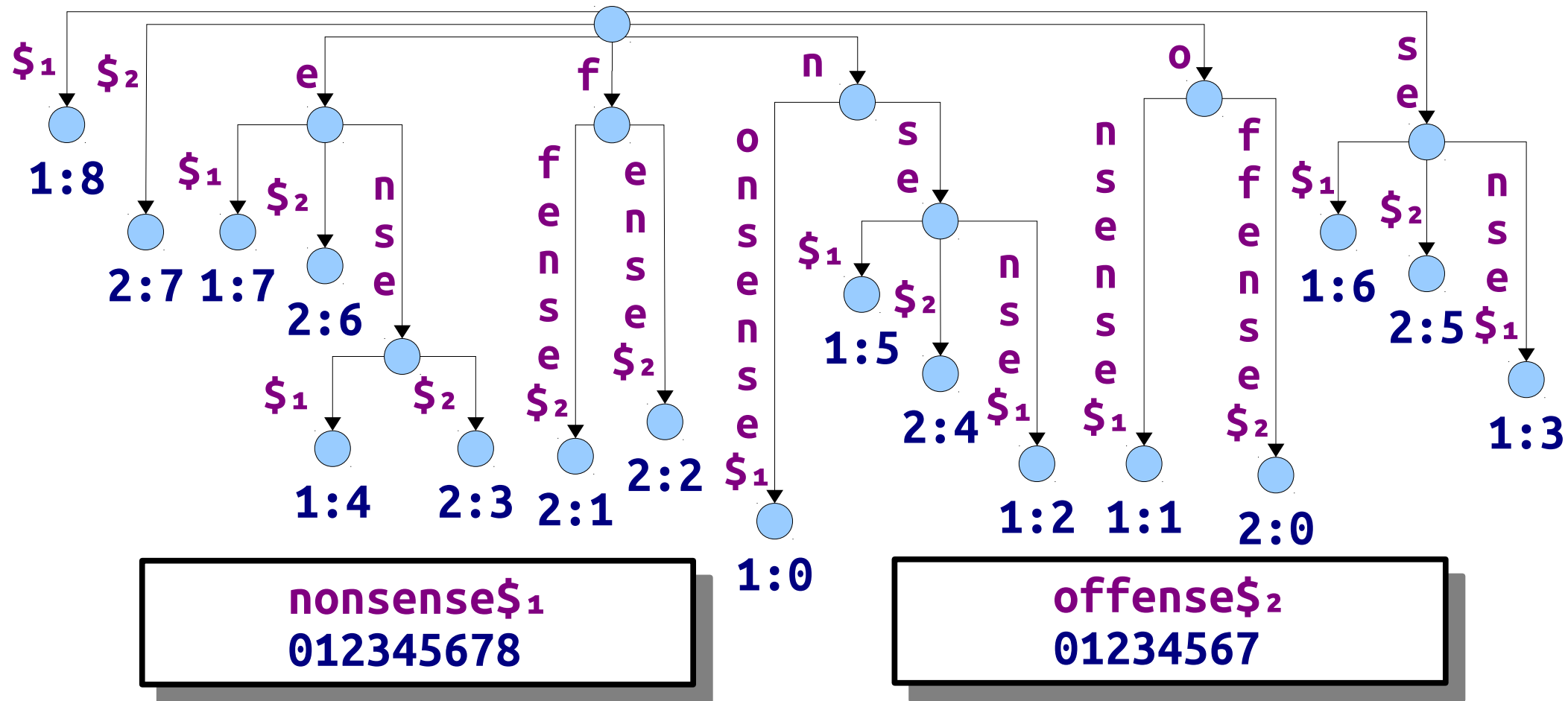
# Generalized Suffix Trees

# Suffix Trees for Multiple Strings

- Suffix trees store information about a single string and exports a huge amount of structural information about that string.
- However, many applications require information about the structure of multiple different strings.

# Generalized Suffix Trees

- A **generalized suffix tree** for  $T_1, \dots, T_k$  is a Patricia trie of all suffixes of  $T_1\$1, \dots, T_k\$k$ . Each  $T_i$  has a unique end marker.
- Leaves are tagged with  $i:j$ , meaning “ $j$ th suffix of string  $T_i$ ”



# Generalized Suffix Trees

- **Claim:** A generalized suffix tree for strings  $T_1, \dots, T_k$  of total length  $m$  can be constructed in time  $\Theta(m)$ .
- Use a two-phase algorithm:
  - Construct a suffix tree for the single string  $T_1\$1T_2\$2 \dots T_k\$k$  in time  $\Theta(m)$ .
    - This will end up with some invalid suffixes.
  - Do a DFS over the suffix tree and prune the invalid suffixes.
    - Runs in time  $O(m)$  if implemented intelligently.

# Applications of Generalized Suffix Trees

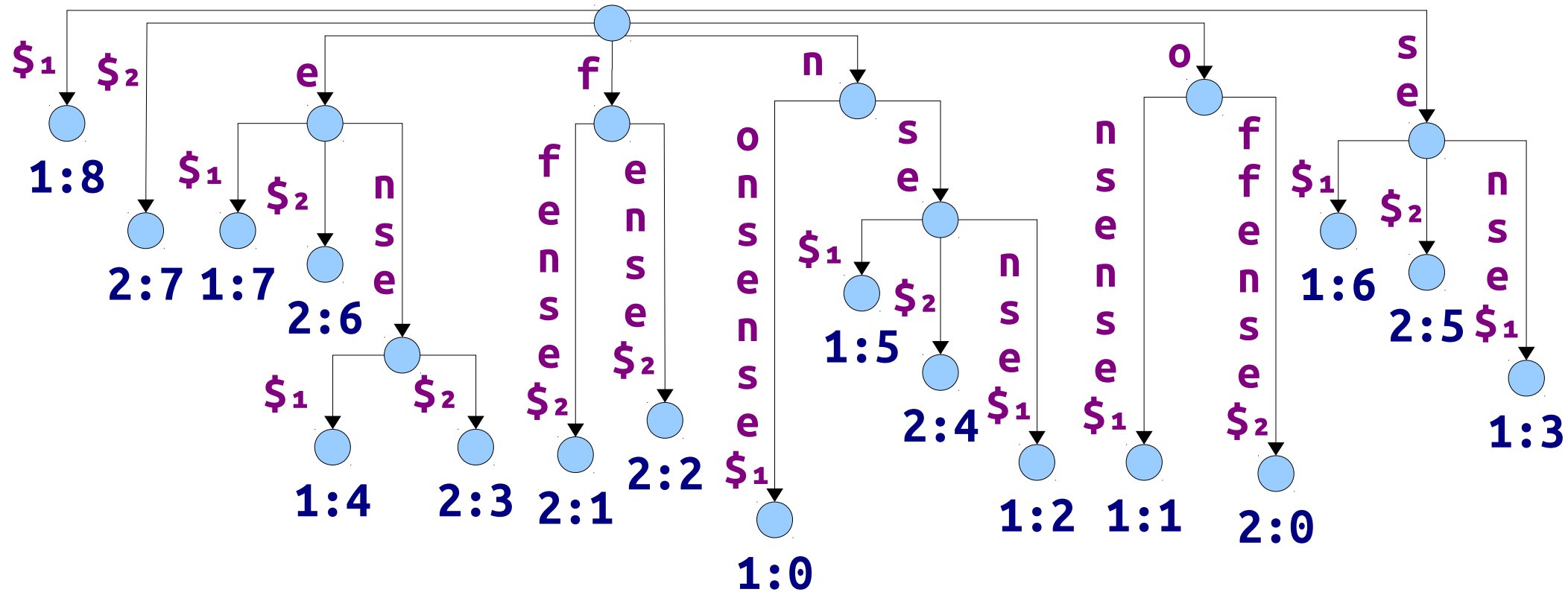
# Longest Common Substring

- Consider the following problem:

Given two strings  $T_1$  and  $T_2$ , find the longest string  $w$  that is a substring of both  $T_1$  and  $T_2$ .

- Can solve in time  $O(|T_1| \cdot |T_2|)$  using dynamic programming.
- Can we do better?

