Suffix Trees and Suffix Arrays

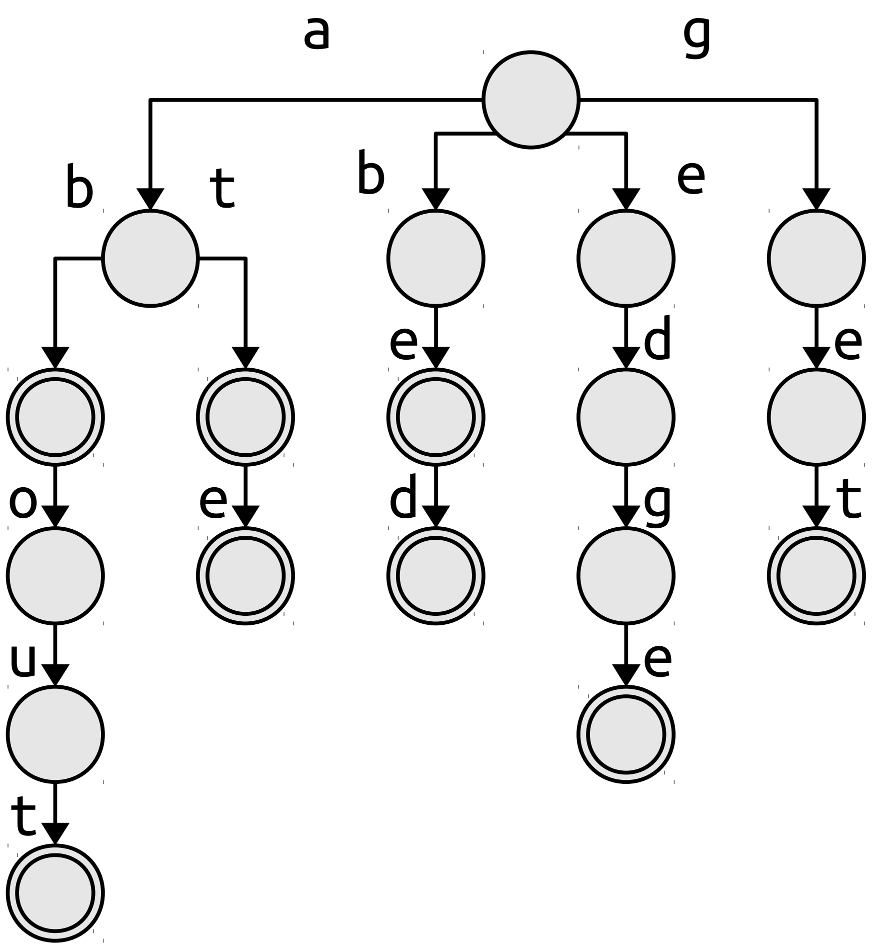
Outline for Today

- ***Suffix Tries***
  - A simple data structure for string searching.
- ***Suffix Trees***
  - A powerful, and flexible data structure for string algorithms.
- ***Suffix Arrays***
  - A compact alternative to suffix trees.
- ***Applications of Suffix Trees and Arrays***
  - There are many!

Review from Last Time

Tries

- A ***trie*** is a tree that stores a collection of strings over some alphabet Σ.
- 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 $P_1, \ldots, P_k$ inside a string T.
- Runtime is $\langle O(n), O(m + z) \rangle$, where
    - $m = |T|$,
    - $n = |P_1| + \ldots + |P_k|$, and
    - z is the number of matches.
- Great for the case where the patterns are fixed and the text to search changes.

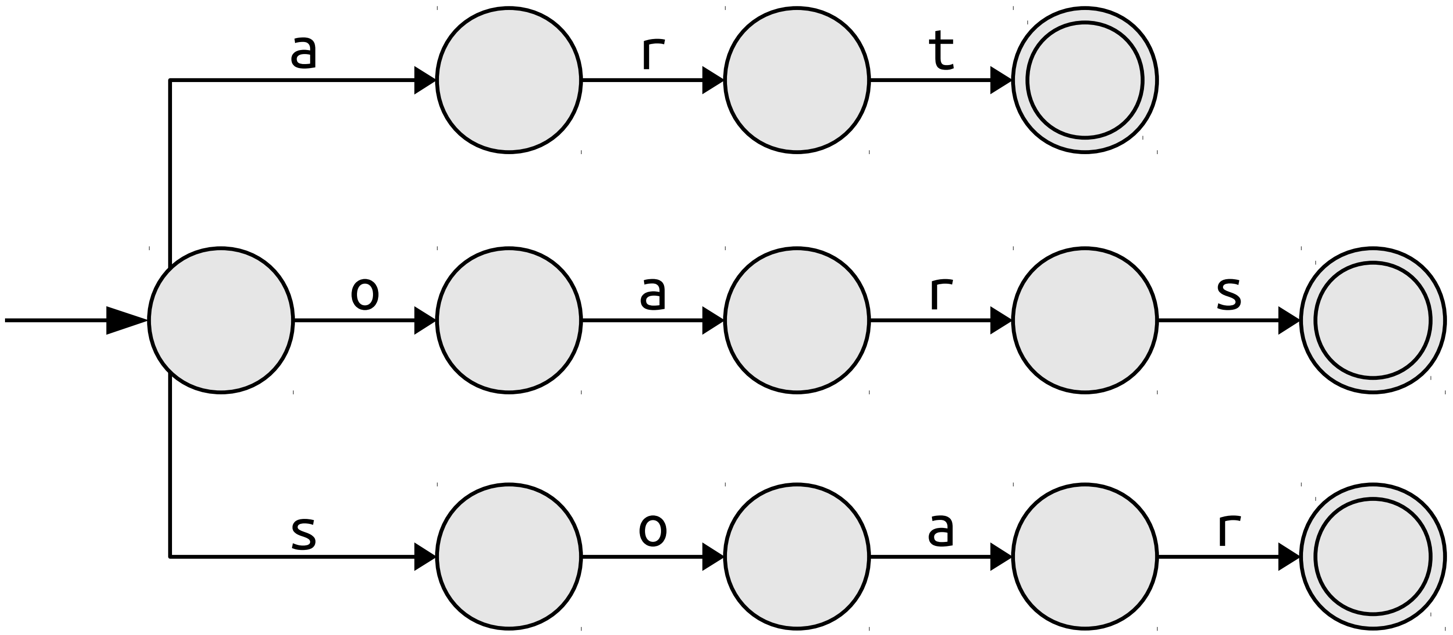
Genomics Databases

- 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, …, 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

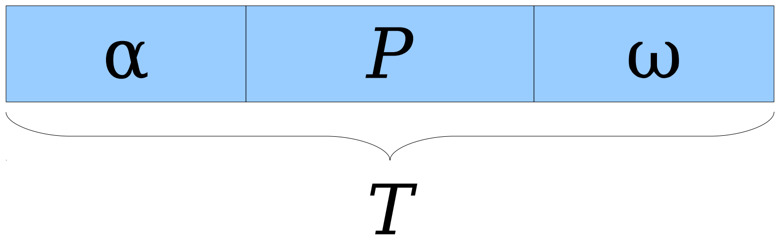
Substrings, Prefixes, and Suffixes

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



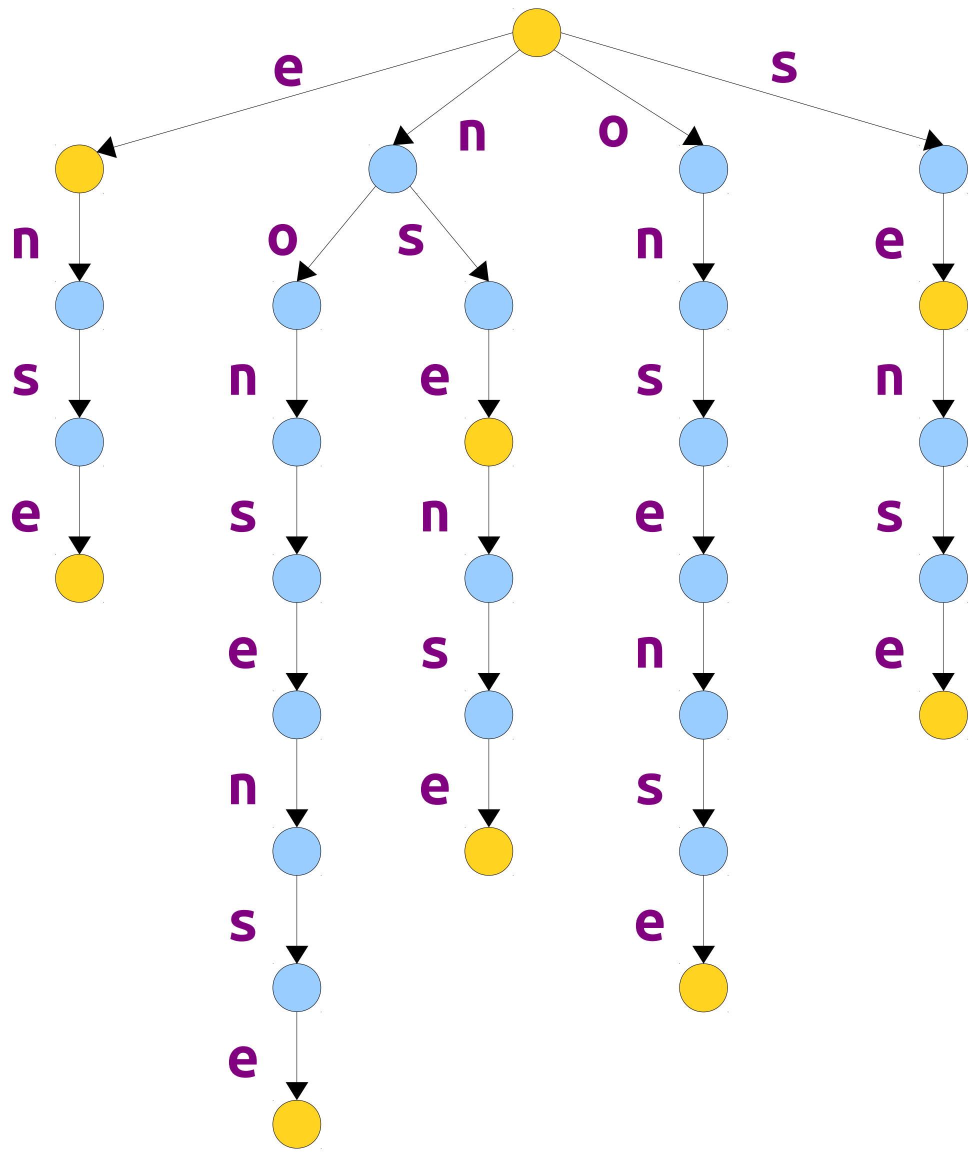
Substrings, Prefixes, and Suffixes

- ***Useful Fact 1:*** Given a trie storing a set of strings S_1, S_2, …, 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 P is a prefix of the suffix $P\omega$ of T.



Suffix Tries

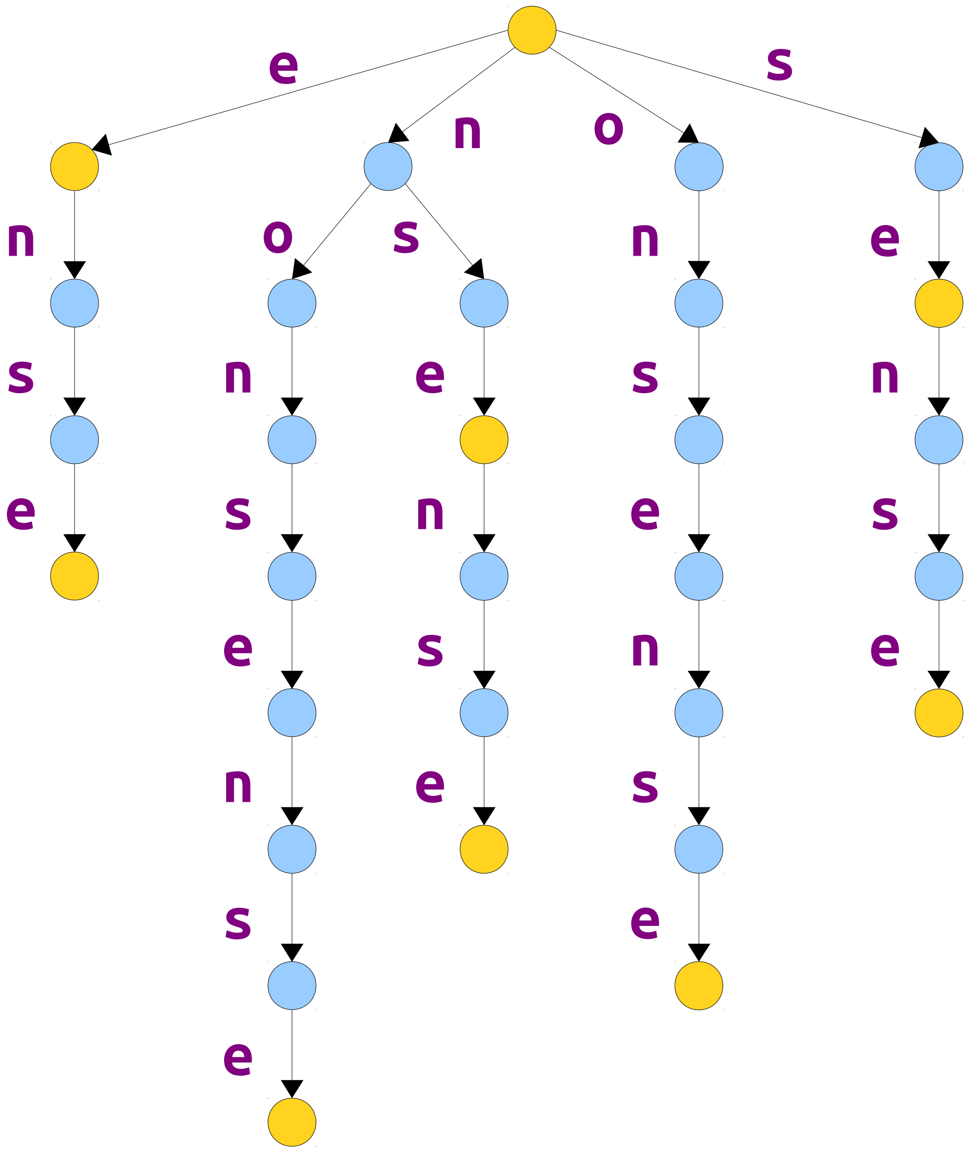
- A ***suffix trie*** of *T* is a trie of all the suffixes of *T*.
- Given any pattern string *P*, we can check in time O(|*P*|) whether *P* is a substring of *T* by seeing whether *P* is a prefix in *T*'s suffix trie.
    - (This checks whether *P* is a prefix of some suffix of *T*.)



nonsense

Suffix Tries

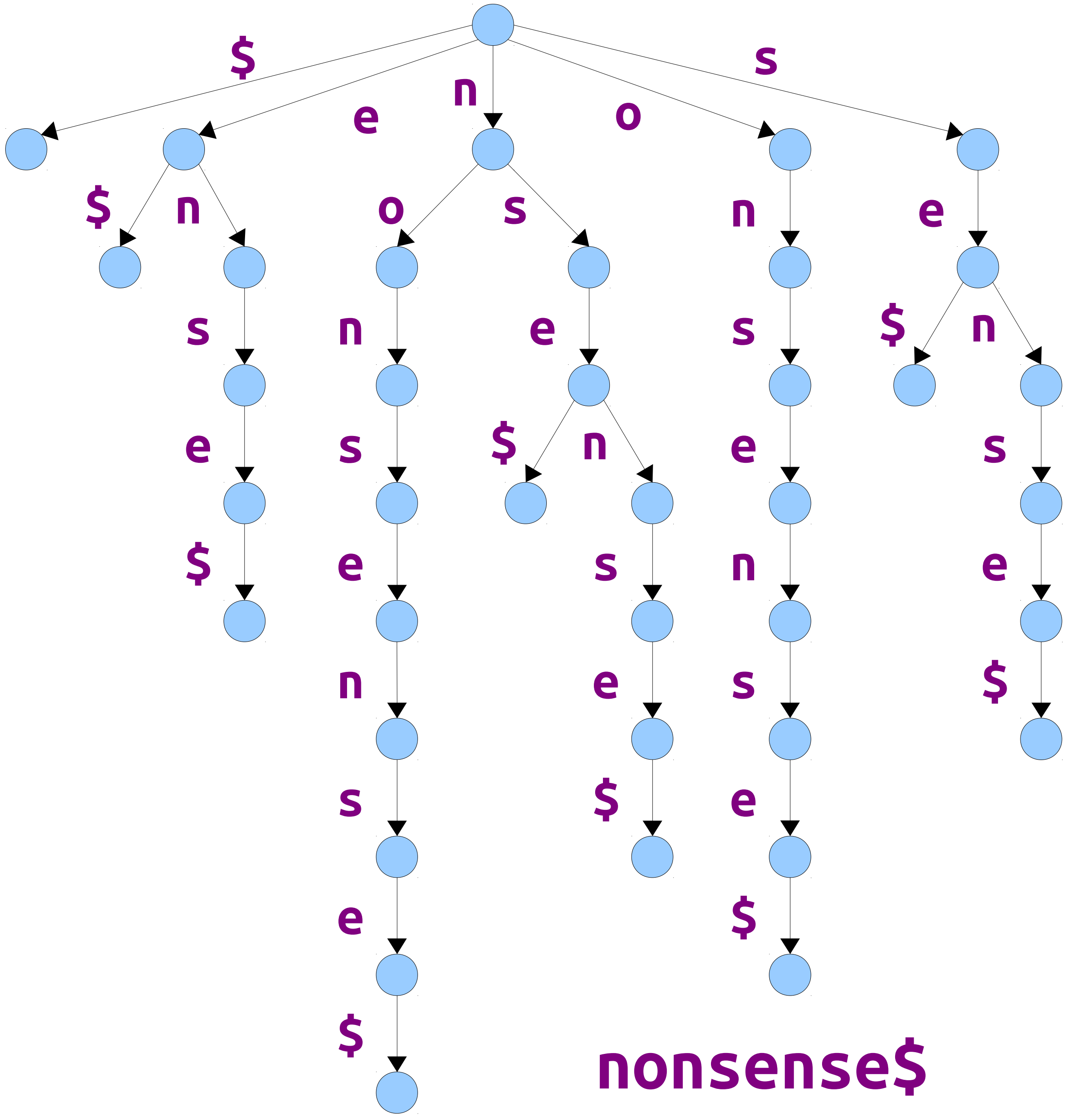
- A ***suffix trie*** of *T* is a trie of all the suffixes of *T*.
- More generally, given any nonempty patterns $P_1, \ldots, P_k$ of total length *n*, we can detect how many of those patterns are substrings of *T* in time O(*n*).
- (Finding all matches is a bit trickier; more on that later.)