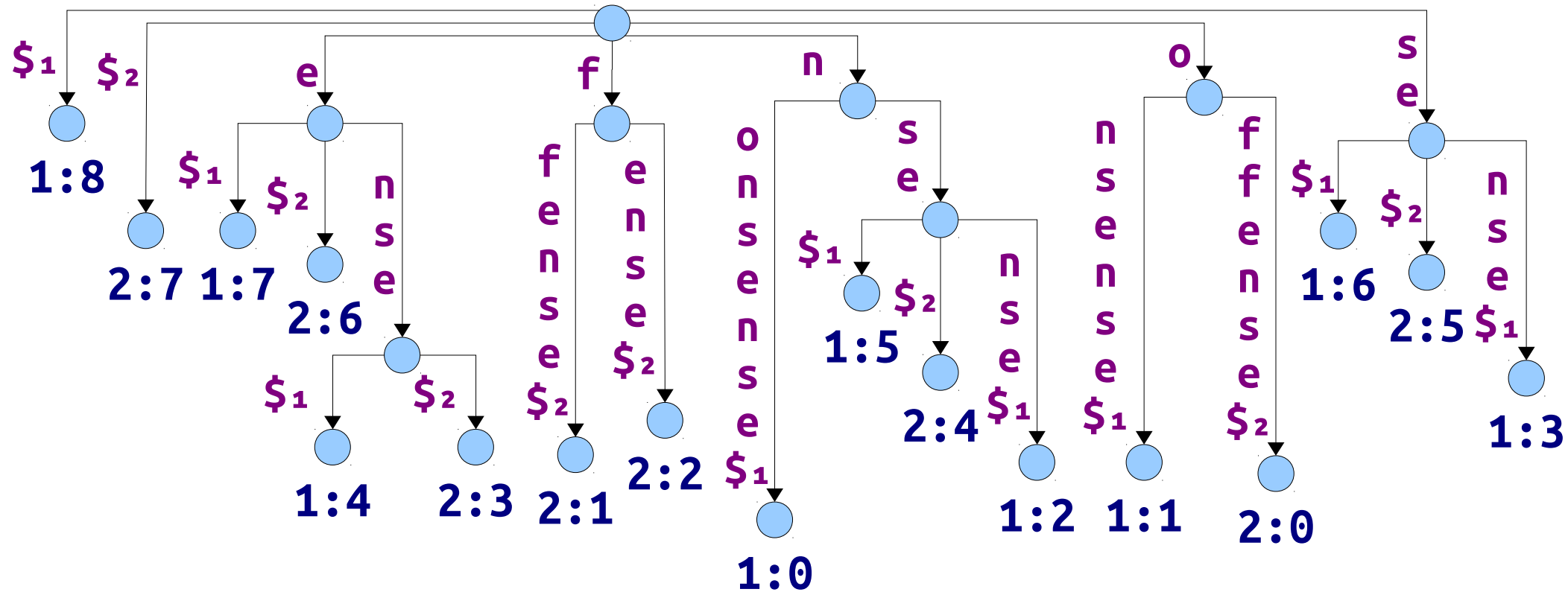
# Longest Common Substring



nonsense\$1  
012345678

offense\$2  
01234567

# Longest Common Substring



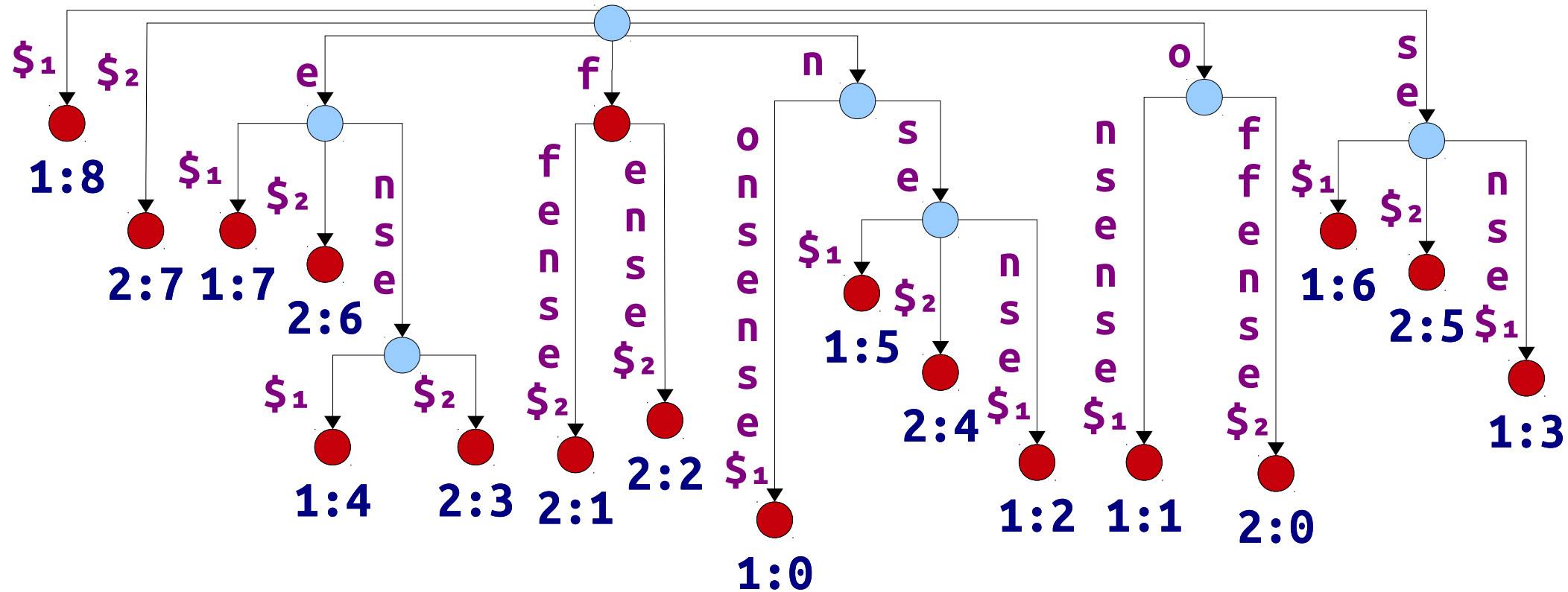
nonsense $\$1$   
012345678

**Observation:** Any common substring of  $T_1$  and  $T_2$  will be a prefix of a suffix of  $T_1$  and a prefix of a suffix of  $T_2$ .

se $\$2$   
567



# Longest Common Substring



nonsense\$1  
012345678

offense\$2  
01234567

# Longest Common Substring

- Build a generalized suffix tree for  $T_1$  and  $T_2$  in time  $O(m)$ .
- Annotate each internal node in the tree with whether that node has at least one leaf node from each of  $T_1$  and  $T_2$ .
  - Takes time  $O(m)$  using DFS.
- Run a DFS over the tree to find the marked node with the highest string depth.
  - Takes time  $O(m)$  using DFS
- Overall time:  **$O(m)$** .

# Longest Common Extensions

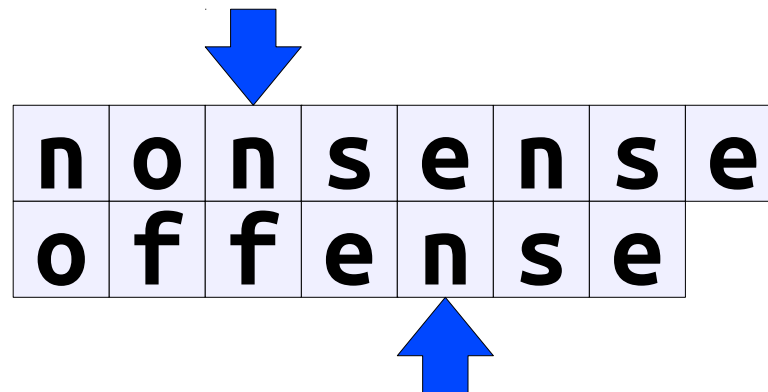
# Longest Common Extensions

- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.

|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| n | o | n | s | e | n | s | e |
| o | f | f | e | n | s | e |   |

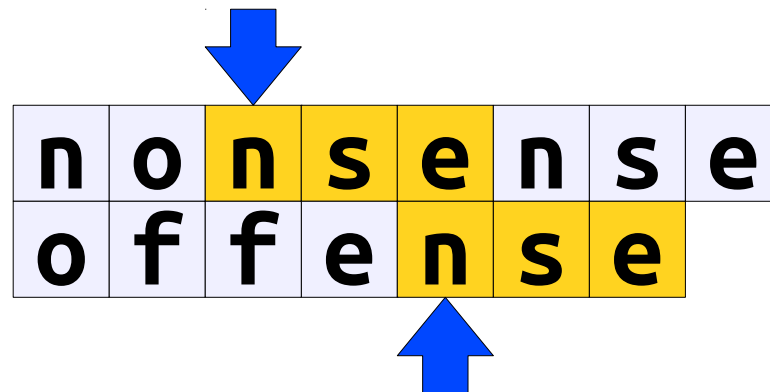
# Longest Common Extensions

- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.



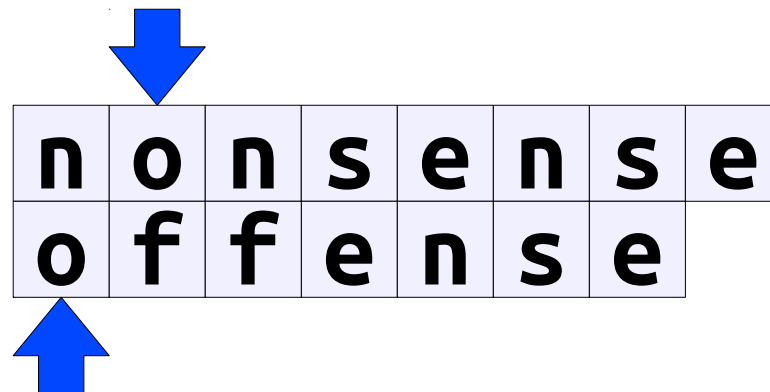
# Longest Common Extensions

- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.



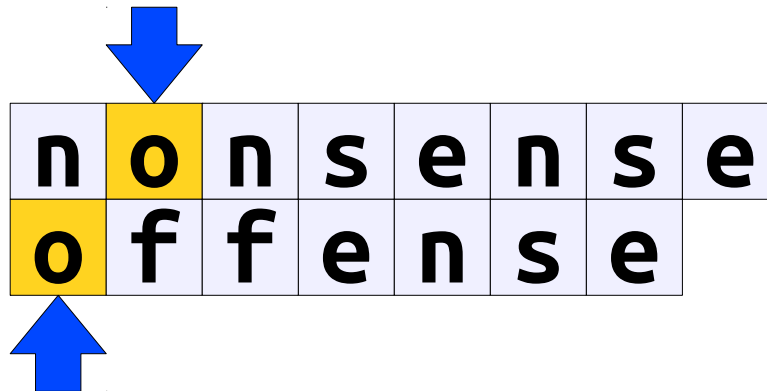
# Longest Common Extensions

- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.



# Longest Common Extensions

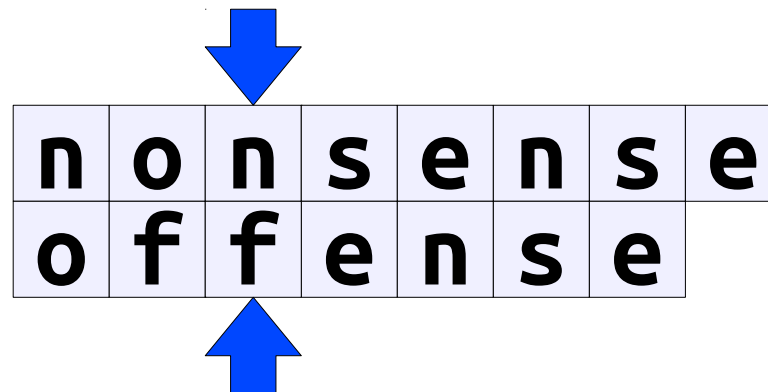
- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.





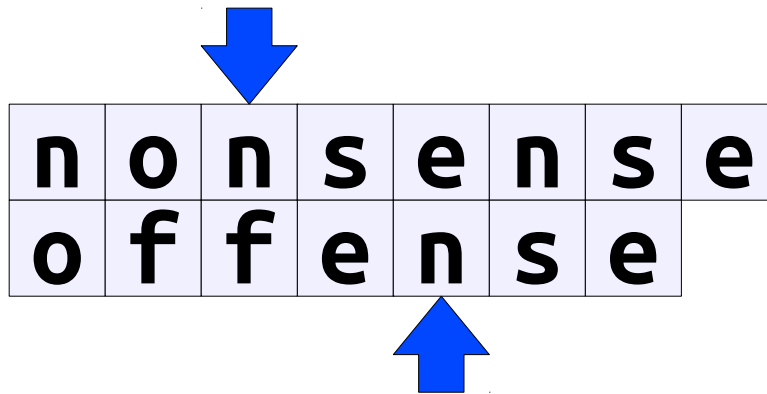
# Longest Common Extensions

- Given two strings  $T_1$  and  $T_2$  and start positions  $i$  and  $j$ , the **longest common extension** of  $T_1$  and  $T_2$ , starting at positions  $i$  and  $j$ , is the length of the longest string  $w$  that appears at position  $i$  in  $T_1$  and position  $j$  in  $T_2$ .
- We'll denote this value by  $\text{LCE}_{T_1, T_2}(i, j)$ .
- Typically,  $T_1$  and  $T_2$  are fixed and multiple  $(i, j)$  queries are specified.



# Longest Common Extensions

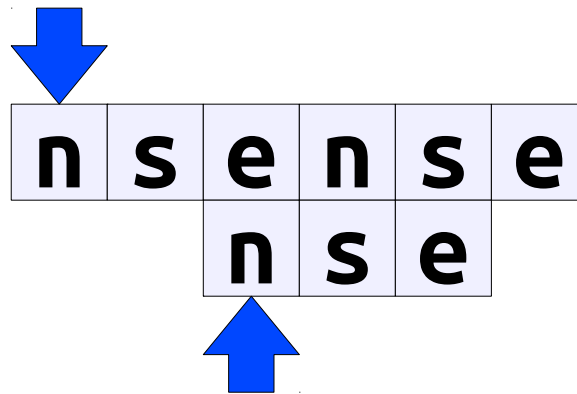
- **Observation:**  $\text{LCE}_{T_1, T_2}(i, j)$  is the length of the longest common prefix of the suffixes of  $T_1$  and  $T_2$  starting at positions  $i$  and  $j$ .



- The generalized suffix tree of  $T_1$  and  $T_2$  makes it easy to query for these suffixes and stores information about their common prefixes.

# Longest Common Extensions

- **Observation:**  $\text{LCE}_{T_1, T_2}(i, j)$  is the length of the longest common prefix of the suffixes of  $T_1$  and  $T_2$  starting at positions  $i$  and  $j$ .



- The generalized suffix tree of  $T_1$  and  $T_2$  makes it easy to query for these suffixes and stores information about their common prefixes.

# Longest Common Extensions

- **Observation:**  $\text{LCE}_{T_1, T_2}(i, j)$  is the length of the longest common prefix of the suffixes of  $T_1$  and  $T_2$  starting at positions  $i$  and  $j$ .

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| n | s | e | n | s | e |
| n | s | e |   |   |   |

- The generalized suffix tree of  $T_1$  and  $T_2$  makes it easy to query for these suffixes and stores information about their common prefixes.

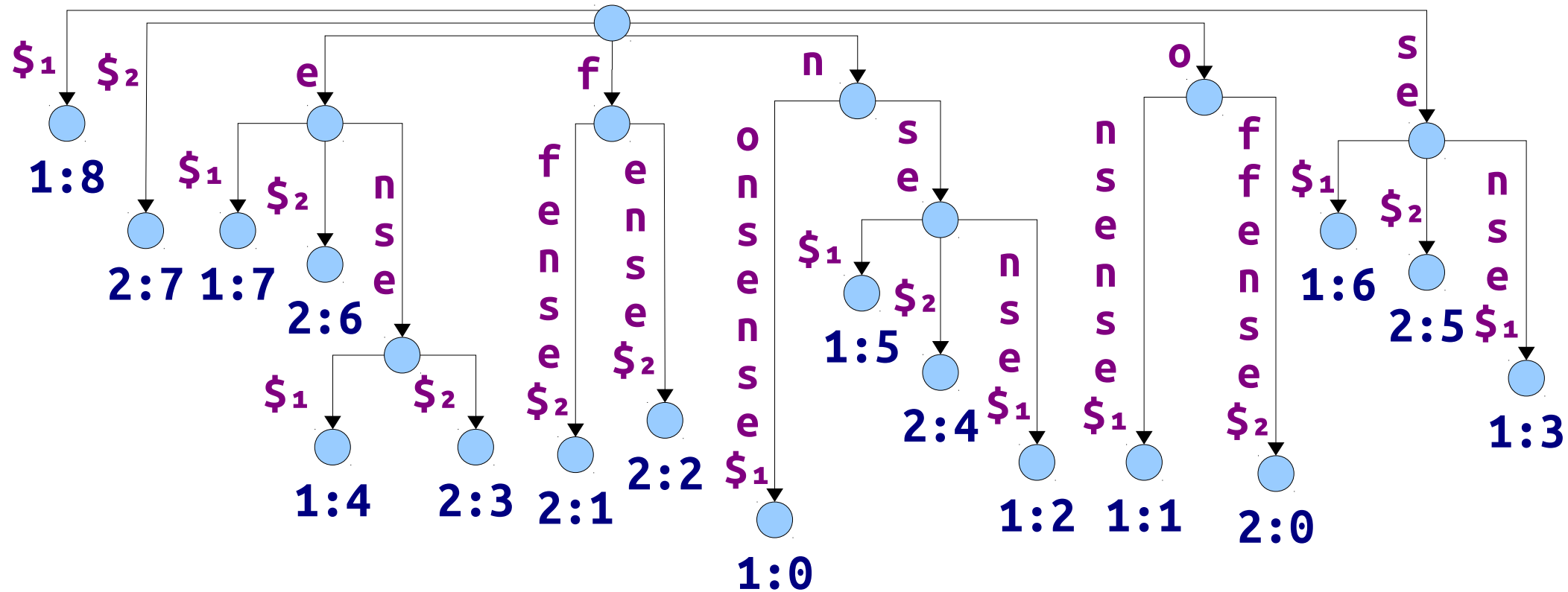
# Longest Common Extensions

- **Observation:**  $\text{LCE}_{T_1, T_2}(i, j)$  is the length of the longest common prefix of the suffixes of  $T_1$  and  $T_2$  starting at positions  $i$  and  $j$ .

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| n | s | e | n | s | e |
| n | s | e |   |   |   |

- The generalized suffix tree of  $T_1$  and  $T_2$  makes it easy to query for these suffixes and stores information about their common prefixes.