nonsense

A Typical Transform

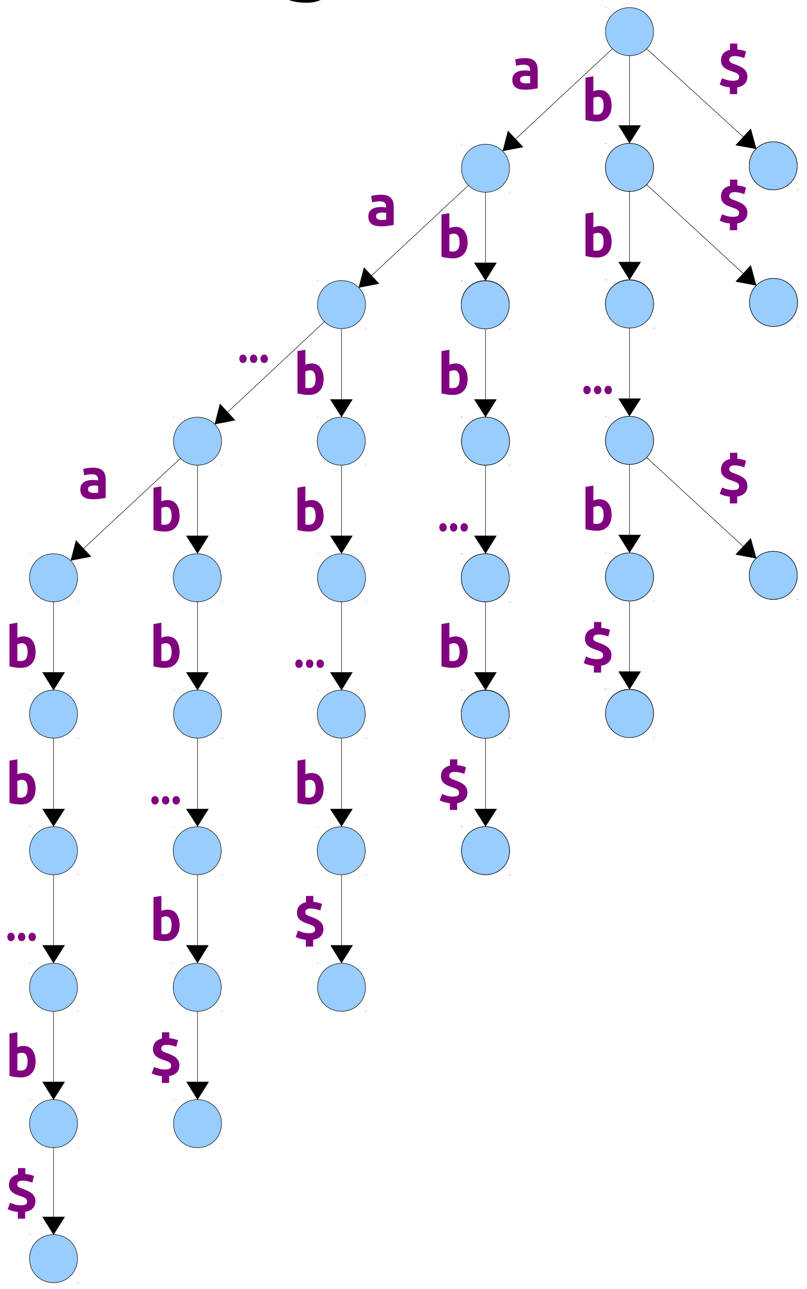
- Append some new character $\$ \notin \Sigma$ to the end of T, then construct the trie for $T\$$.
- The new $ character lexicographically precedes all other characters.
    - This is usually called the ***sentinel***; think of it like the Theoryland version of a null terminator.
- Leaf nodes correspond to suffixes.
- Internal nodes correspond to prefixes of those suffixes.



Constructing Suffix Tries

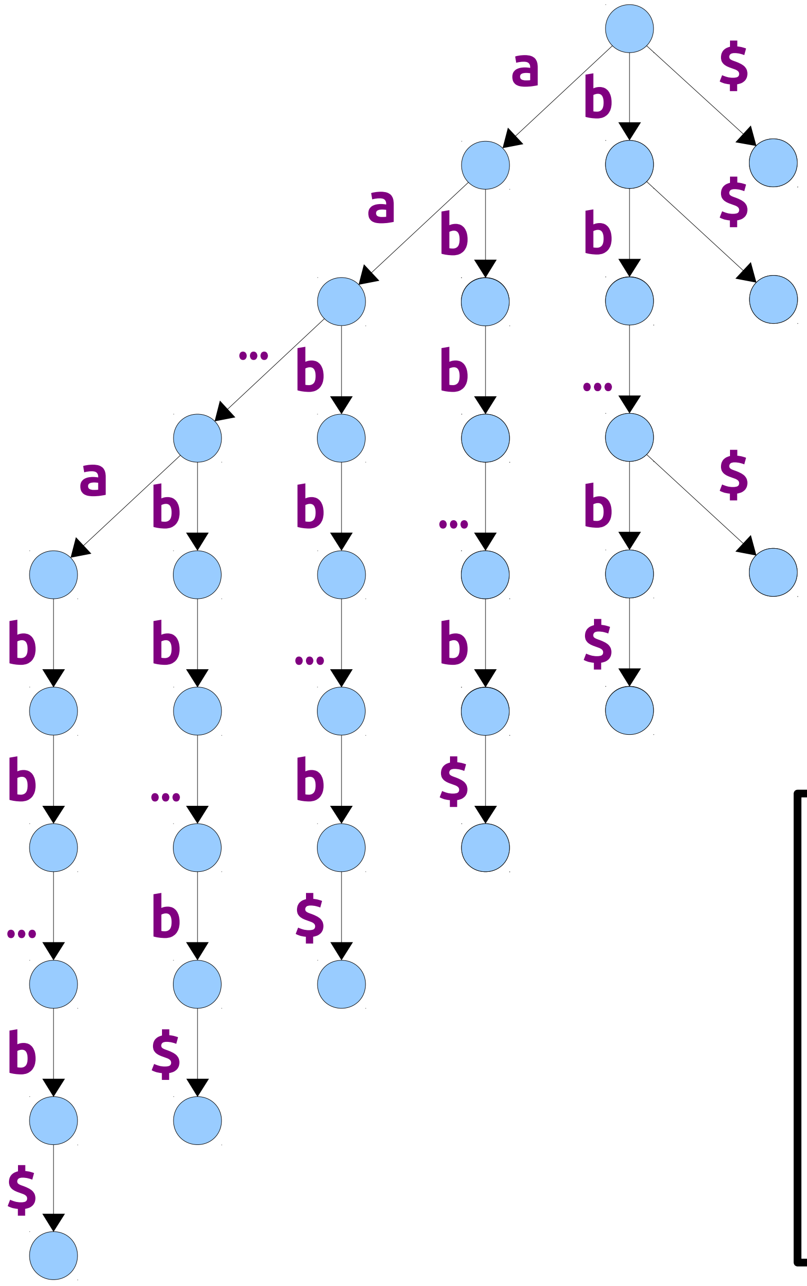
- Once we build a single suffix trie for string *T*, we can efficiently detect whether patterns match in time O(n).
- ***Question:*** How long does it take to construct a suffix trie?
- ***Problem:*** There's an $\Omega(m^2)$ lower bound on the worst-case complexity of *any* algorithm for building suffix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

A Degenerate Case



$a^m b^m \$$

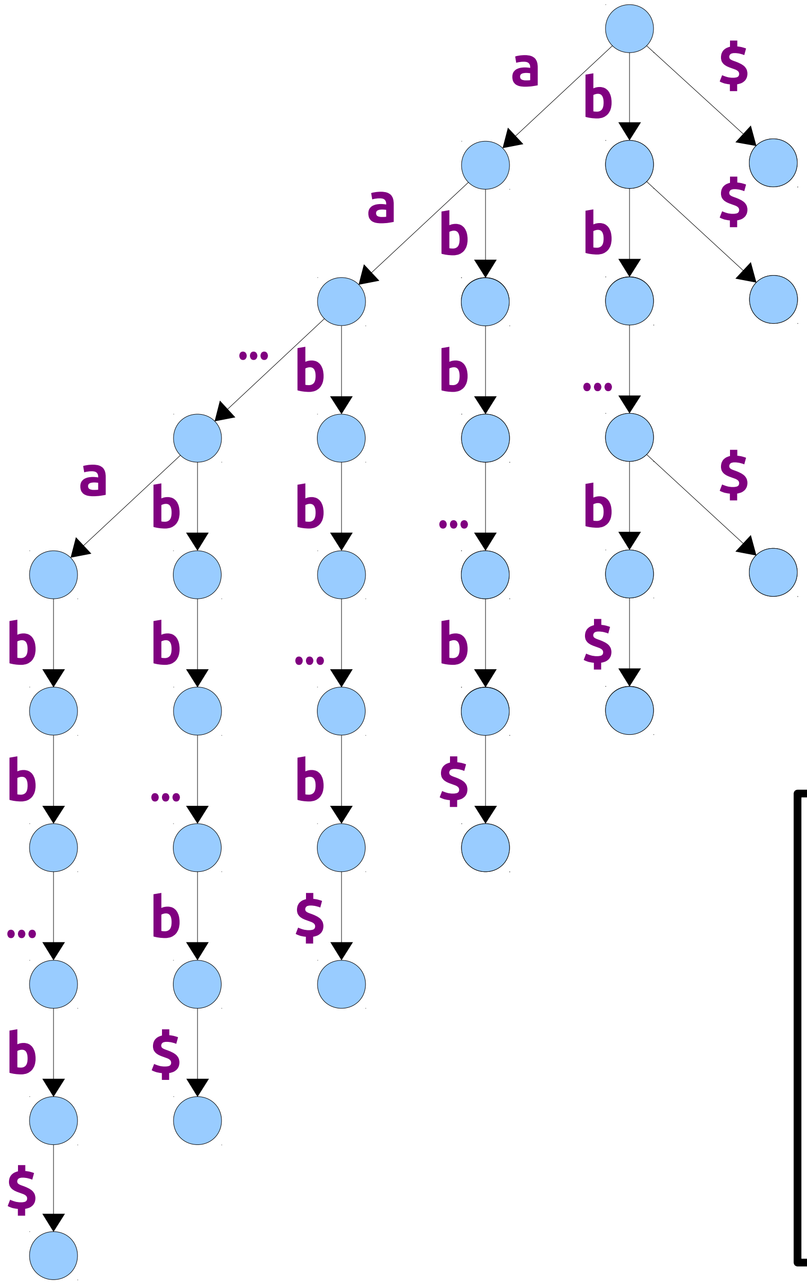
A Degenerate Case



$a^m b^m \$$

There are $\Theta(m)$ copies of nodes chained together as $b^m\$$.

A Degenerate Case



$a^m b^m \$$

There are $\Theta(m)$ copies of nodes chained together as $b^m\$$.

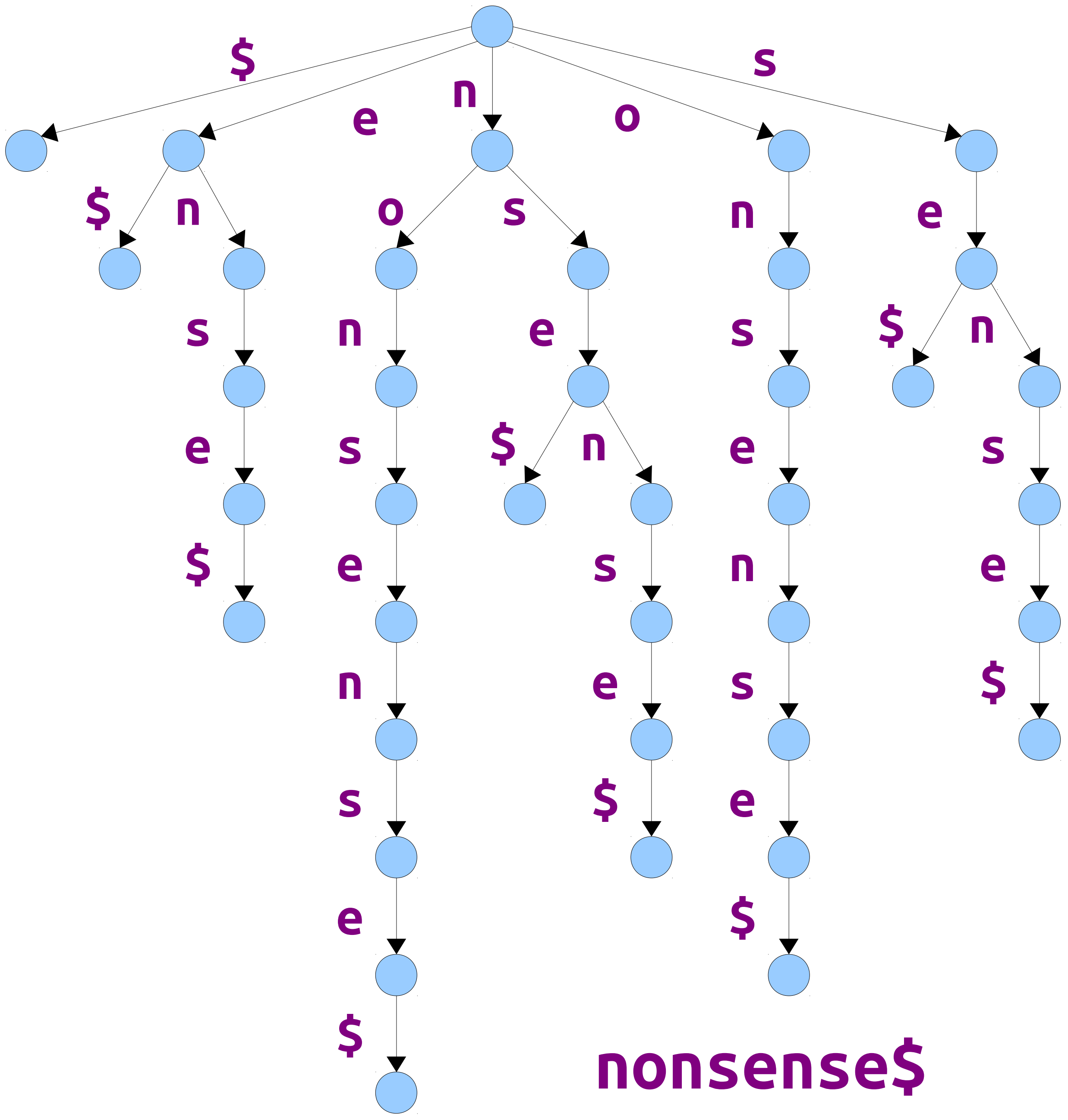
Space usage: $\Omega(m^2)$.

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?