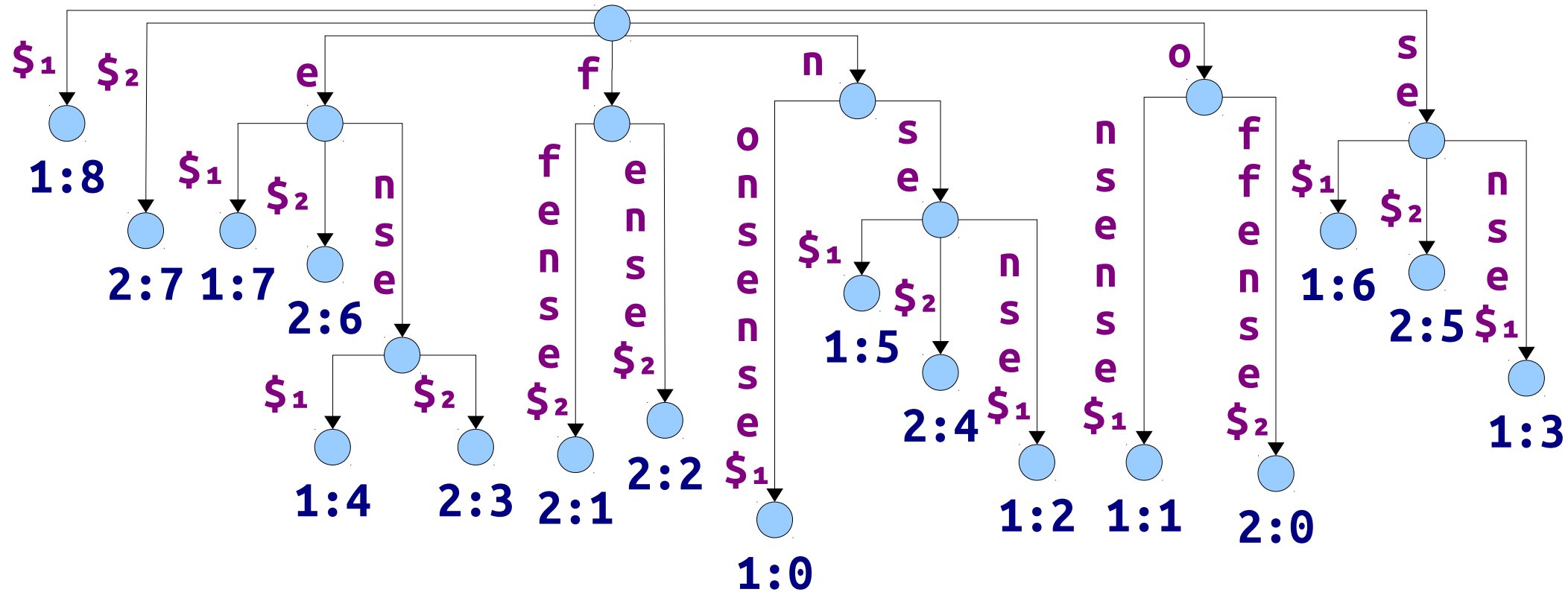
# An Observation



nonsense $\$1$   
012345678

offense $\$2$   
01234567

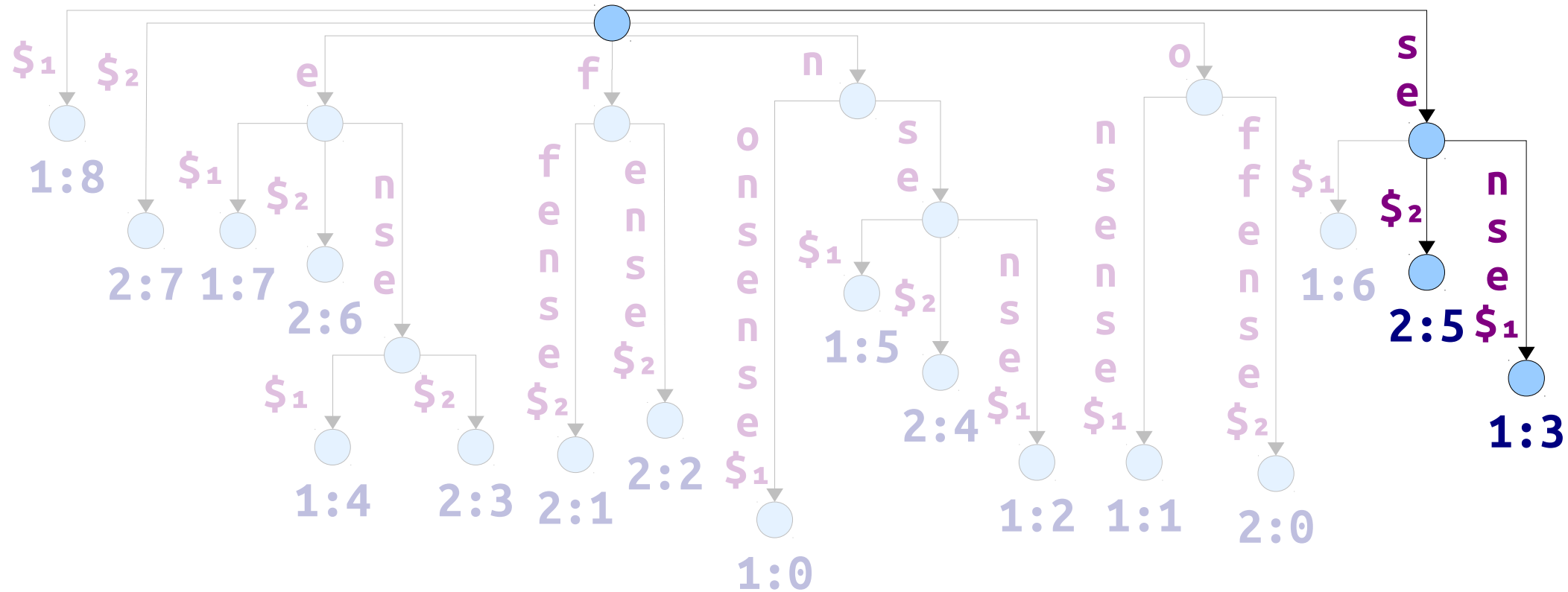
# An Observation



non**sense** $\$1$   
012345678

off**ense** $\$2$   
01234567

# An Observation

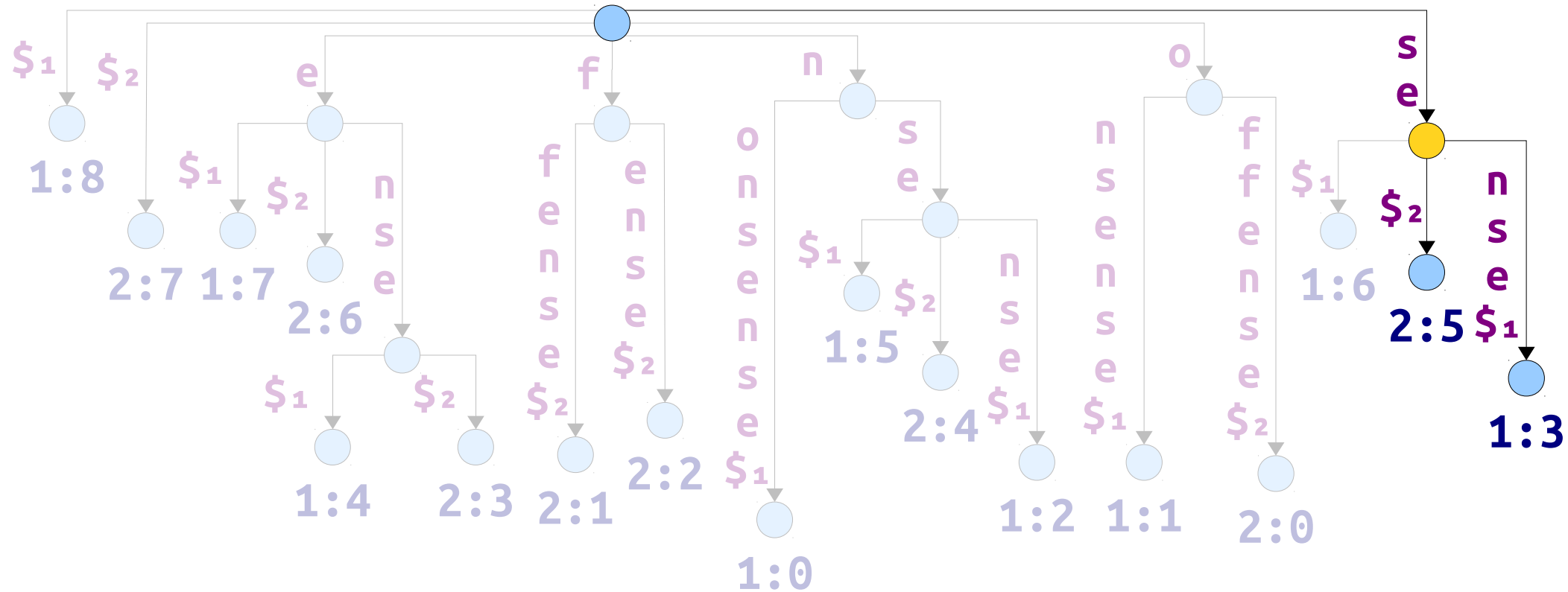


non**sense**\$<sub>1</sub>  
012345678

offense<sub>2</sub>  
01234567



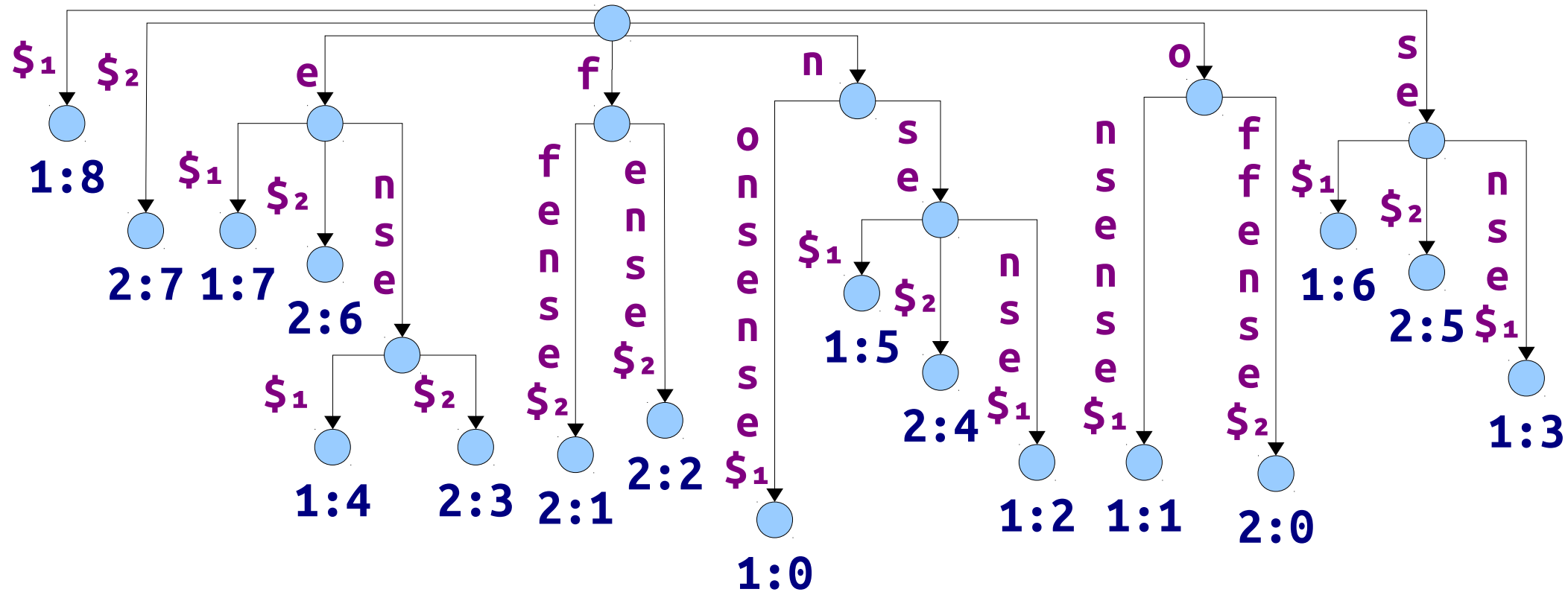
# An Observation



non**sense** $\$1$   
012345678

off**ense** $\$2$   
01234567

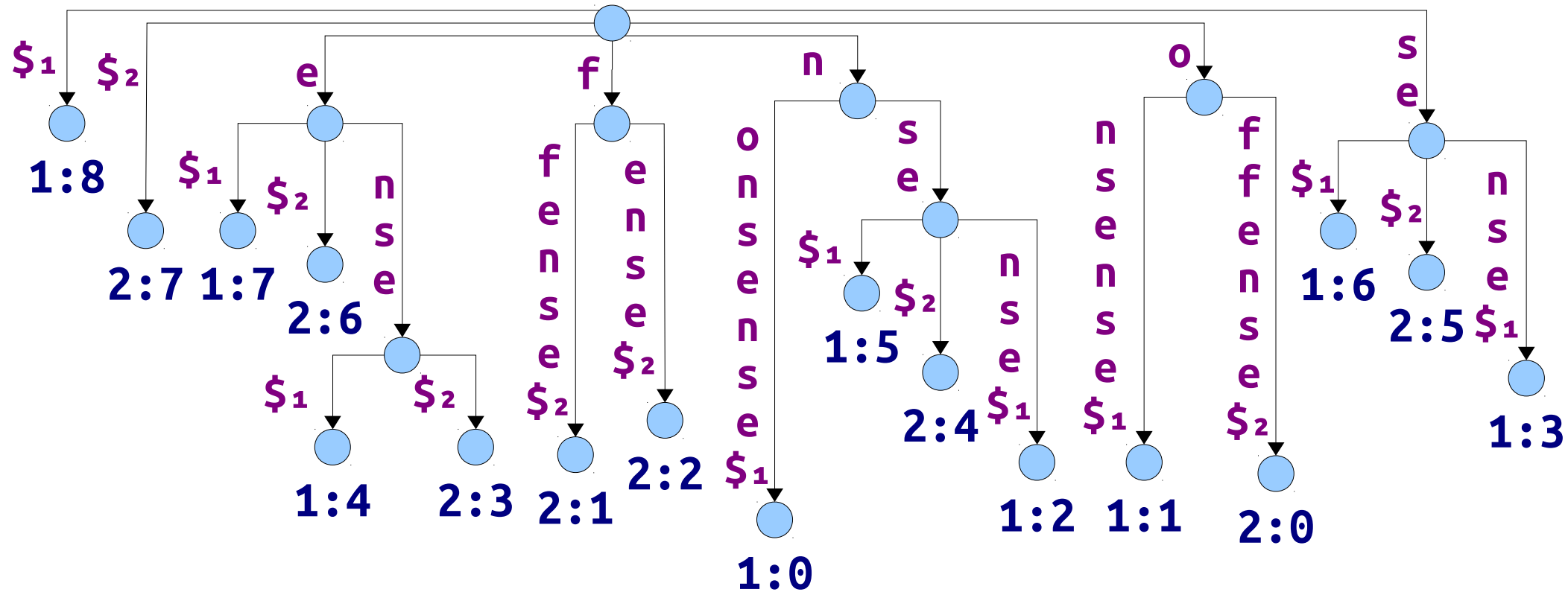
# An Observation



nonsense $\$1$   
012345678

offense $\$2$   
01234567

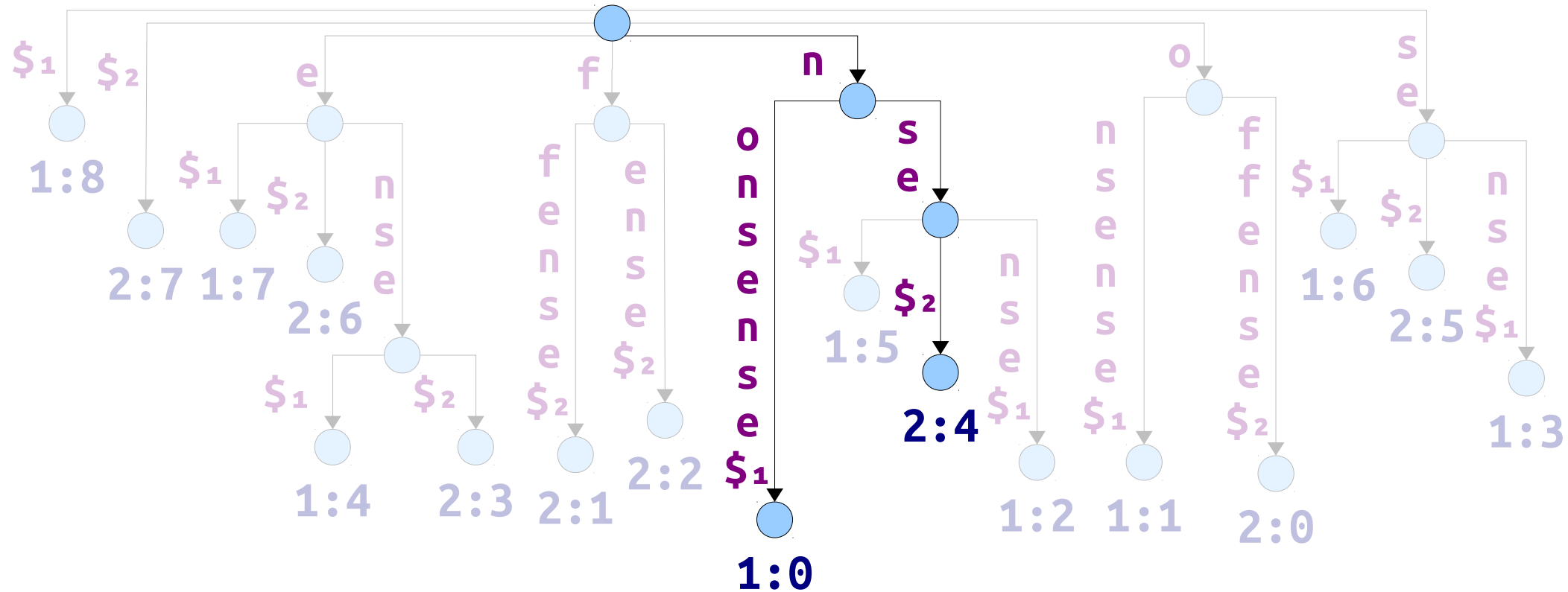
# An Observation



nonsense $\$1$   
012345678

offense $\$2$   
01234567

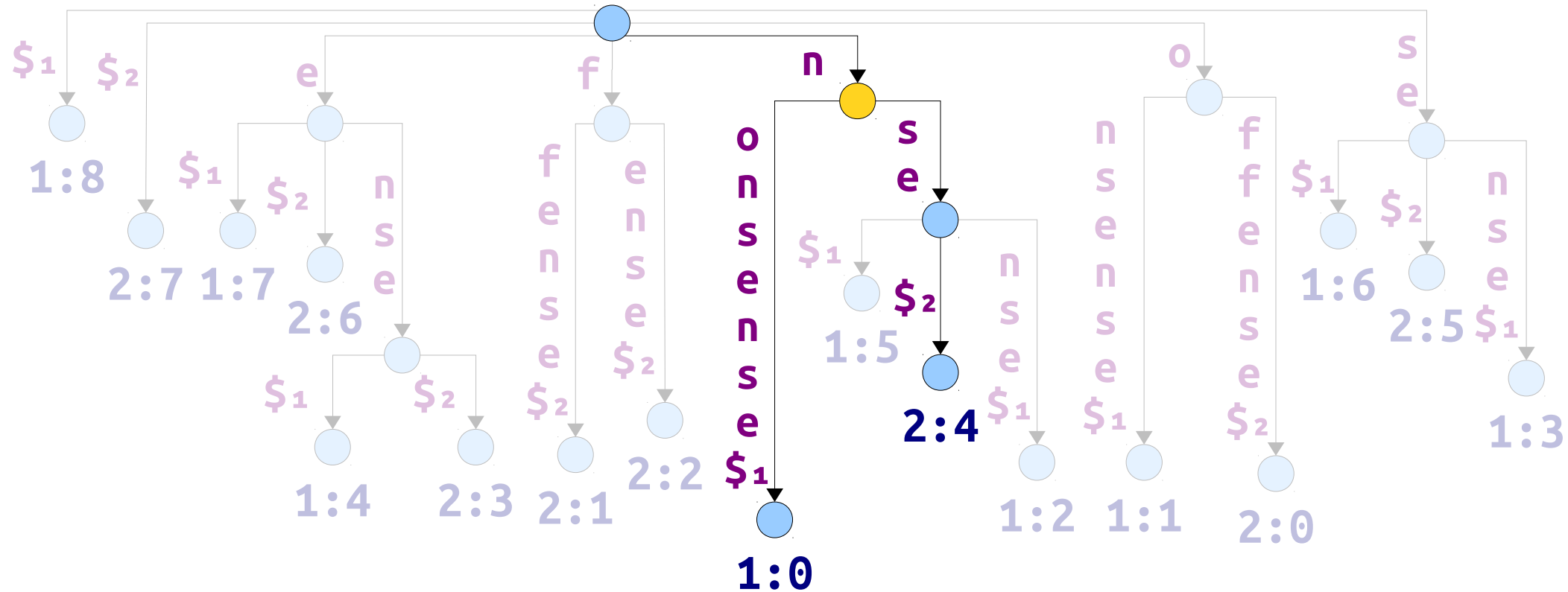
# An Observation



nonsense $\$1$   
012345678

offense $\$2$   
01234567

# An Observation



nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567

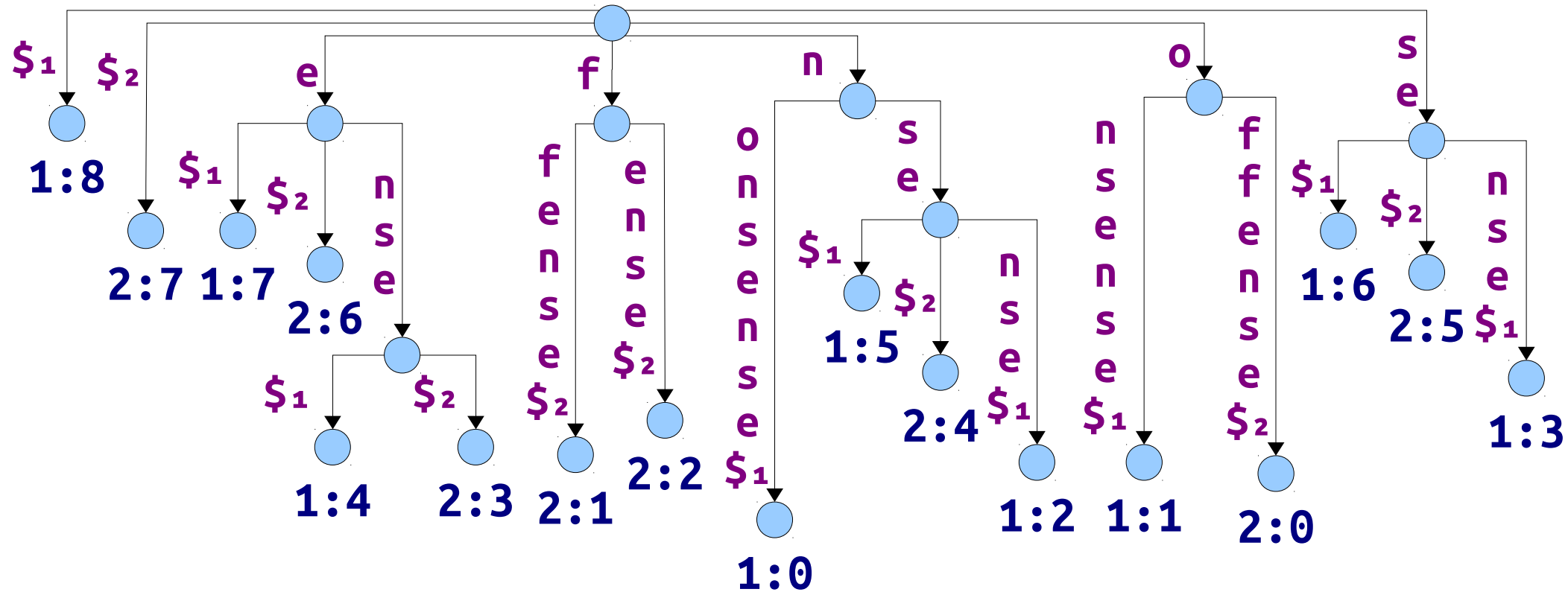
# An Observation

- **Notation:** Let  $S[i:]$  denote the suffix of string  $S$  starting at position  $i$ .
- **Claim:**  $\text{LCE}_{T_1, T_2}(i, j)$  is given by the string label of the LCA of  $T_1[i:]$  and  $T_2[j:]$  in the generalized suffix tree of  $T_1$  and  $T_2$ .
- And hey... don't we have a way of computing these in time  $O(1)$ ?

# Computing LCE's

- Given two strings  $T_1$  and  $T_2$ , construct a generalized suffix tree for  $T_1$  and  $T_2$  in time  $O(m)$ .
- Construct an LCA data structure for the generalized suffix tree in time  $O(m)$ .
  - Use Fischer-Heun plus an Euler tour of the nodes in the tree.
- Can now query for the node representing the LCE in time  $O(1)$ .