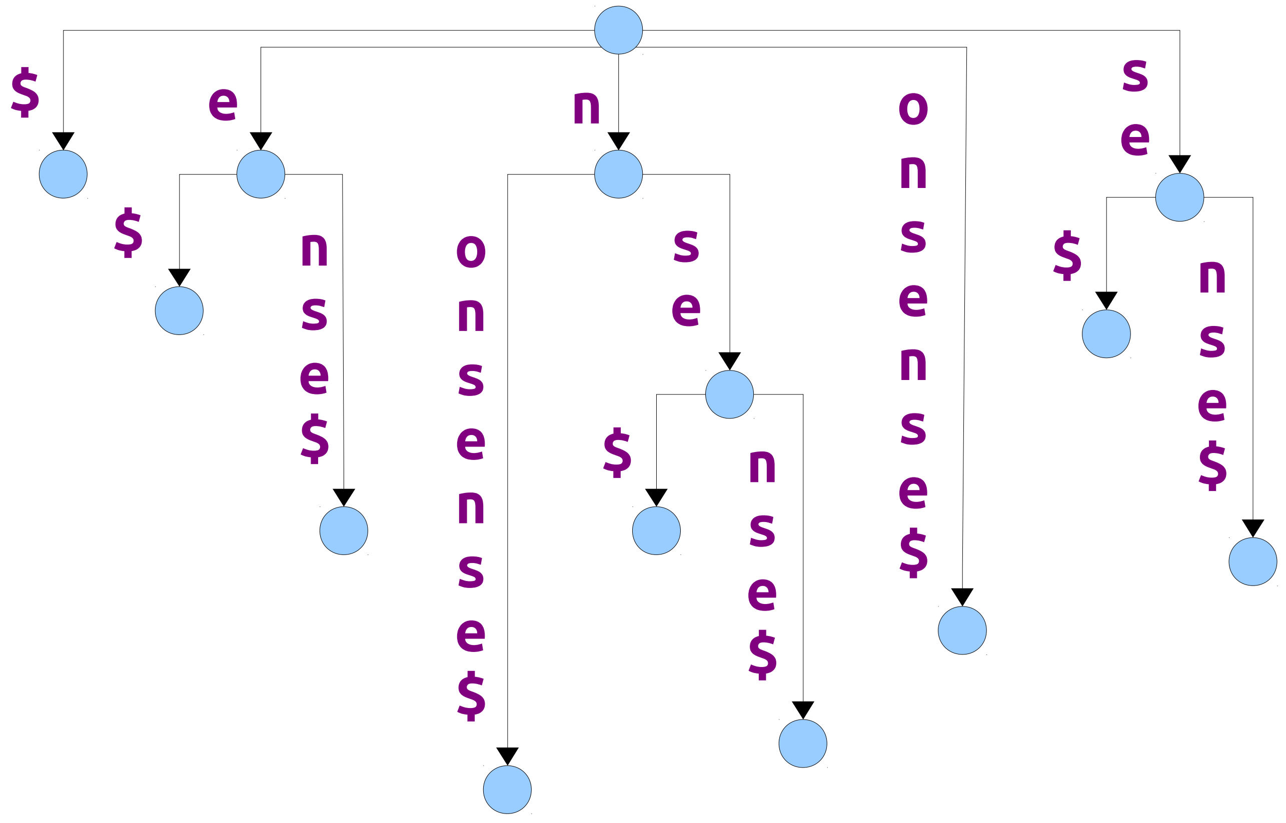
Patricia Tries

- 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

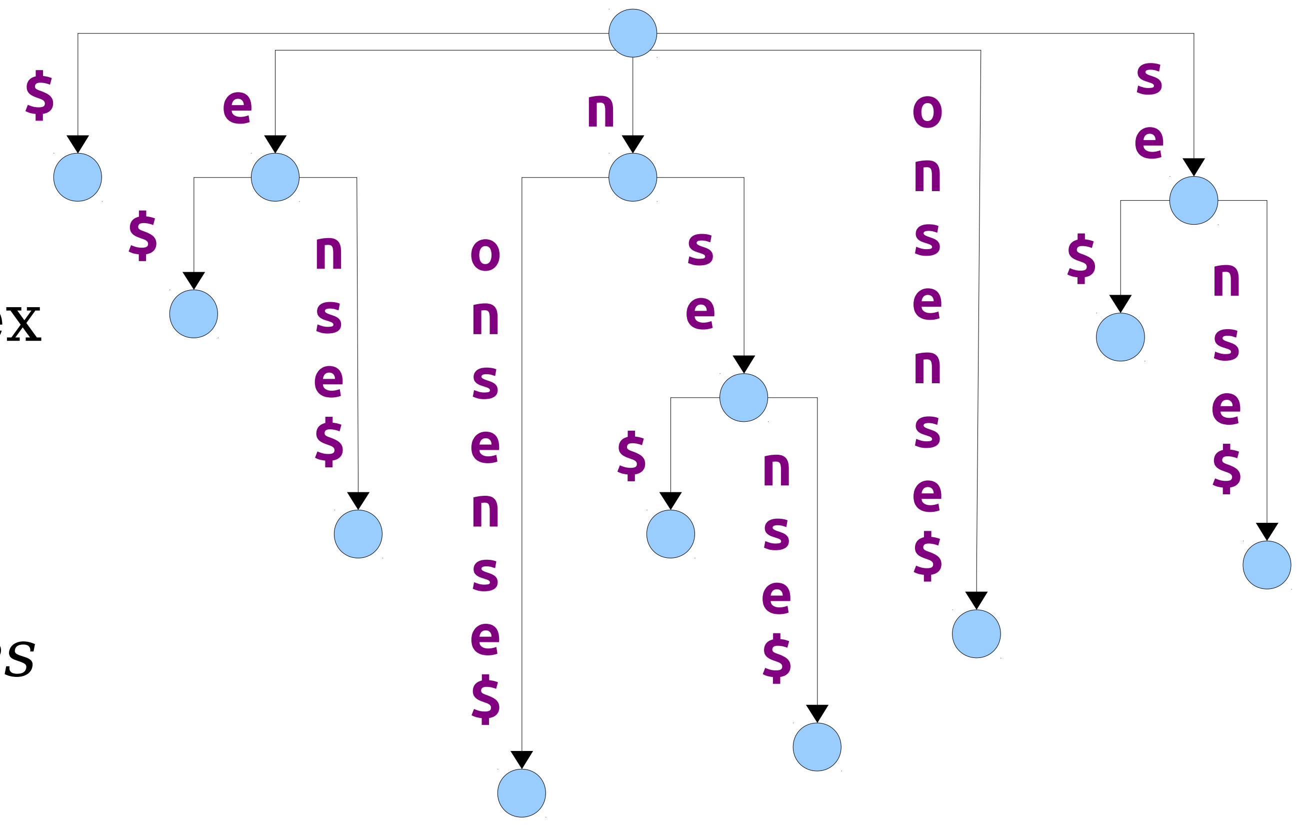
- A “silly” node in a trie is a node that has exactly one child.
- A ***Patricia trie*** (or ***radix trie***) is a trie where all “silly” nodes are merged with their parents.



nonsense$

Suffix Trees

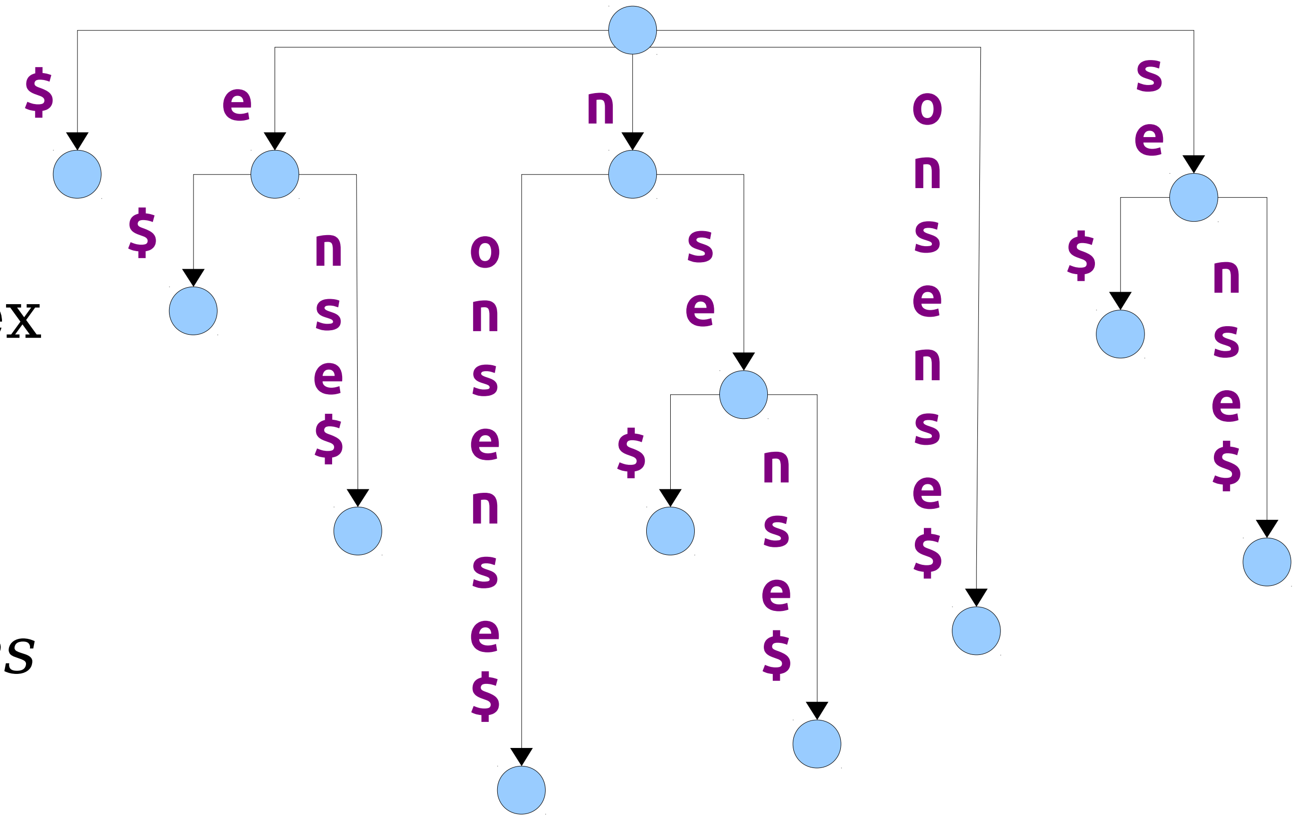
- A ***suffix tree*** for a string *T* is an Patricia trie of *T*$ where each leaf is labeled with the index where the corresponding suffix starts in *T*$.
- (Note that suffix *trees* aren't the same as suffix *tries*. To the best of my knowledge, suffix *tries* aren't used anywhere.)



nonsense$

Suffix Trees

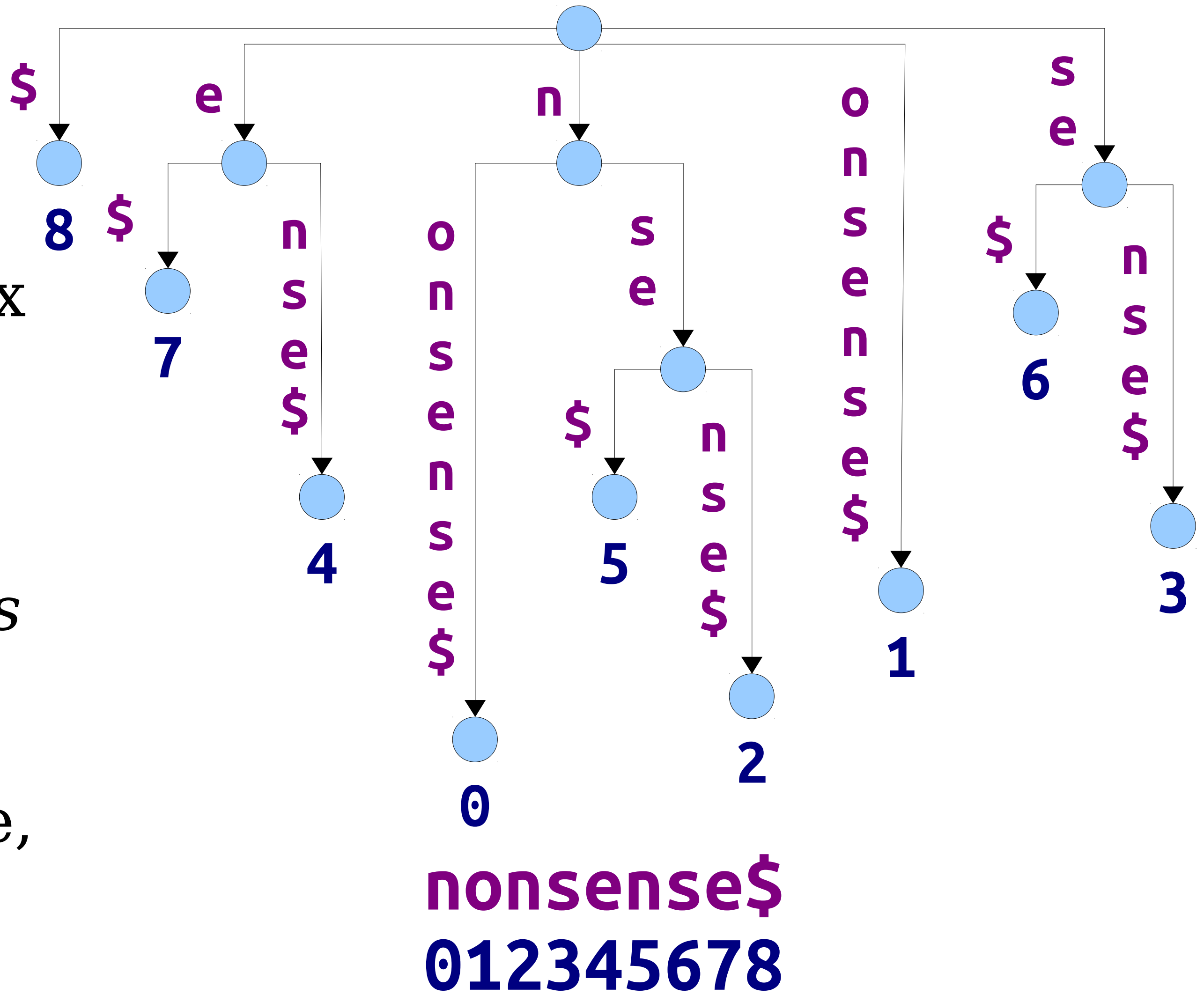
- A ***suffix tree*** for a string *T* is an Patricia trie of *T*$ where each leaf is labeled with the index where the corresponding suffix starts in *T*$.
- (Note that suffix *trees* aren't the same as suffix *tries*. To the best of my knowledge, suffix *tries* aren't used anywhere.)



nonsense$
012345678

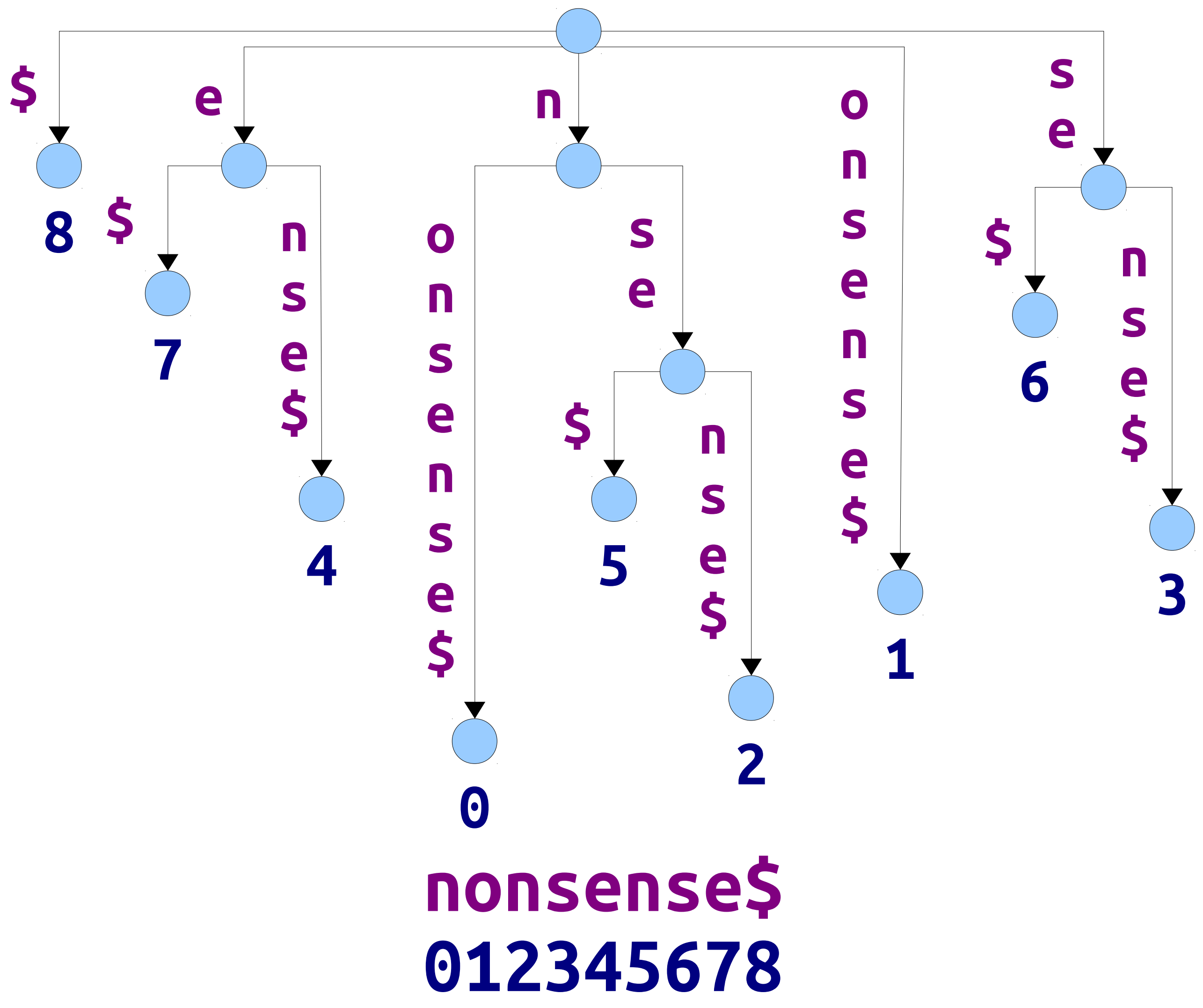
Suffix Trees

- 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*$.
- (Note that suffix *trees* aren't the same as suffix *tries*. To the best of my knowledge, suffix *tries* aren't used anywhere.)



Properties of Suffix Trees

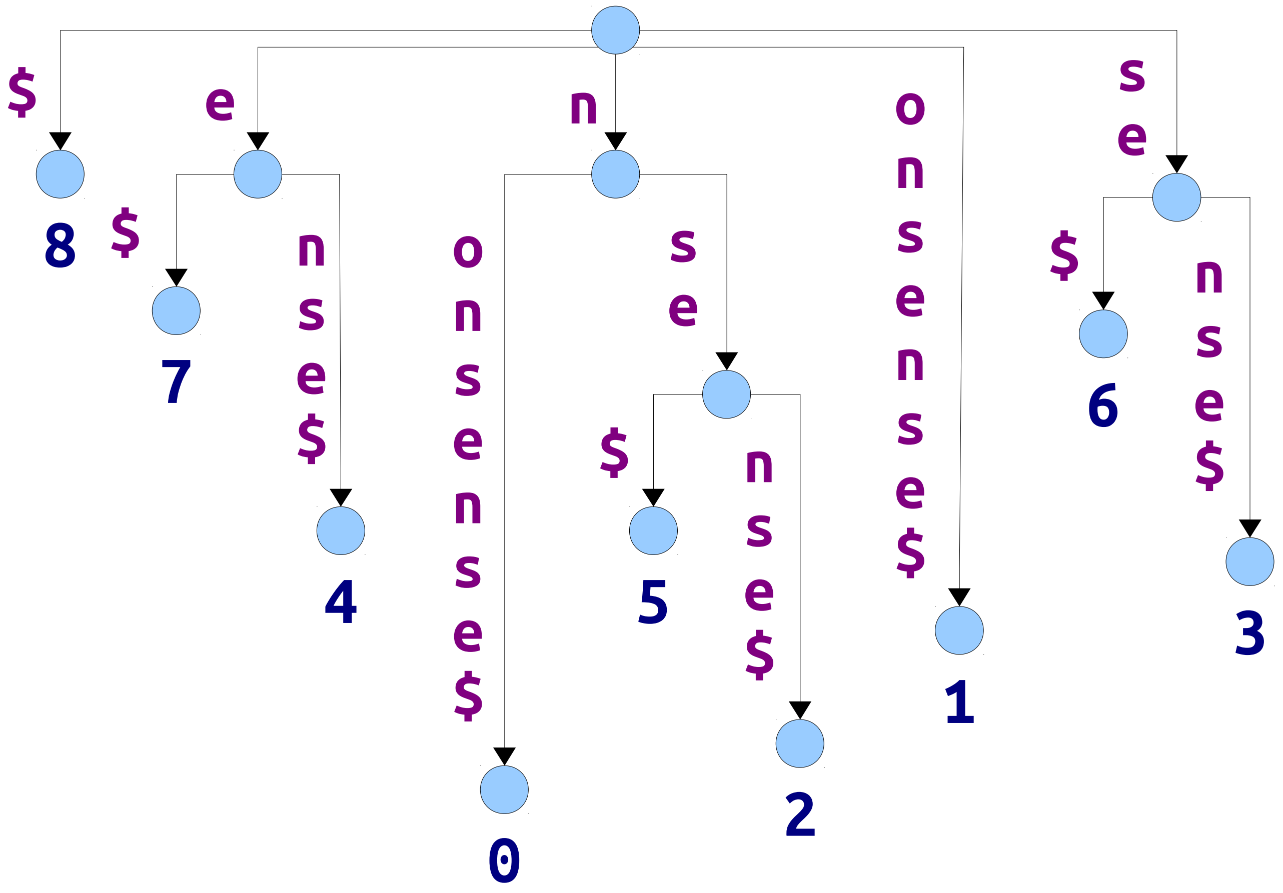
- If $|T| = m$, the suffix tree has exactly $m + 1$ leaf nodes.
- For any $T \neq \varepsilon$, all internal nodes in the suffix tree have at least two children.
- Number of nodes in a suffix tree is $\Theta(m)$.



Suffix Tree Representations

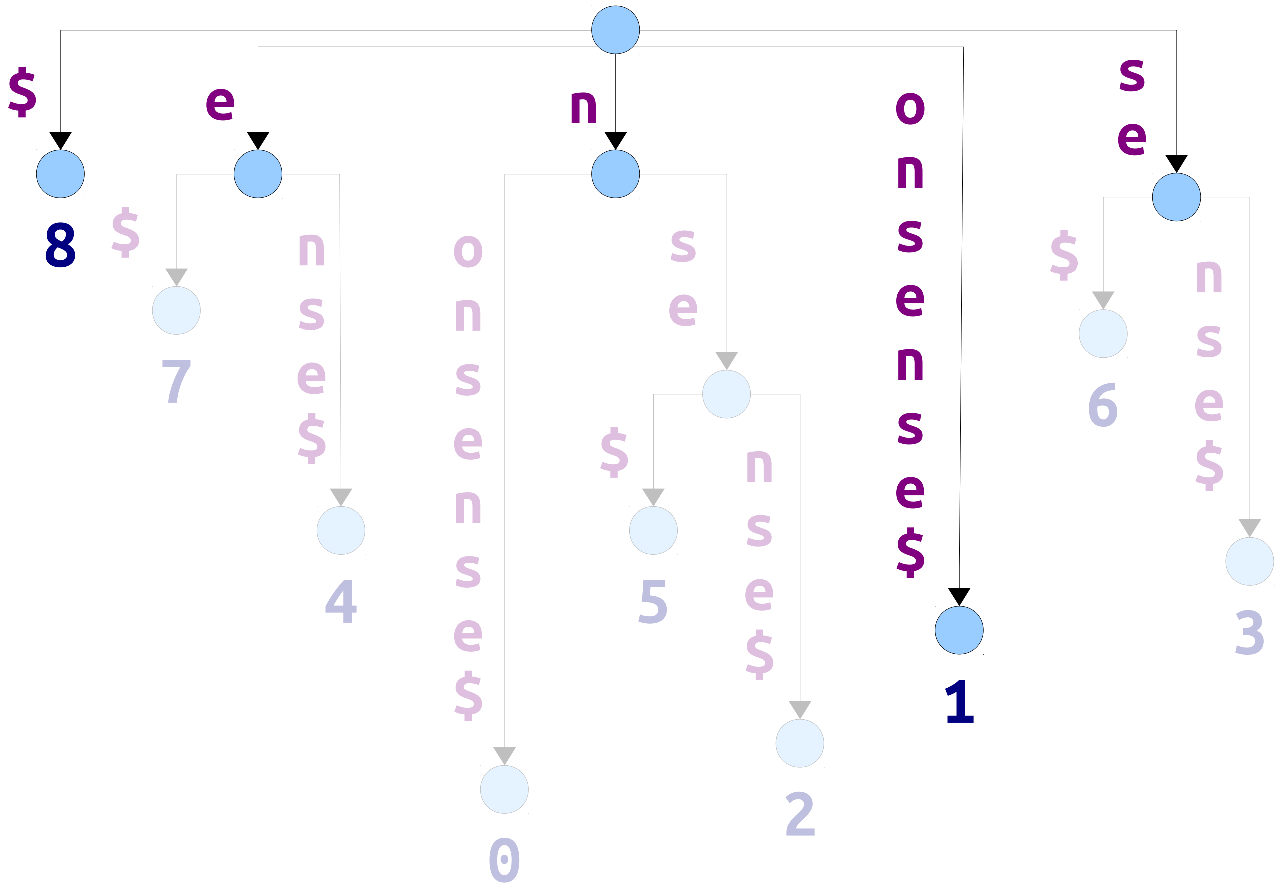
- Suffix trees may have $\Theta(m)$ nodes, but the labels on the edges can have size $\omega(1)$.
- This means that a naïve representation of a suffix tree may take $\omega(m)$ space.
- ***Useful fact:*** Each edge in a suffix tree is labeled with a consecutive range of characters from w.
- ***Trick:***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

Suffix Tree Representations



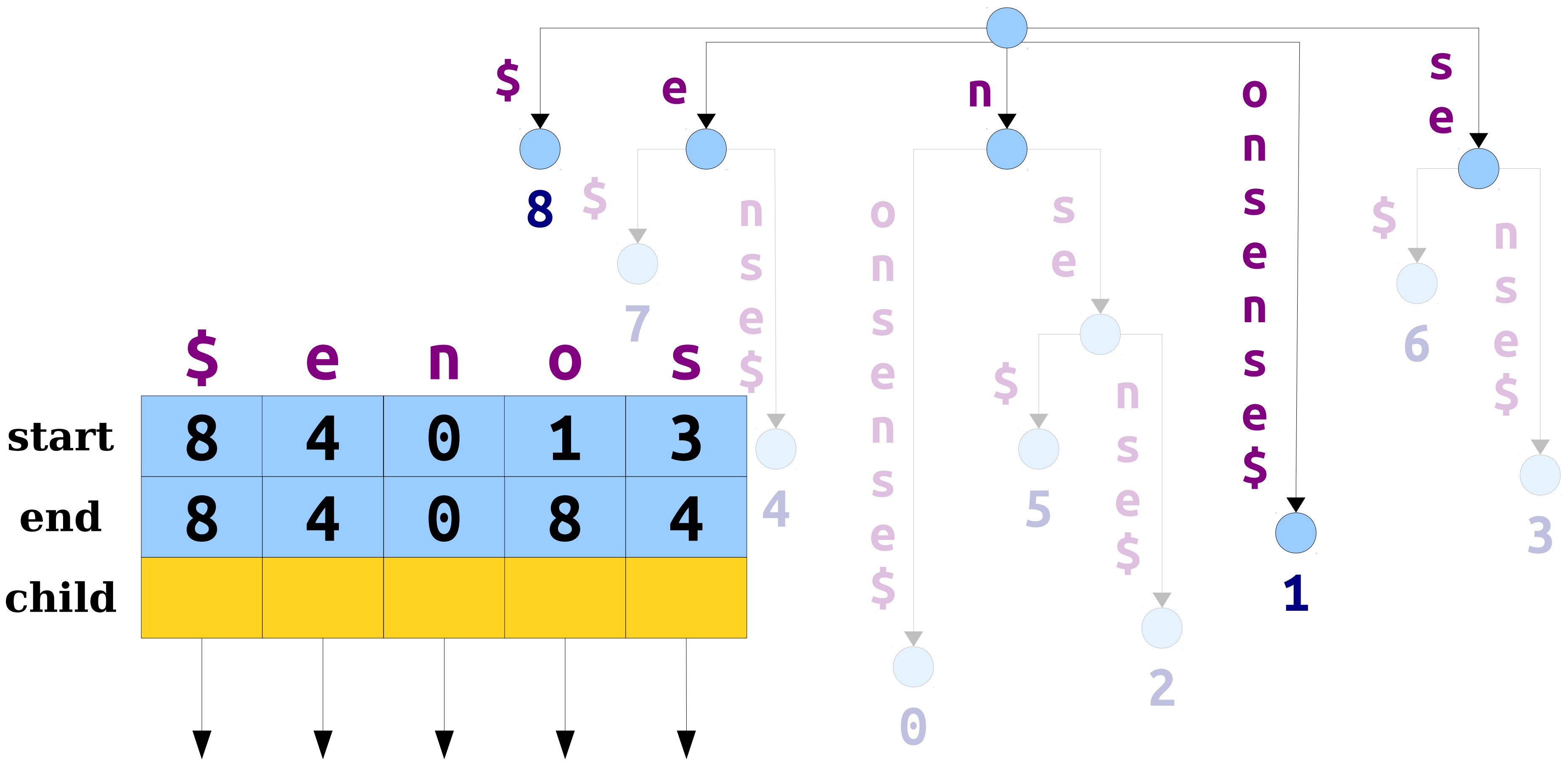
nonsense$
012345678

Suffix Tree Representations



nonsense$
012345678

Suffix Tree Representations



	$	e	n	o	s
start	8	4	0	1	3
end	8	4	0	8	4
child					

nonsense$
012345678

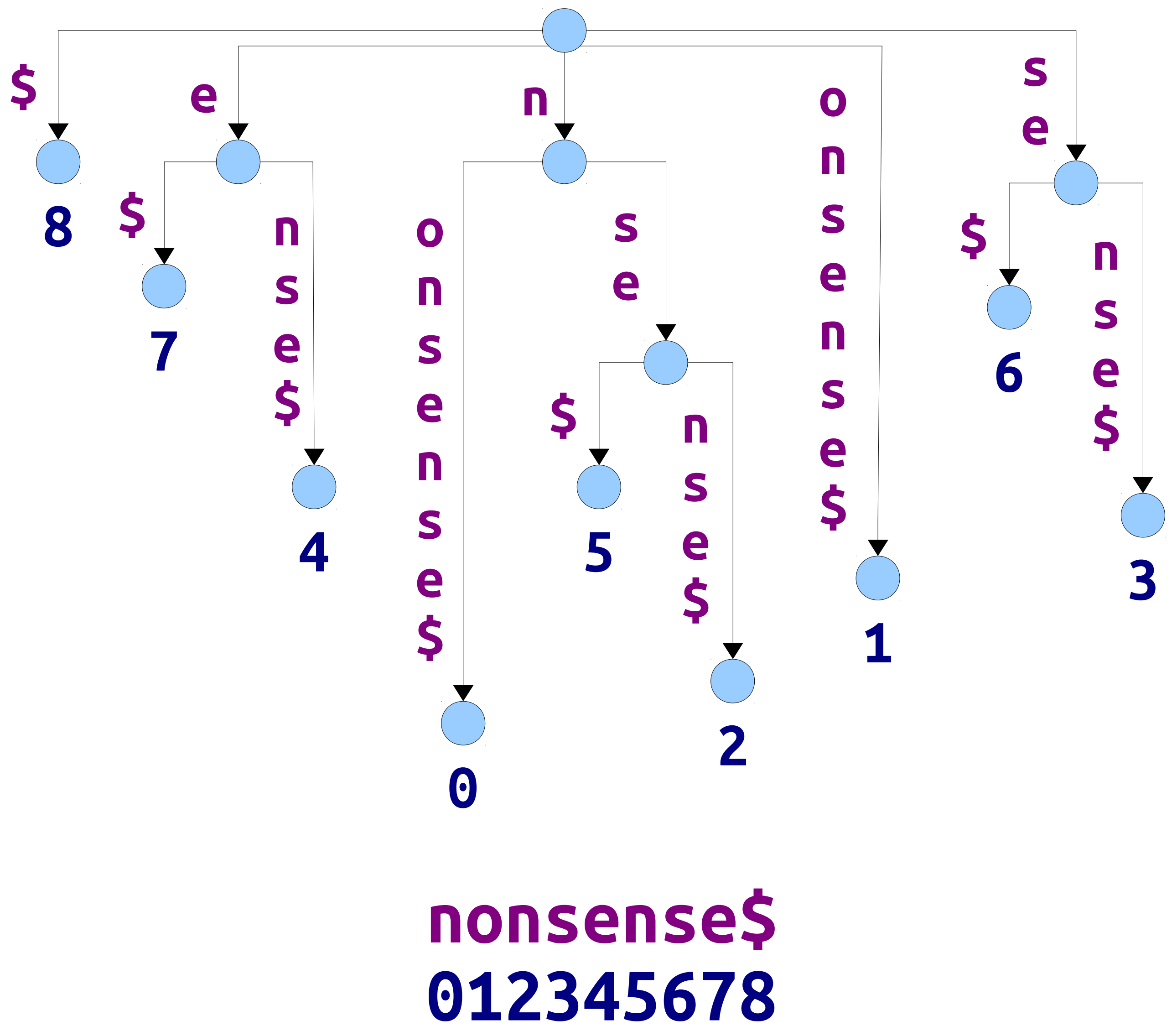
Building Suffix Trees

- ***Claim:*** It's possible to build a suffix tree for a string of length m in time $\Theta(m)$.
- *These algorithms are not trivial!* We'll discuss one of them next time.