# One Last Detail

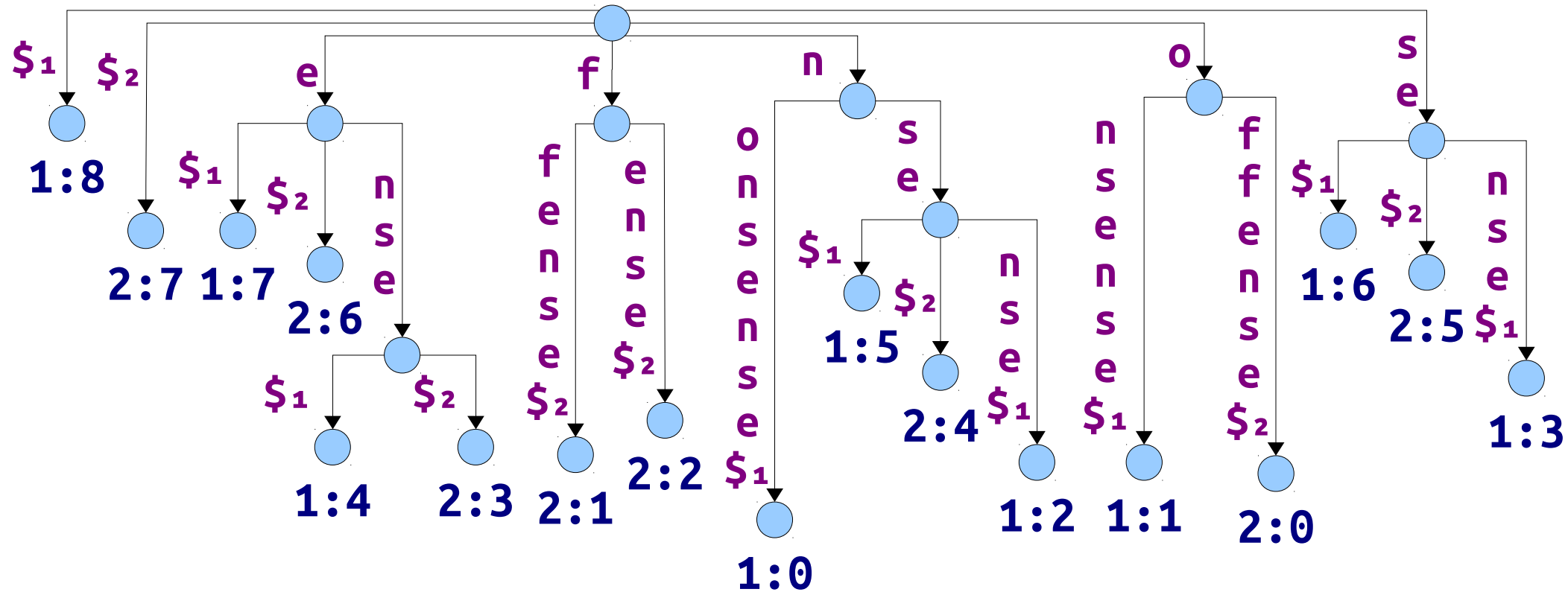


nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567



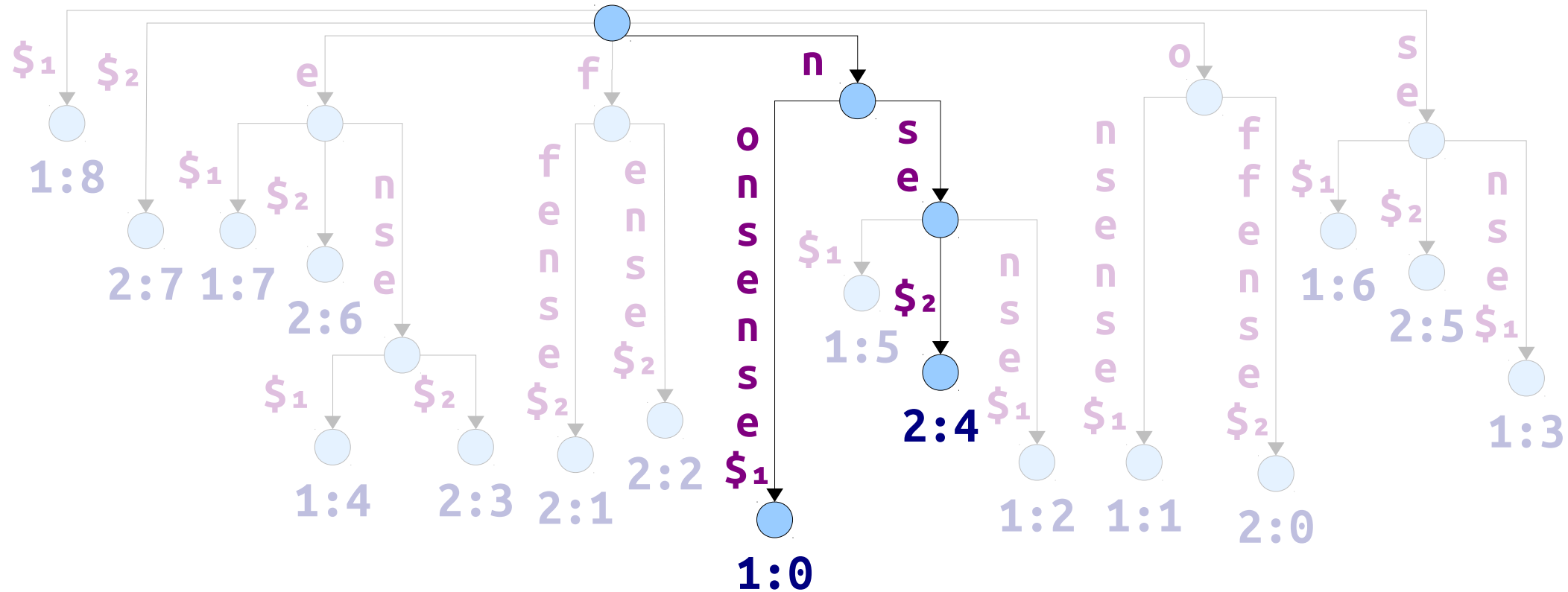
# One Last Detail



nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567

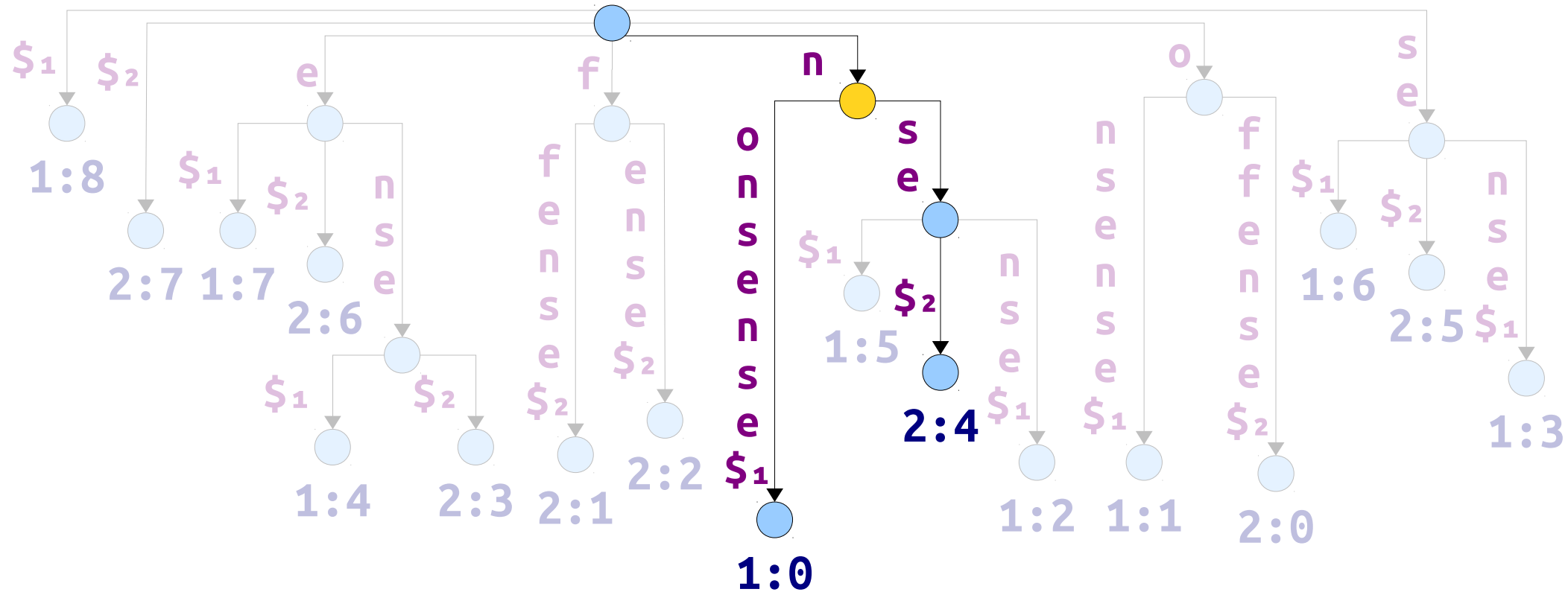
# One Last Detail



nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567

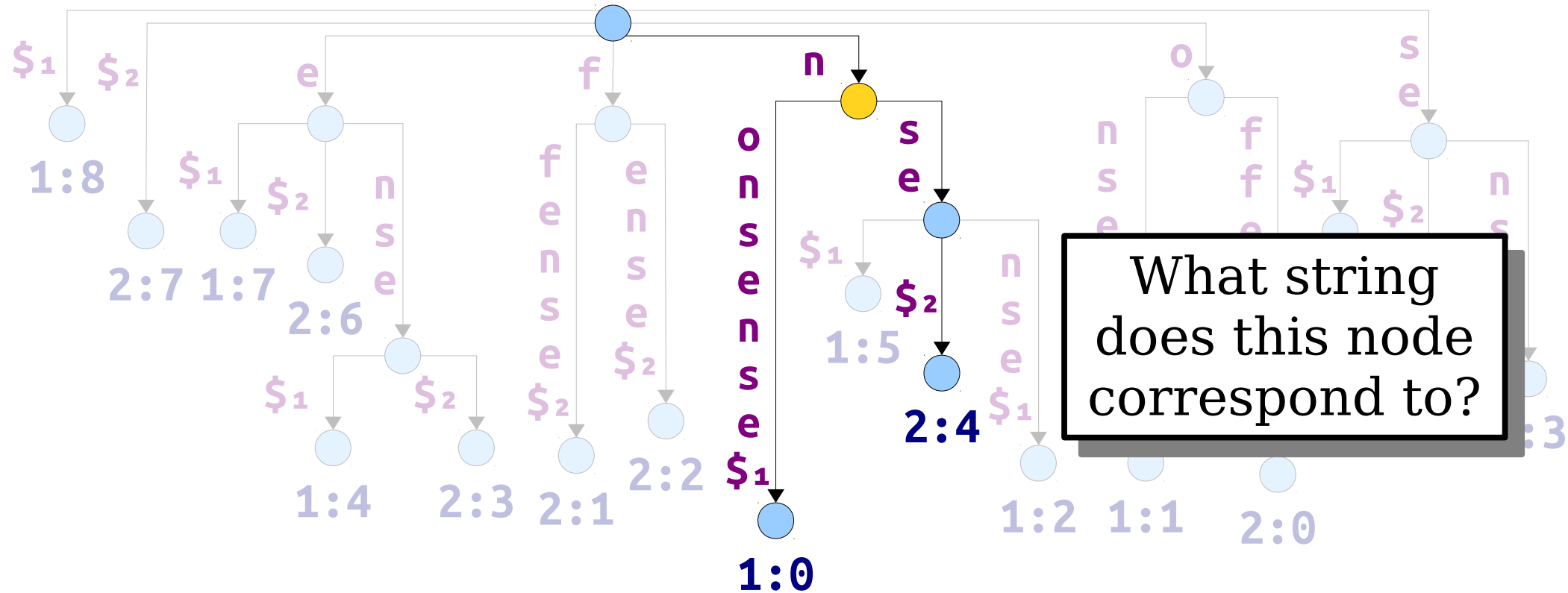
# One Last Detail



nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567

# One Last Detail



nonsense\$<sub>1</sub>  
012345678

offense\$<sub>2</sub>  
01234567

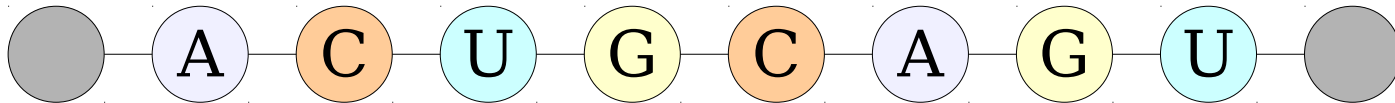
# The Overall Construction

- Using an  $O(m)$ -time DFS, annotate each node in the suffix tree with its string depth.
- To compute LCE:
  - Find the leaves corresponding to  $T_1[i:]$  and  $T_2[j:]$ .
  - Find their LCA; let its string depth be  $d$ .
  - Report  $T_1[i:i + d - 1]$  or  $T_2[j:j + d - 1]$ .
- Overall, requires  $O(m)$  preprocessing time to support  $O(1)$  query time.

***An Application:*** Longest Palindromic  
Substring

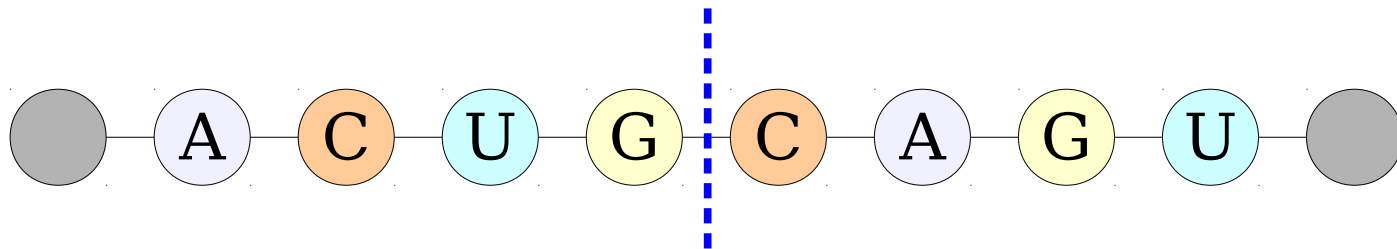
# Palindromes

- A ***palindrome*** is a string that's the same forwards and backwards.
- A ***palindromic substring*** of a string  $T$  is a substring of  $T$  that's a palindrome.
- Surprisingly, of great importance in computational biology.