Application: String Search

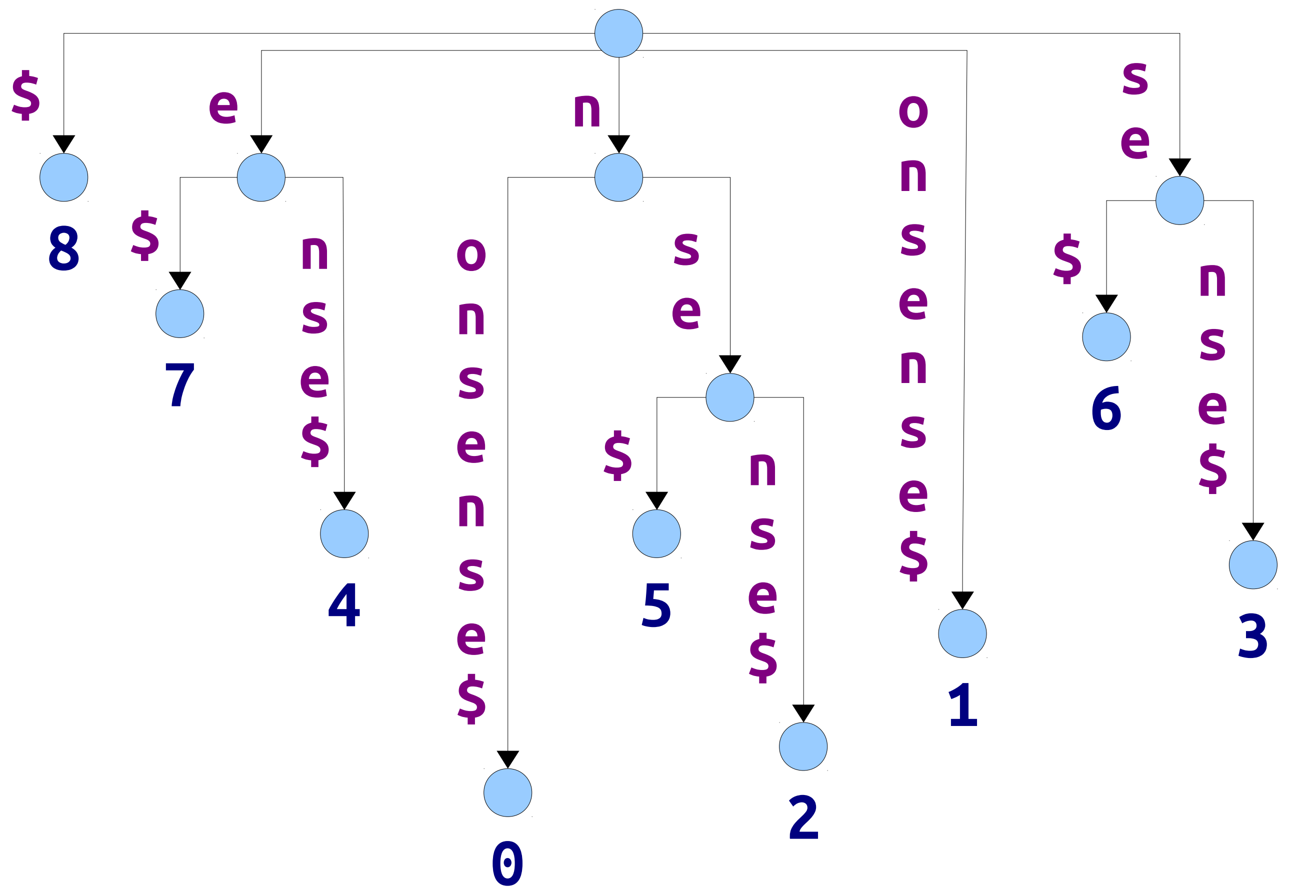
String Matching

- 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.

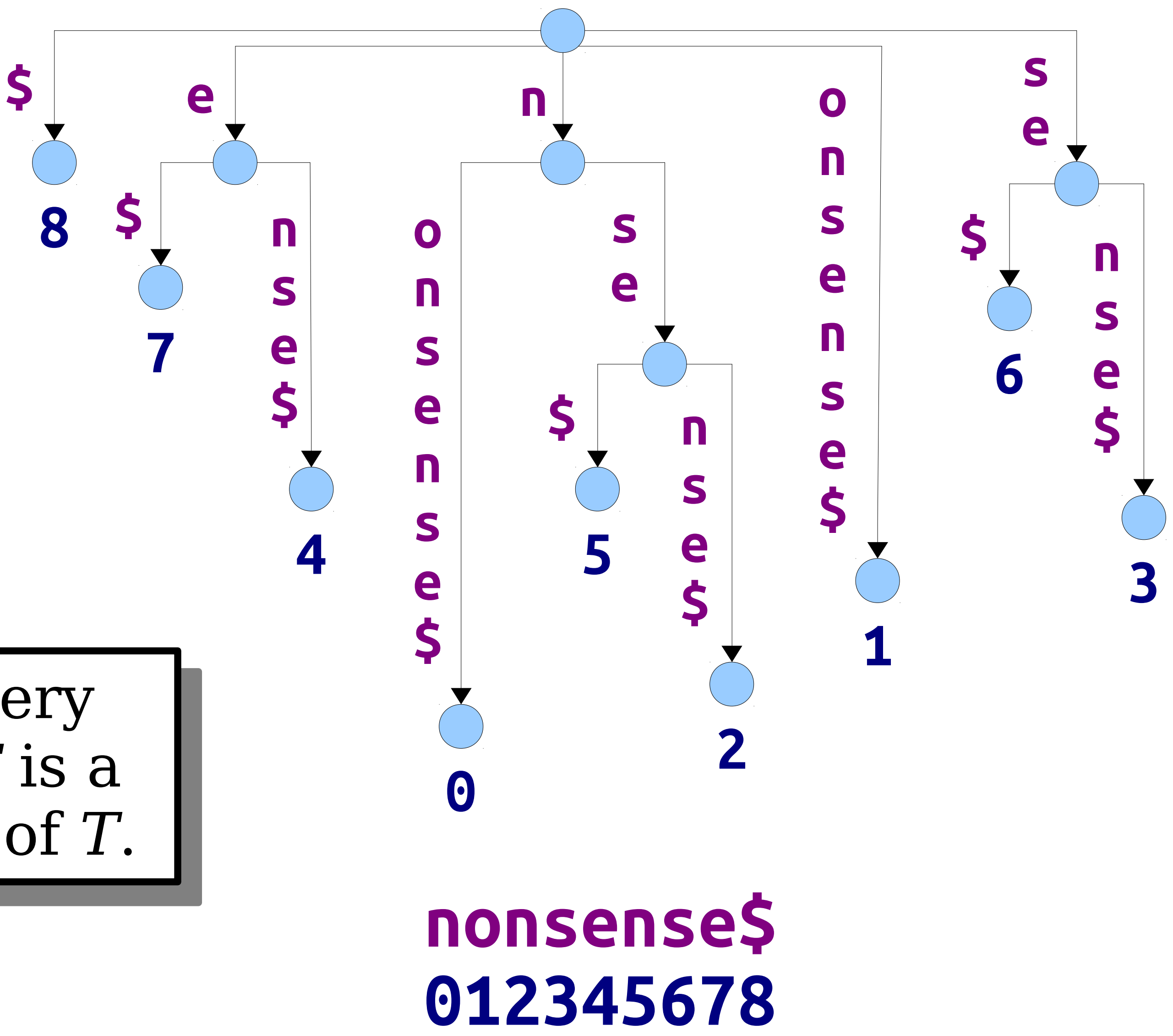


nonsense$
012345678

String Matching

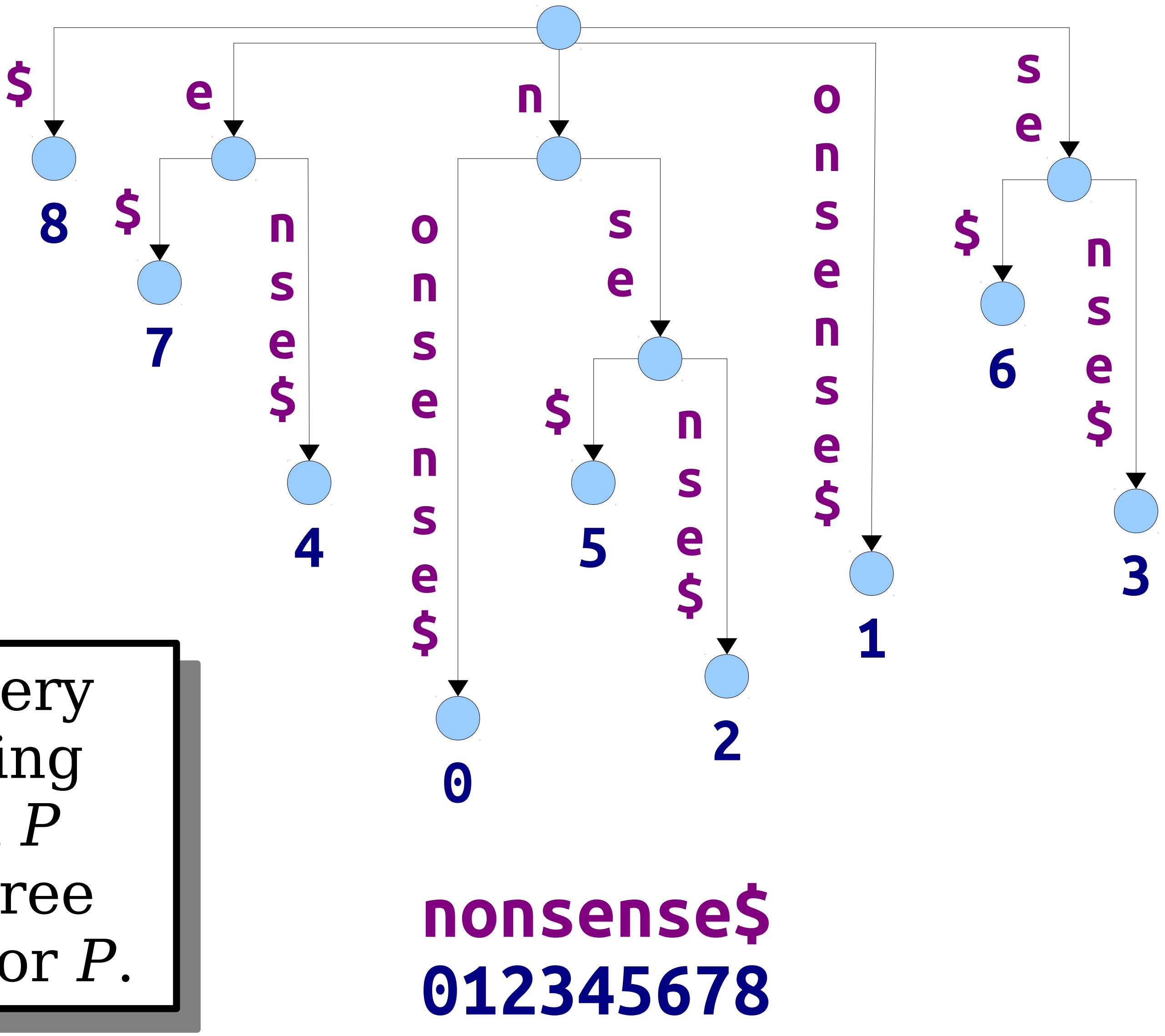
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O($n + z$), where z is the number of matches.

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



String Matching

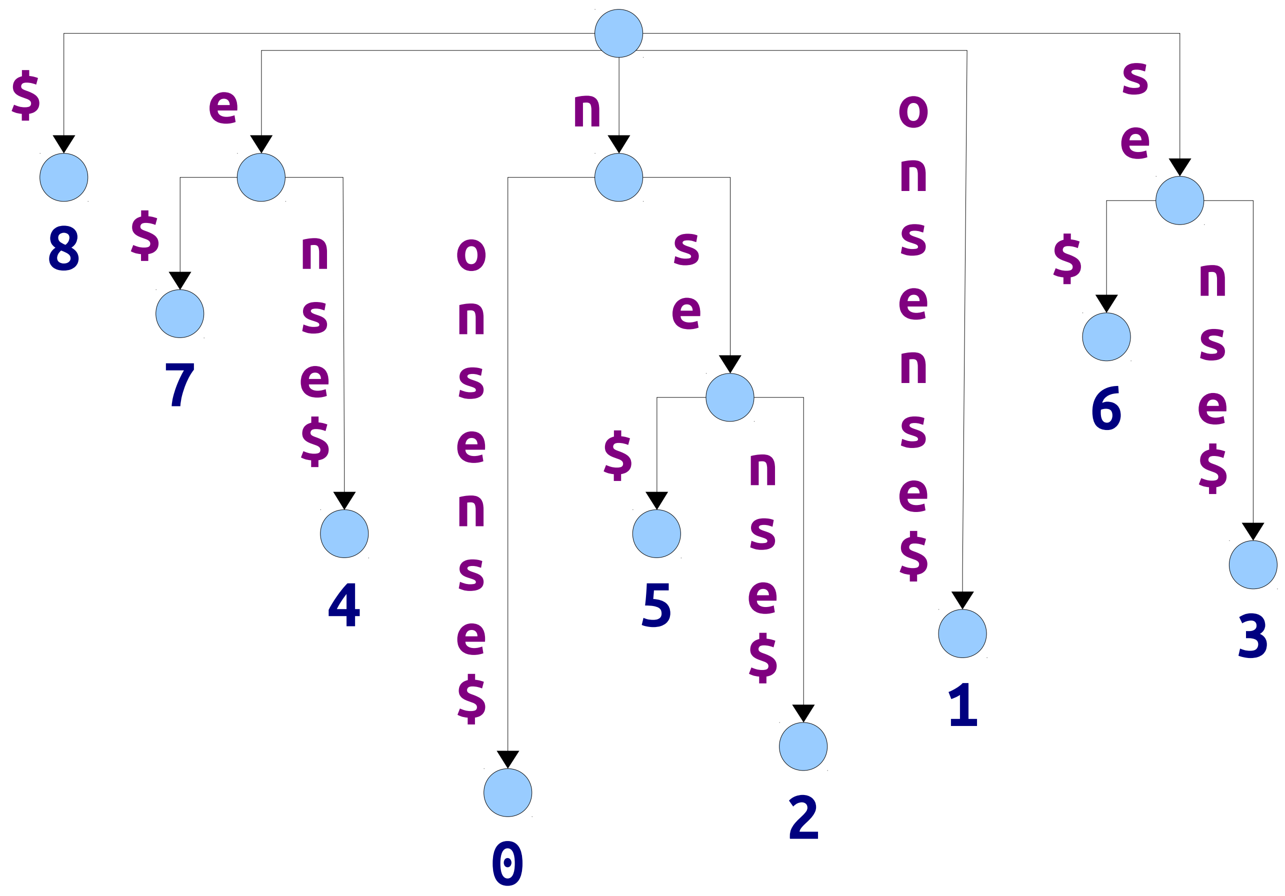
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O($n + z$), where z is the number of matches.



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

String Matching

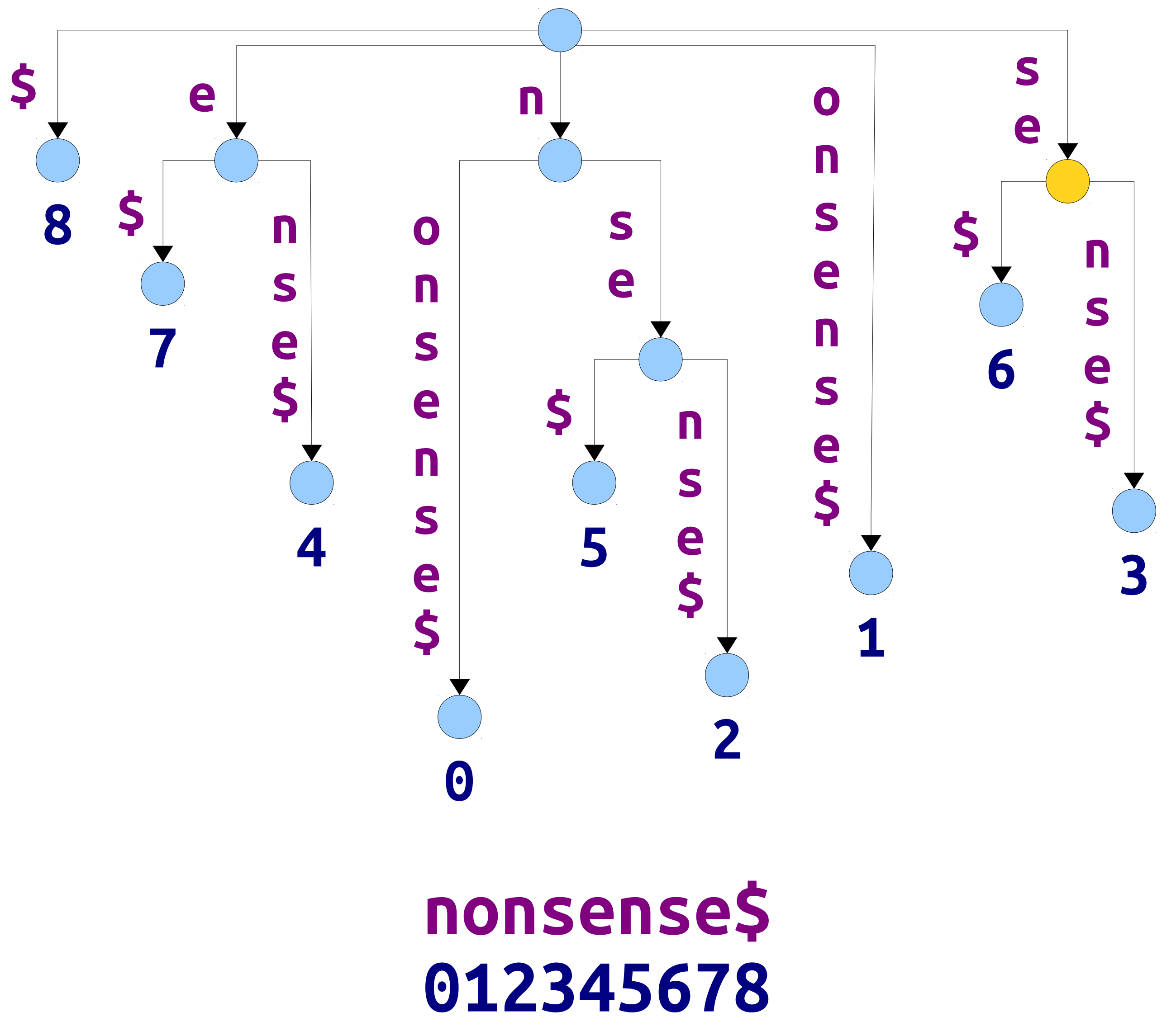
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O(n + z), where z is the number of matches.



nonsense$
012345678

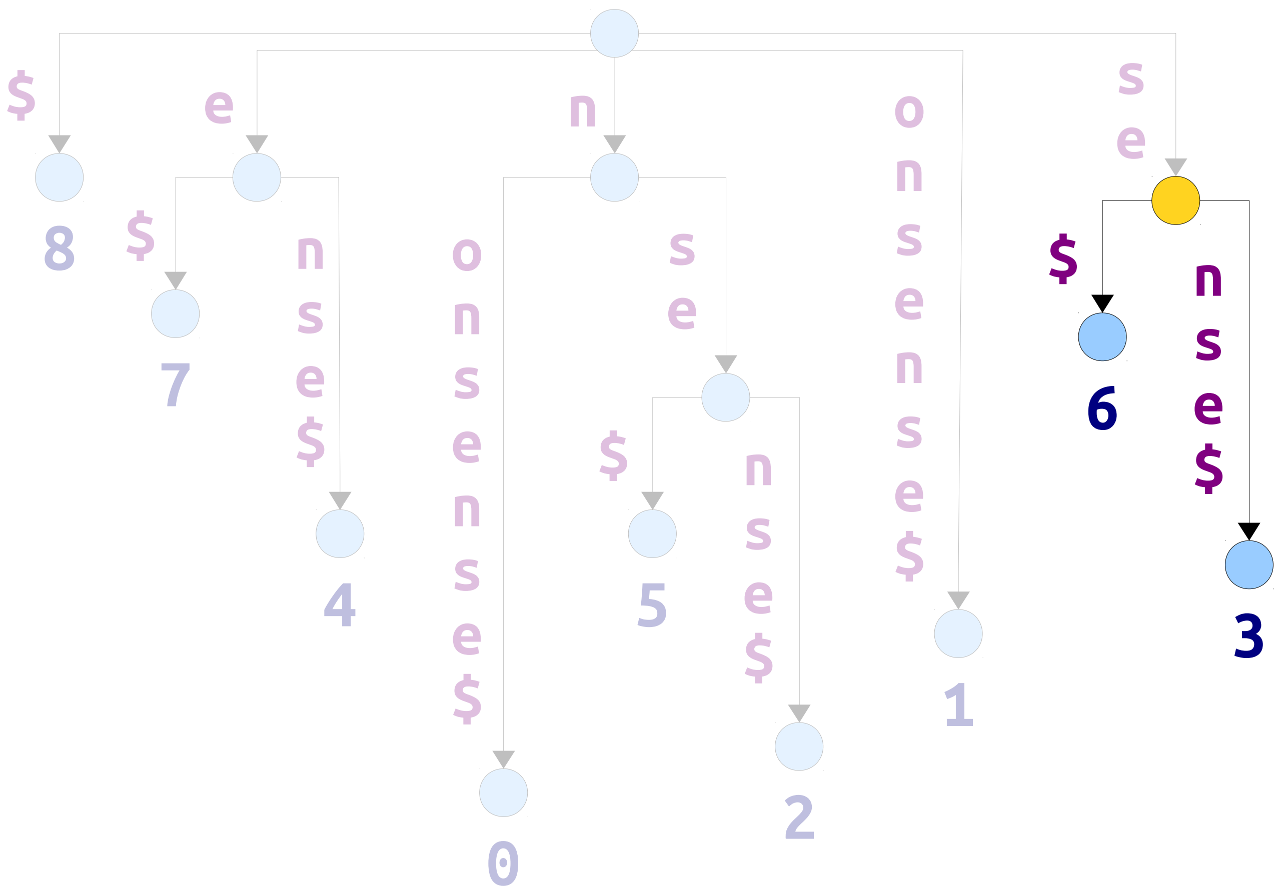
String Matching

- ***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.



String Matching

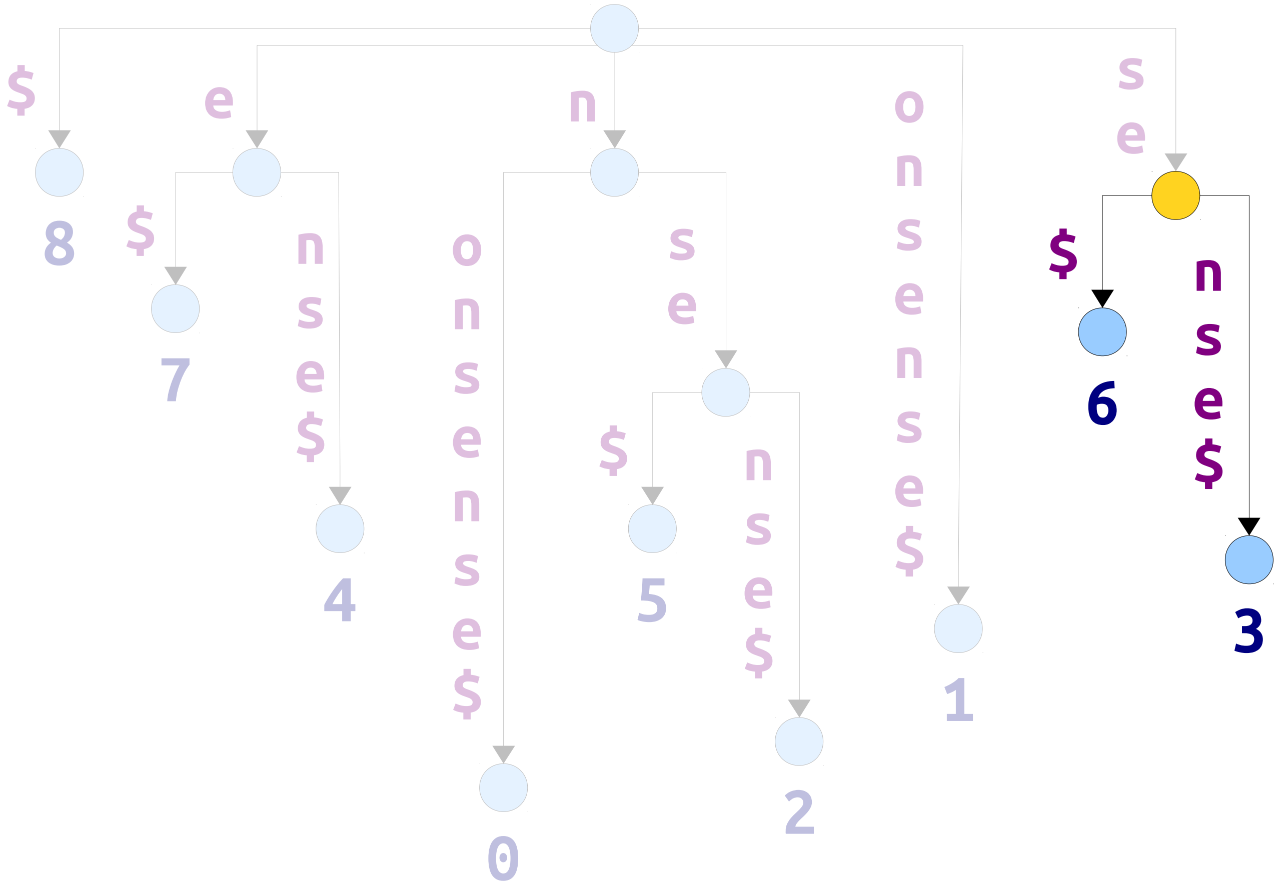
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O($n + z$), where z is the number of matches.



nonsense$
012345678

String Matching

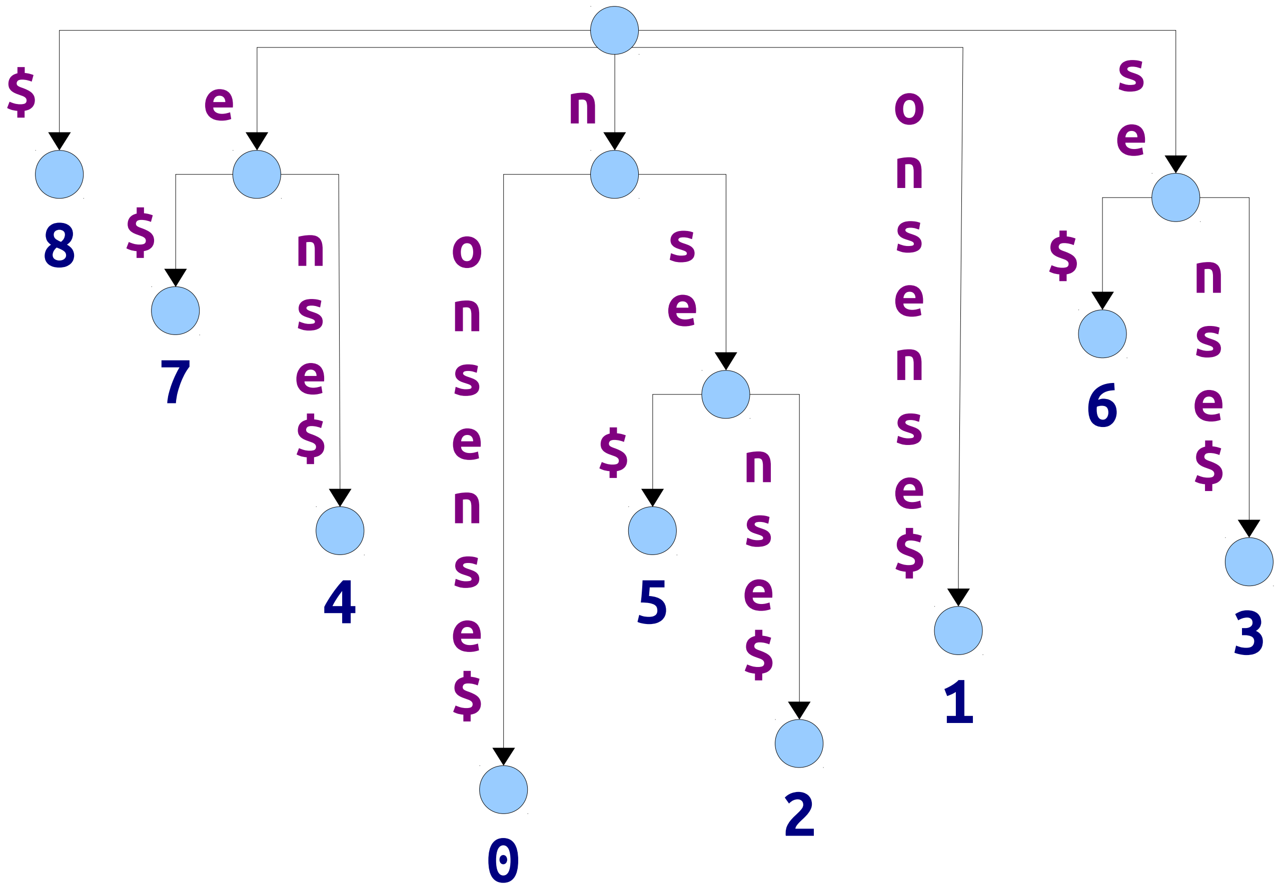
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O(n + z), where z is the number of matches.



nonsense\$
012345678

String Matching

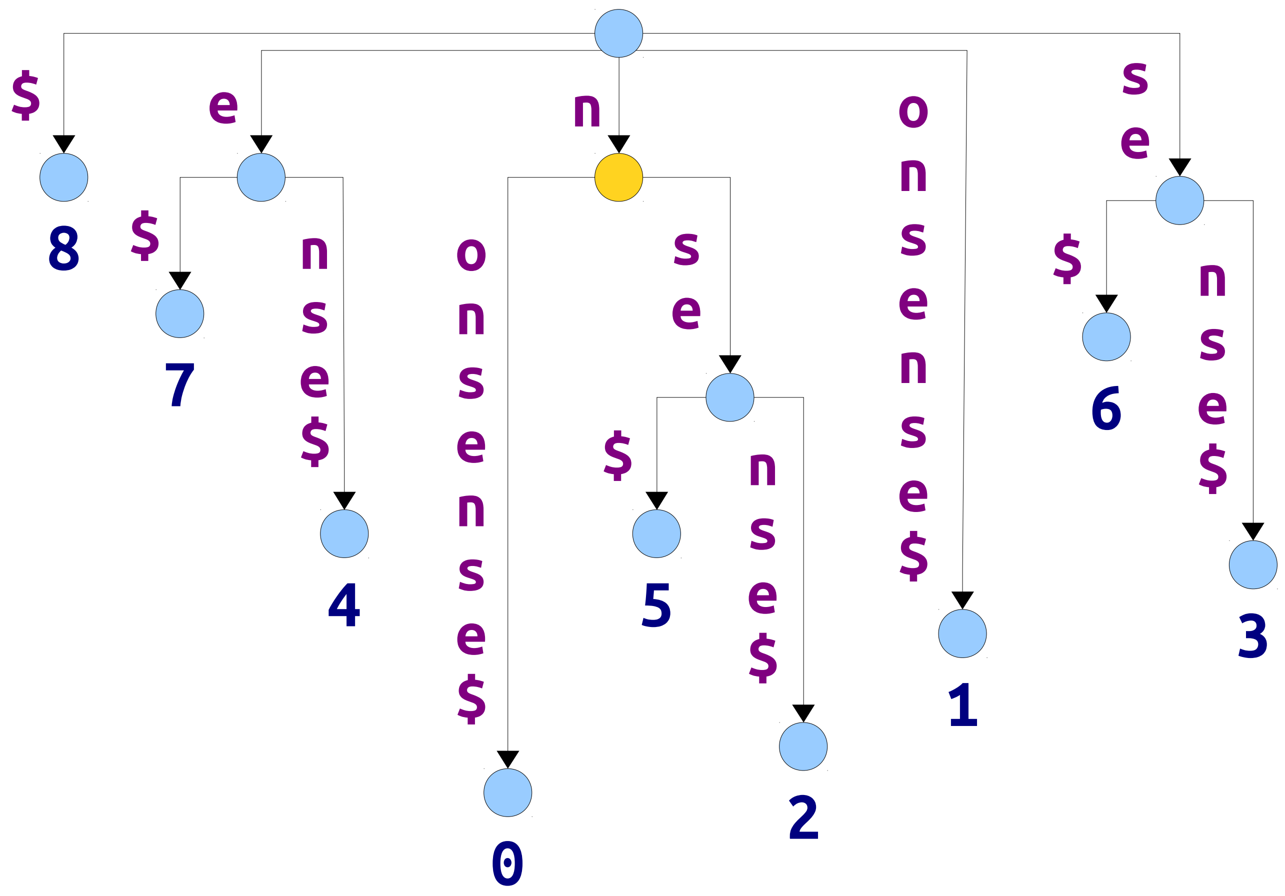
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O($n + z$), where z is the number of matches.