# Palindromes

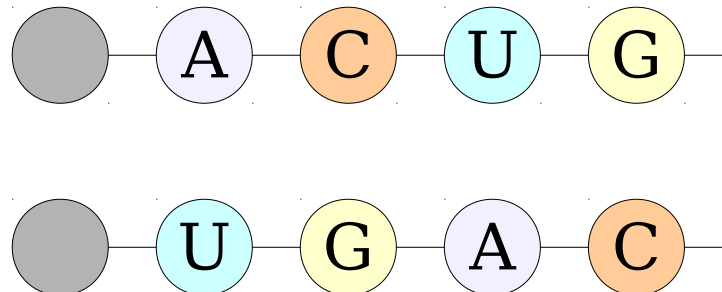
- A ***palindrome*** is a string that's the same forwards and backwards.
- A ***palindromic substring*** of a string  $T$  is a substring of  $T$  that's a palindrome.
- Surprisingly, of great importance in computational biology.





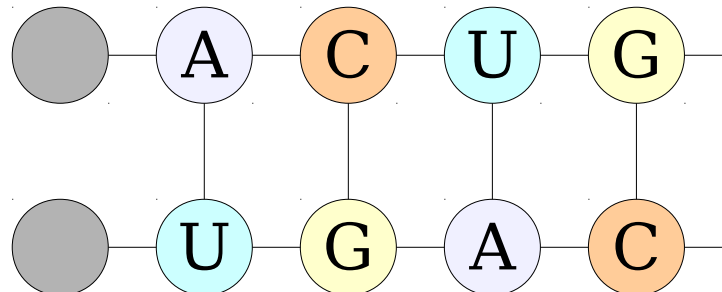
# Palindromes

- A ***palindrome*** is a string that's the same forwards and backwards.
- A ***palindromic substring*** of a string  $T$  is a substring of  $T$  that's a palindrome.
- Surprisingly, of great importance in computational biology.



# Palindromes

- A ***palindrome*** is a string that's the same forwards and backwards.
- A ***palindromic substring*** of a string  $T$  is a substring of  $T$  that's a palindrome.
- Surprisingly, of great importance in computational biology.



# Longest Palindromic Substring

- The *longest palindromic substring* problem is the following:

Given a string  $T$ , find the longest substring of  $T$  that is a palindrome.

- How might we solve this problem?

# An Initial Idea

- To deal with the issues of strings going forwards and backwards, start off by forming  $T$  and  $T^R$ , the reverse of  $T$ .
- **Initial Idea:** Find the longest common substring of  $T$  and  $T^R$ .
- Unfortunately, this doesn't work:
  - $T = \text{abcdabaadbcbabb}$
  - $T^R = \text{bbabcdaabadcba}$
  - Longest common substring: **abcda**
  - Longest palindromic substring: **aa**

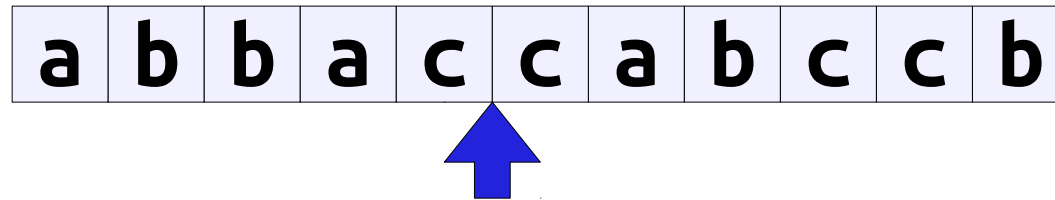
# Palindrome Centers and Radii

- For now, let's focus on even-length palindromes.
- An even-length palindrome substring  $ww^R$  of a string  $T$  has a *center* and *radius*:
  - **Center:** The spot between the duplicated center character.
  - **Radius:** The length of the string going out in each direction.
- **Idea:** For each center, find the largest corresponding radius.

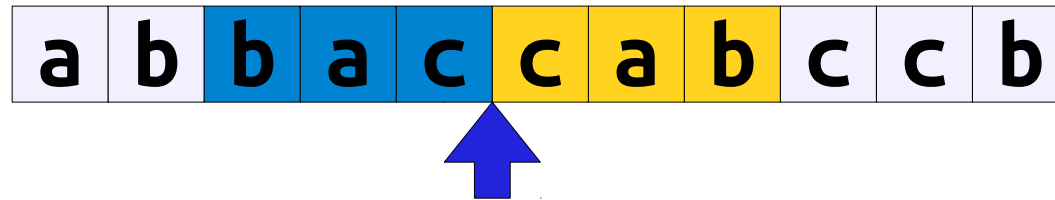
# Palindrome Centers and Radii

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|

# Palindrome Centers and Radii

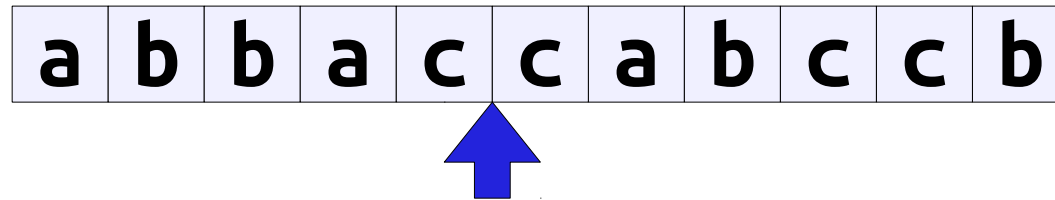


# Palindrome Centers and Radii

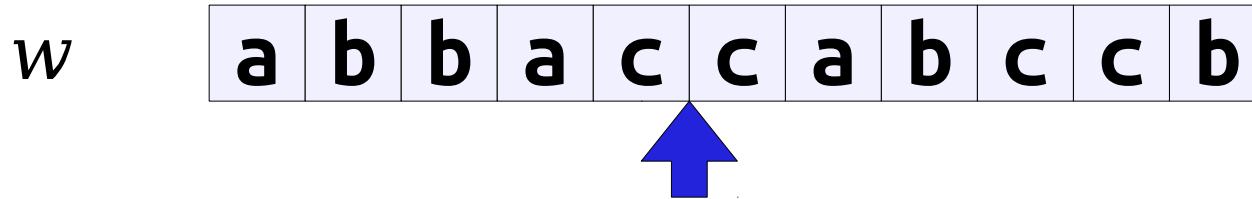




# Palindrome Centers and Radii



# Palindrome Centers and Radii



# Palindrome Centers and Radii

$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|

# Palindrome Centers and Radii

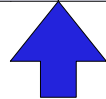
$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|

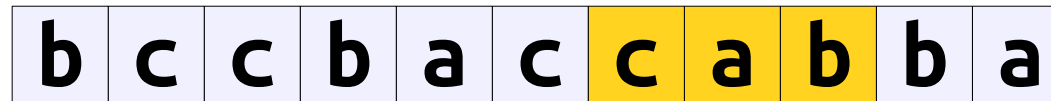


# Palindrome Centers and Radii

$w$



$w^R$



# Palindrome Centers and Radii

$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|

# Palindrome Centers and Radii

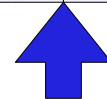
$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|



# Palindrome Centers and Radii

$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|





# Palindrome Centers and Radii

$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|

# Palindrome Centers and Radii

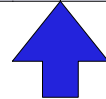
$w$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| a | b | b | a | c | c | a | b | c | c | b |
|---|---|---|---|---|---|---|---|---|---|---|



$w^R$

|   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|
| b | c | c | b | a | c | c | a | b | b | a |
|---|---|---|---|---|---|---|---|---|---|---|



# An Algorithm

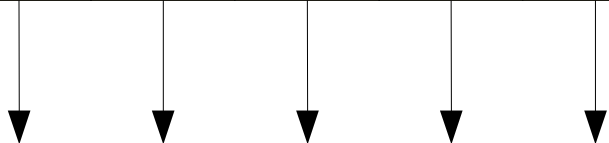
- In time  $O(m)$ , construct  $T^R$ .
- Preprocess  $T$  and  $T^R$  in time  $O(m)$  to support LCE queries.
- For each spot between two characters in  $T$ , find the longest palindrome centered at that location by executing LCE queries on the corresponding locations in  $T$  and  $T^R$ .
  - Each query takes time  $O(1)$  if it just reports the length.
  - Total time:  $O(m)$ .
- Report the longest string found this way.
- Total time:  **$O(m)$** .

# Suffix Trees: The Catch

# Space Usage

- Suffix trees are memory hogs.
- Suppose  $\Sigma = \{A, C, G, T, \$\}$ .
- Each internal node needs 15 machine words: for each character, words for the start/end index and a child pointer.

|       | A | C | T | G | \$ |
|-------|---|---|---|---|----|
| start | 8 | 4 | 0 | 1 | 3  |
| end   | 8 | 4 | 0 | 8 | 4  |
| child |   |   |   |   |    |



This is still  $O(m)$ , but it's a *huge* hidden constant!

# Combating Space Usage

- In 1990, Udi Manber and Gene Myers introduced the *suffix array* as a space-efficient alternative to suffix trees.
- Requires one word per character; typically, an extra word is stored as well (details next Tuesday)
- Can't support all operations permitted by suffix trees, but has much better performance.
- Curious? Details are next time!

# Summary

- Given a string, it's possible to build a suffix tree for it in time  $\Theta(m)$ . Suffix trees support
  - efficient detection of all matching substrings,
  - efficient detection of duplicated substrings,
  - efficient detection of common substrings,
  - efficient detection of common extensions,and a lot more!
- Suffix trees use space  $\Theta(m)$ , but with a huge hidden constant factor.
- Building suffix trees is hard. We'll see how to do it next time.

# Next Time

- **Suffix Arrays**
  - A space-efficient alternative to suffix trees.
- **LCP Arrays**
  - A useful auxiliary data structure for speeding up suffix arrays.
- **Constructing Suffix Trees**
  - How on earth do you build suffix trees in time  $O(m)$ ?
- **Constructing Suffix Arrays**
  - Start by building suffix arrays in time  $O(m)$ ...
- **Constructing LCP Arrays**
  - ... and adding in LCP arrays in time  $O(m)$ .