nonsense$
012345678

String Matching

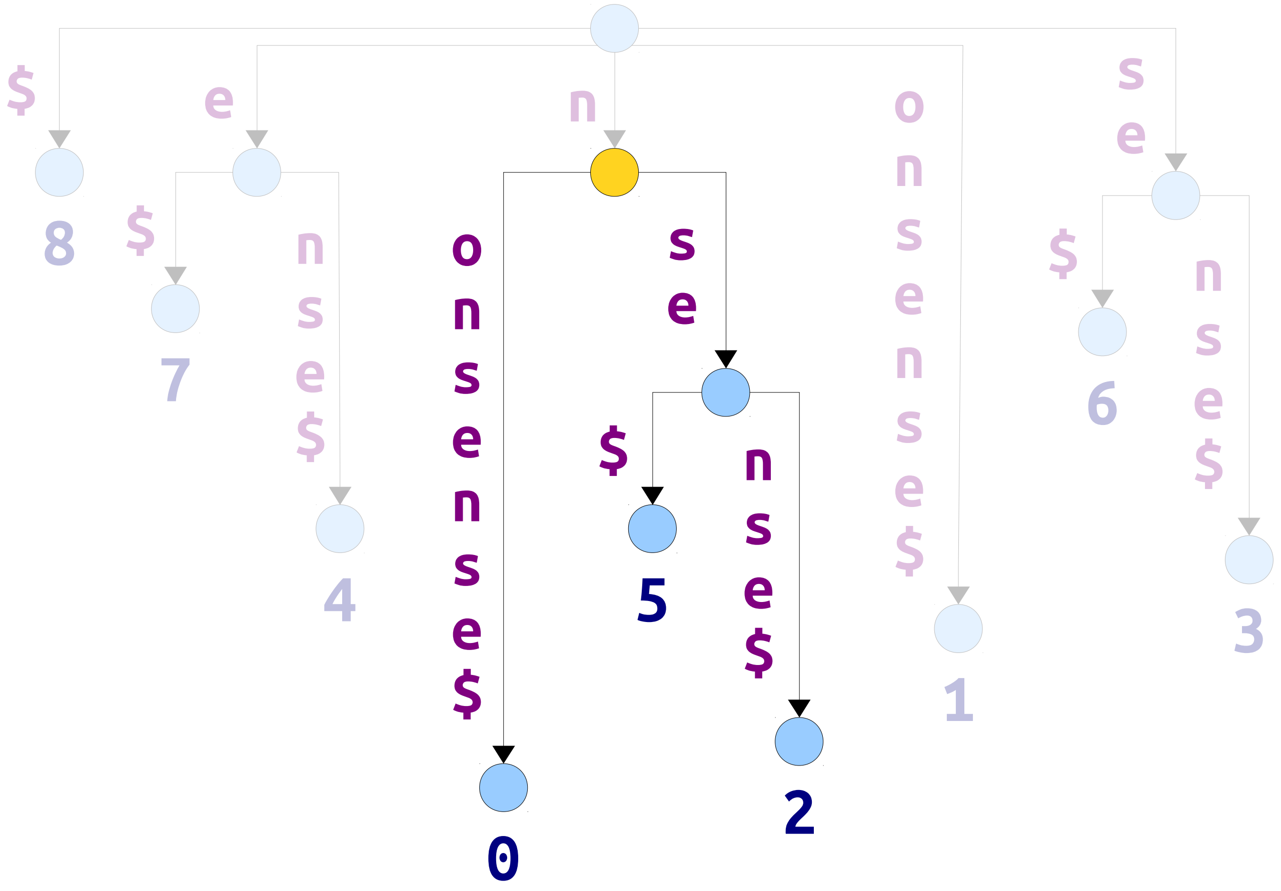
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O($n + z$), where z is the number of matches.



nonsense$
012345678

String Matching

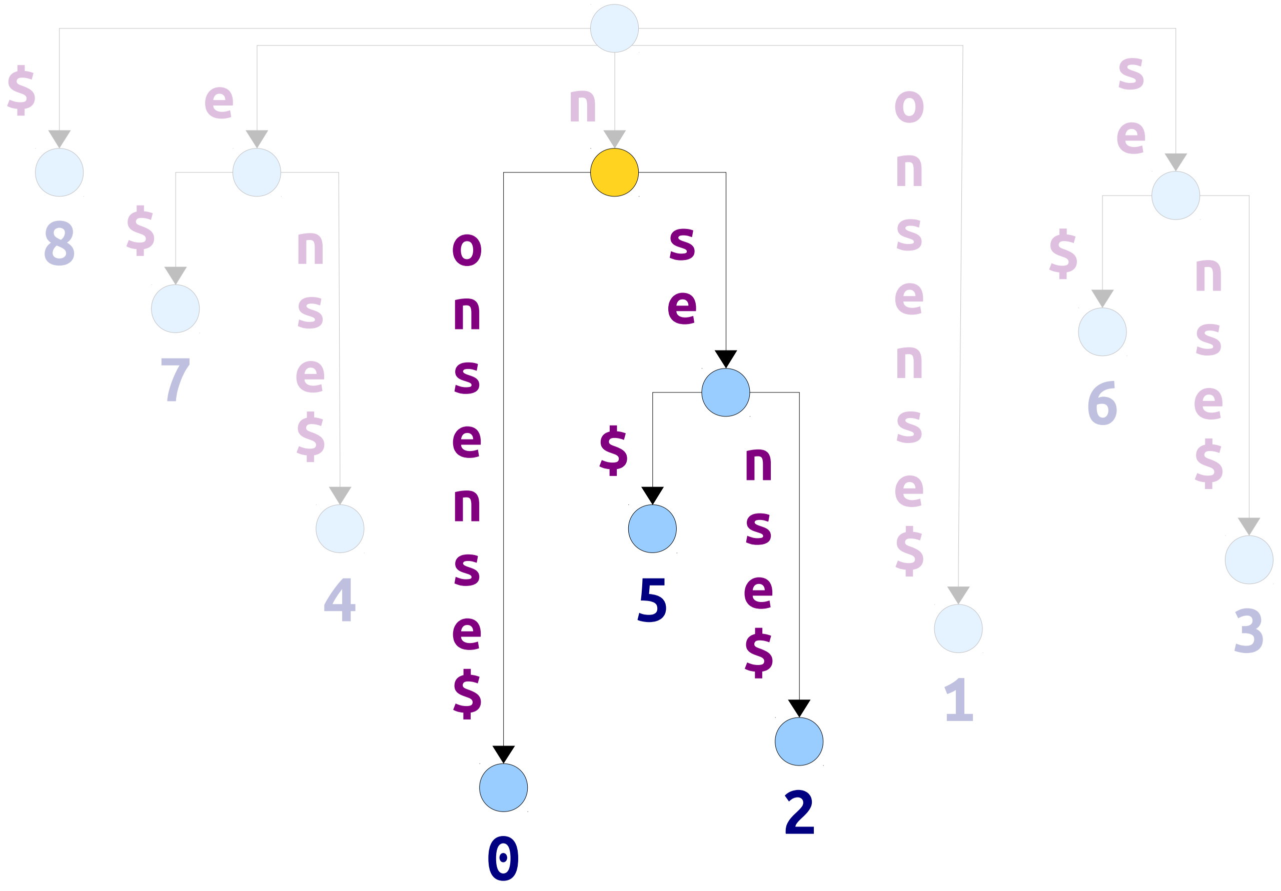
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O(n + z), where z is the number of matches.



nonsense$
012345678

String Matching

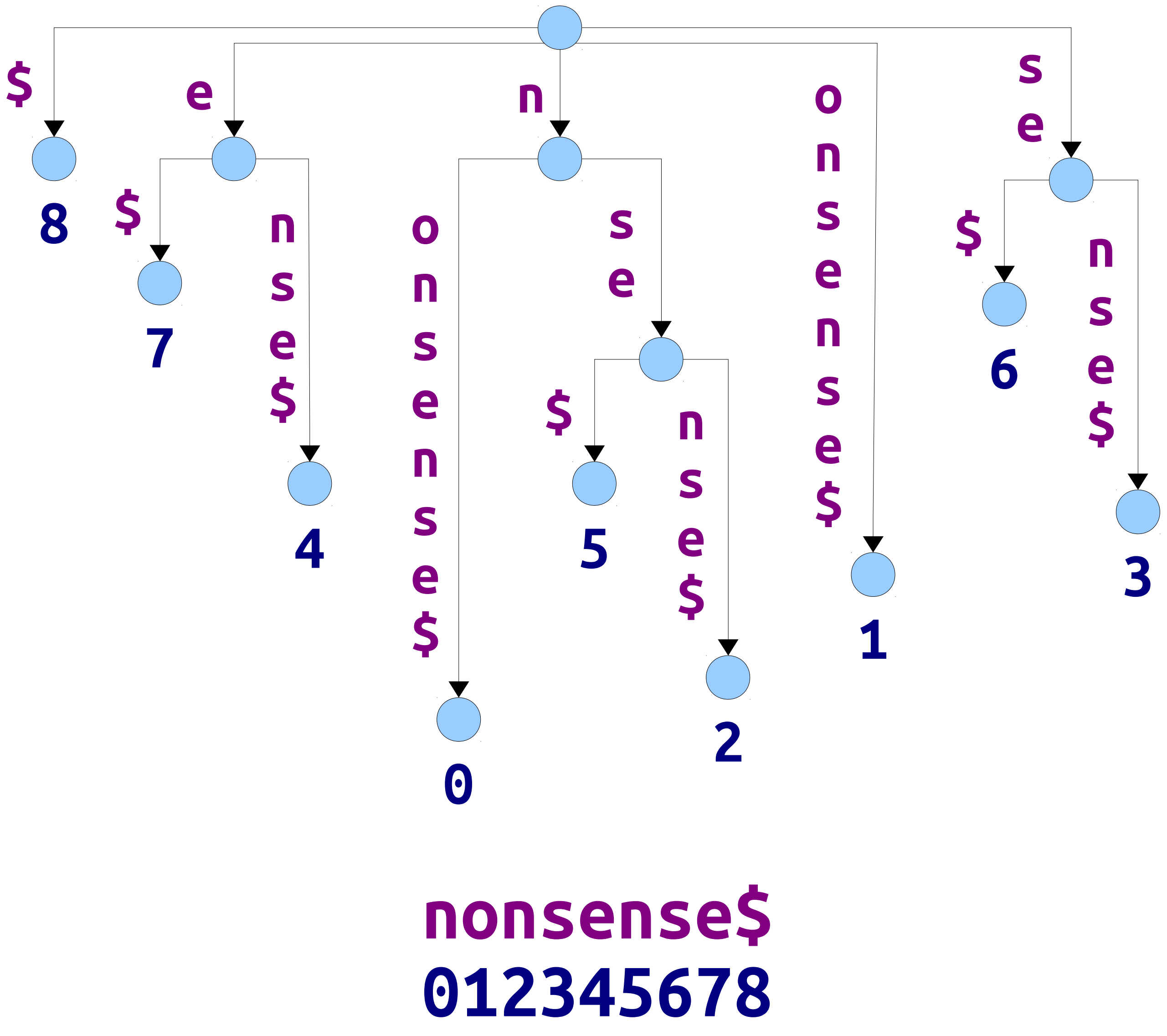
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O(n + z), where z is the number of matches.



nonsense$
012345678

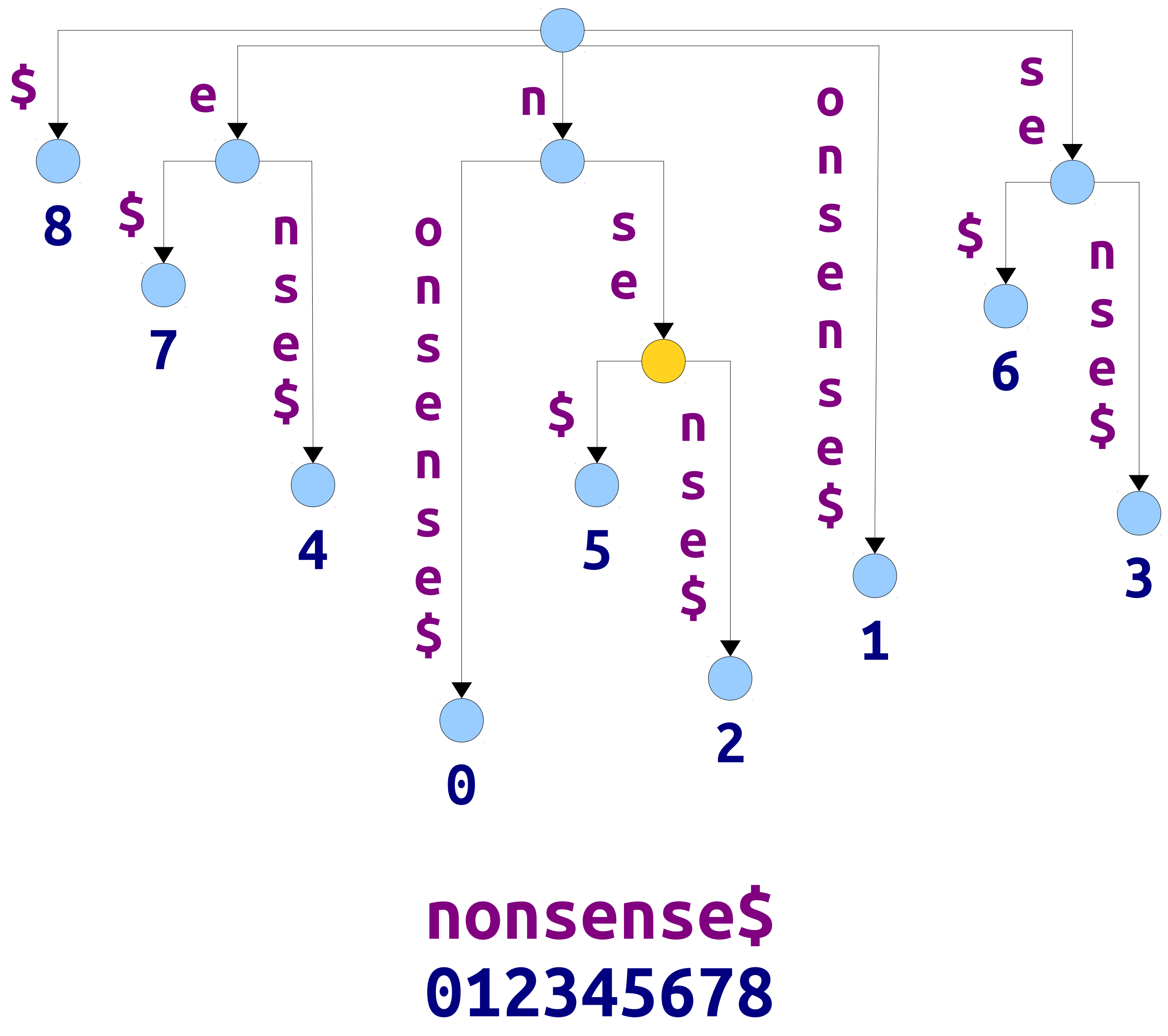
String Matching

- ***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.



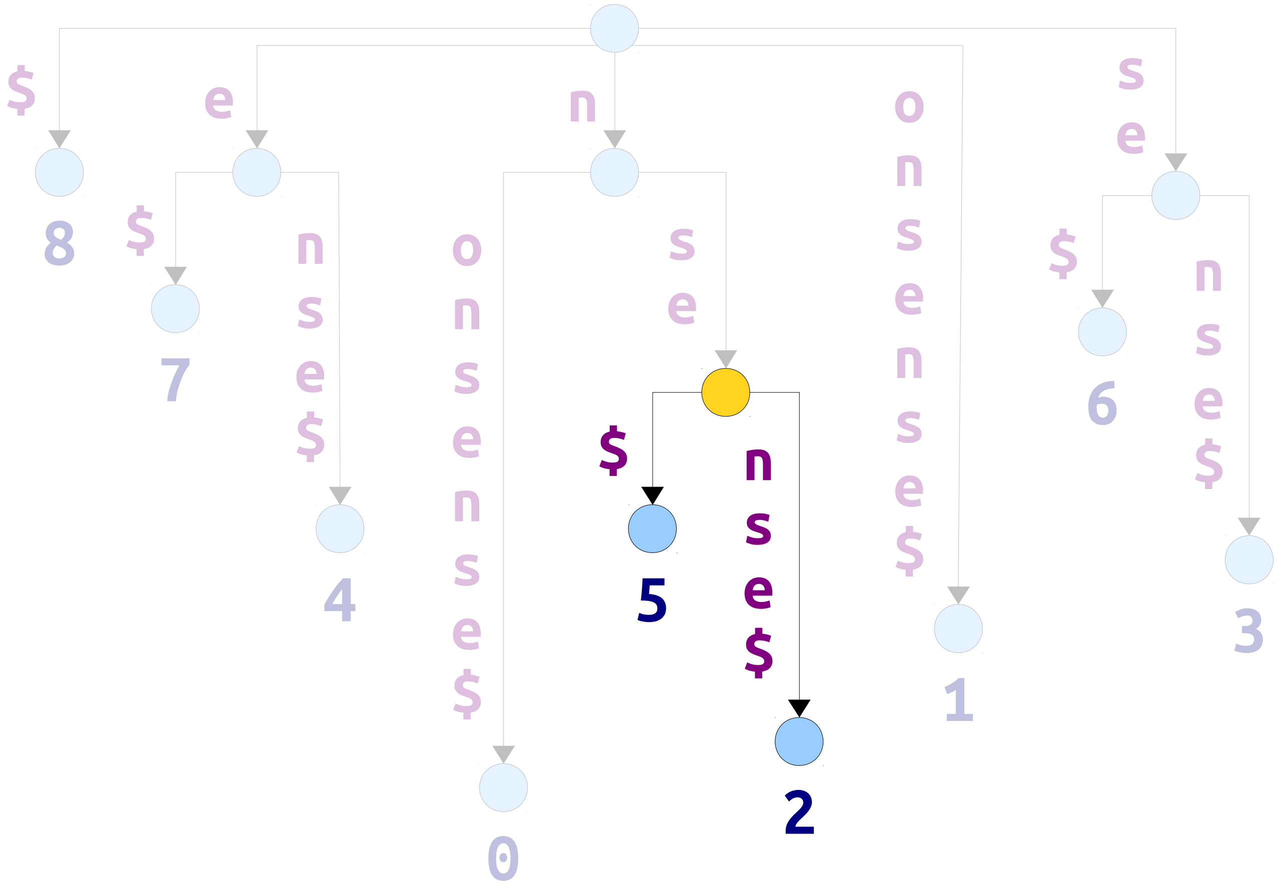
String Matching

- ***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.



String Matching

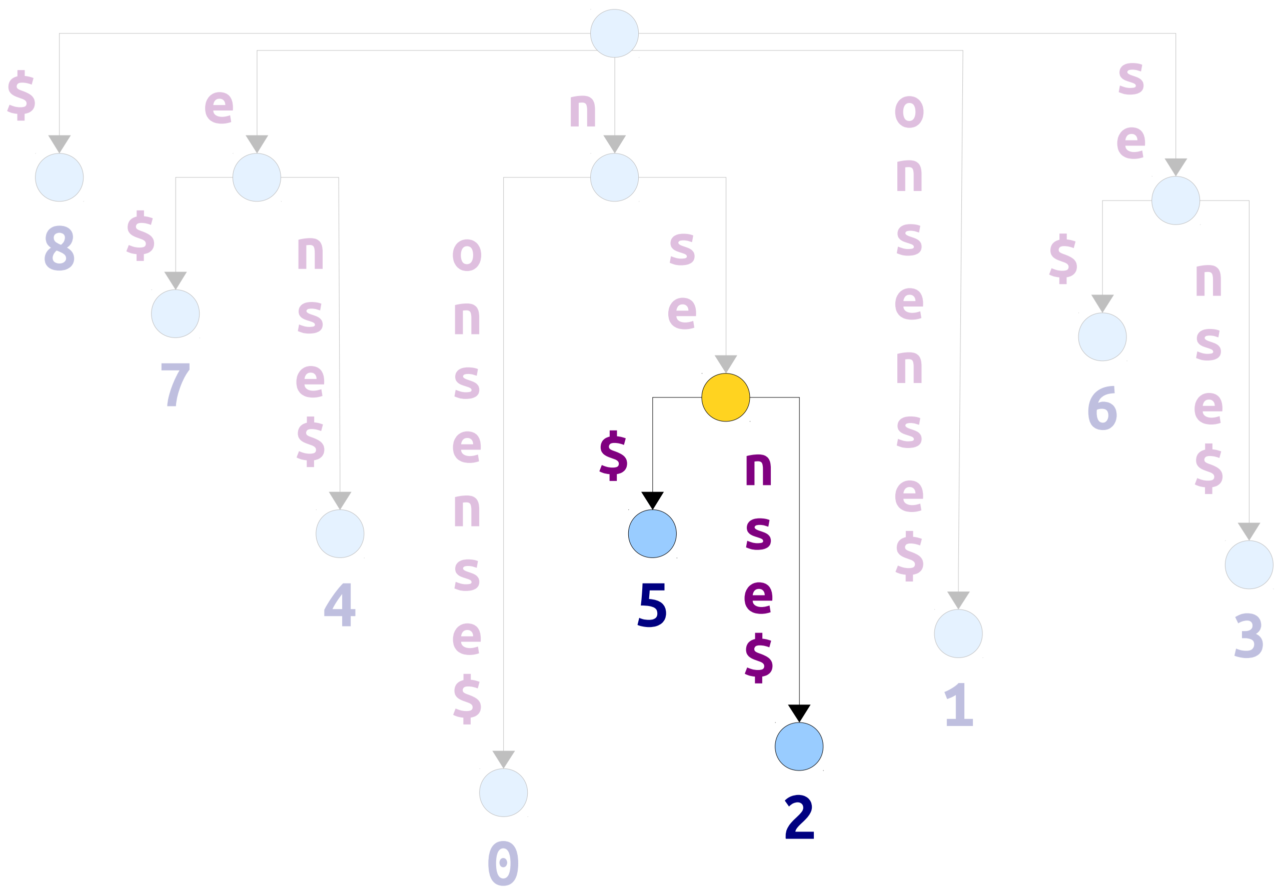
- ***Claim:*** After spending O(m) time preprocessing T\$, can find ***all*** matches of a string P in time O(n + z), where z is the number of matches.



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, … P_k of total length n, suffix trees can find all matches of those patterns in time $O(m + n + z)$.
    - Build the tree in time $O(m)$, then 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

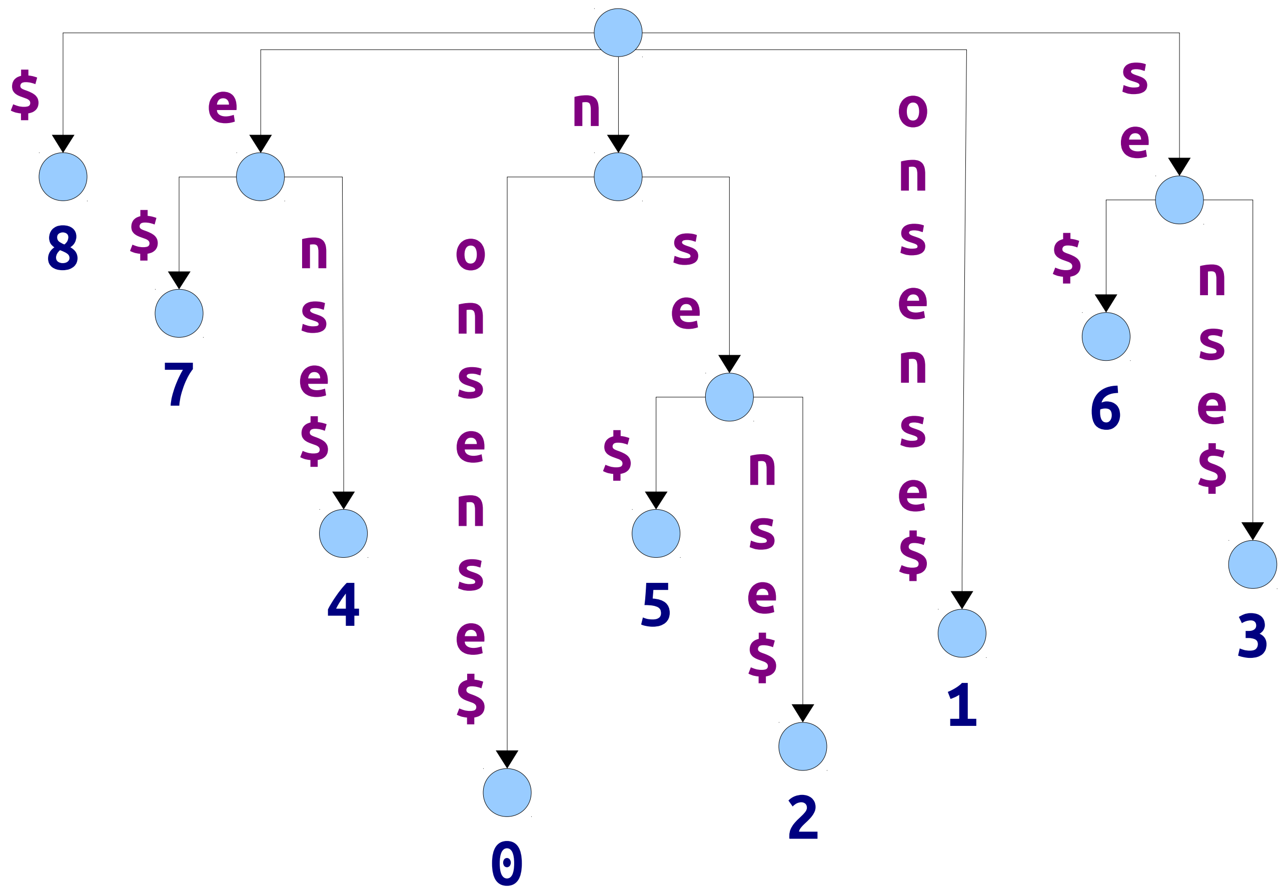
Longest Repeated Substring

- Consider the following problem:

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

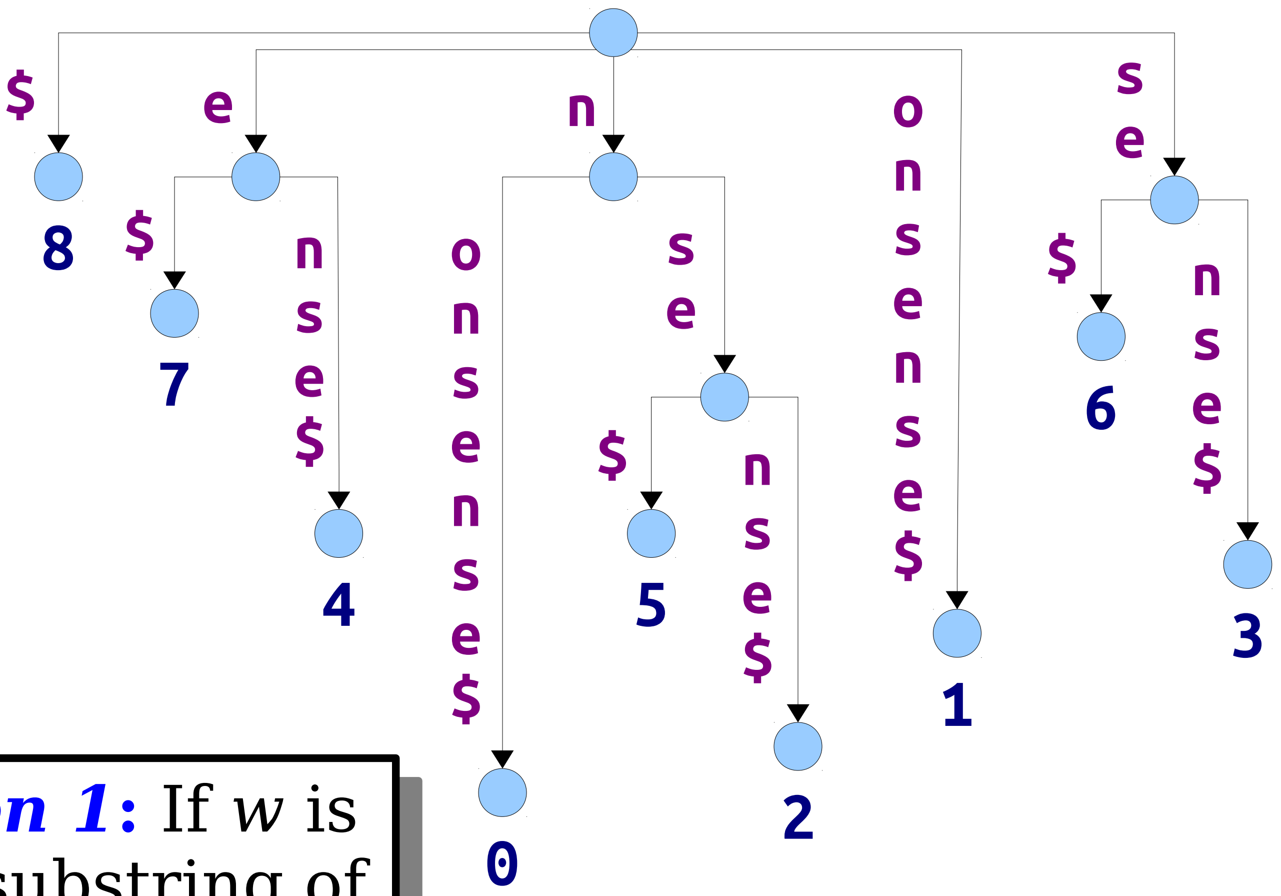
- Some examples:
  - In `monsoon`, the longest repeated substring is `on`.
  - In `banana`, the longest repeated substring is `ana`. (The substrings can overlap.)
- Applications to computational biology: more than half of the human genome is formed from repeated DNA sequences!

Longest Repeated Substring



nonsense$
012345678

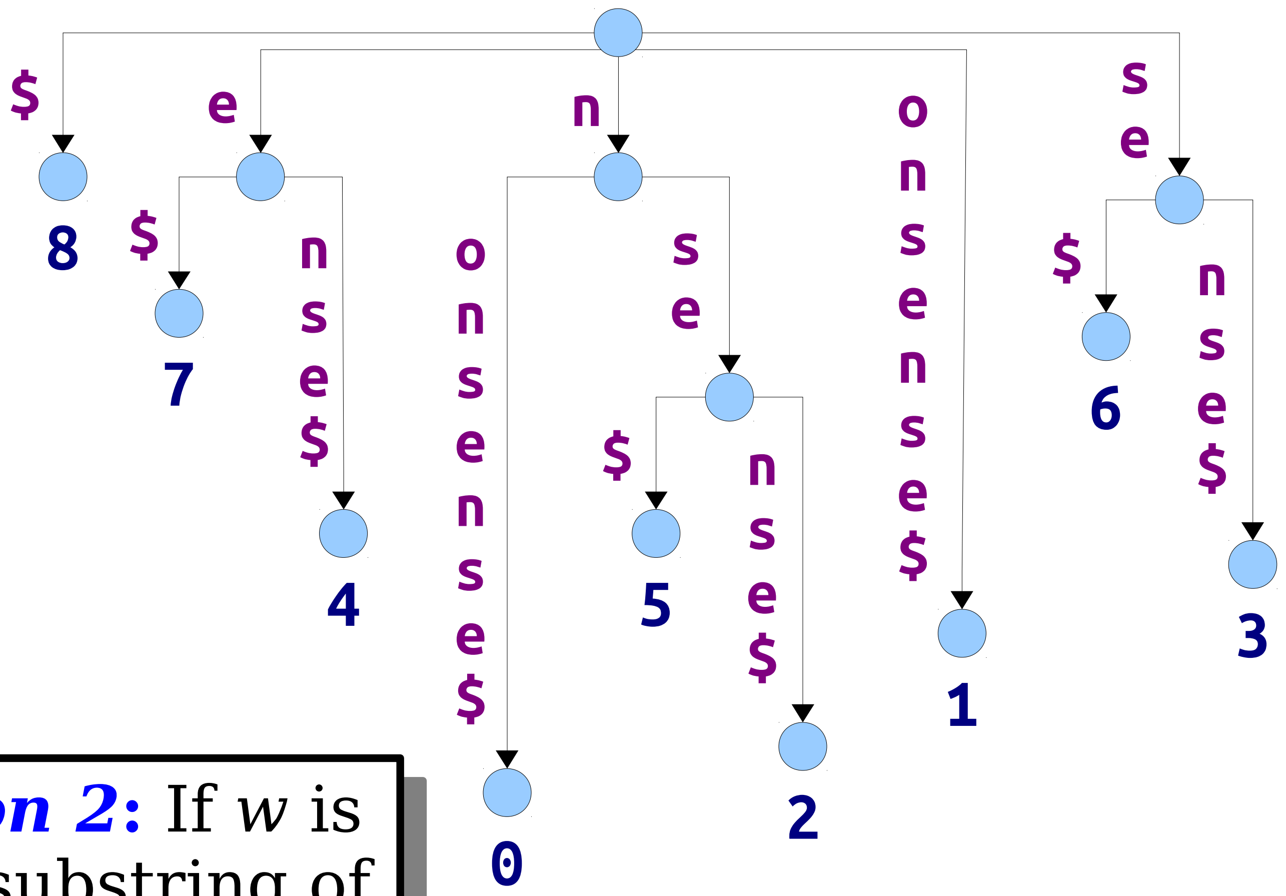
Longest Repeated Substring



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

nonsense$
012345678

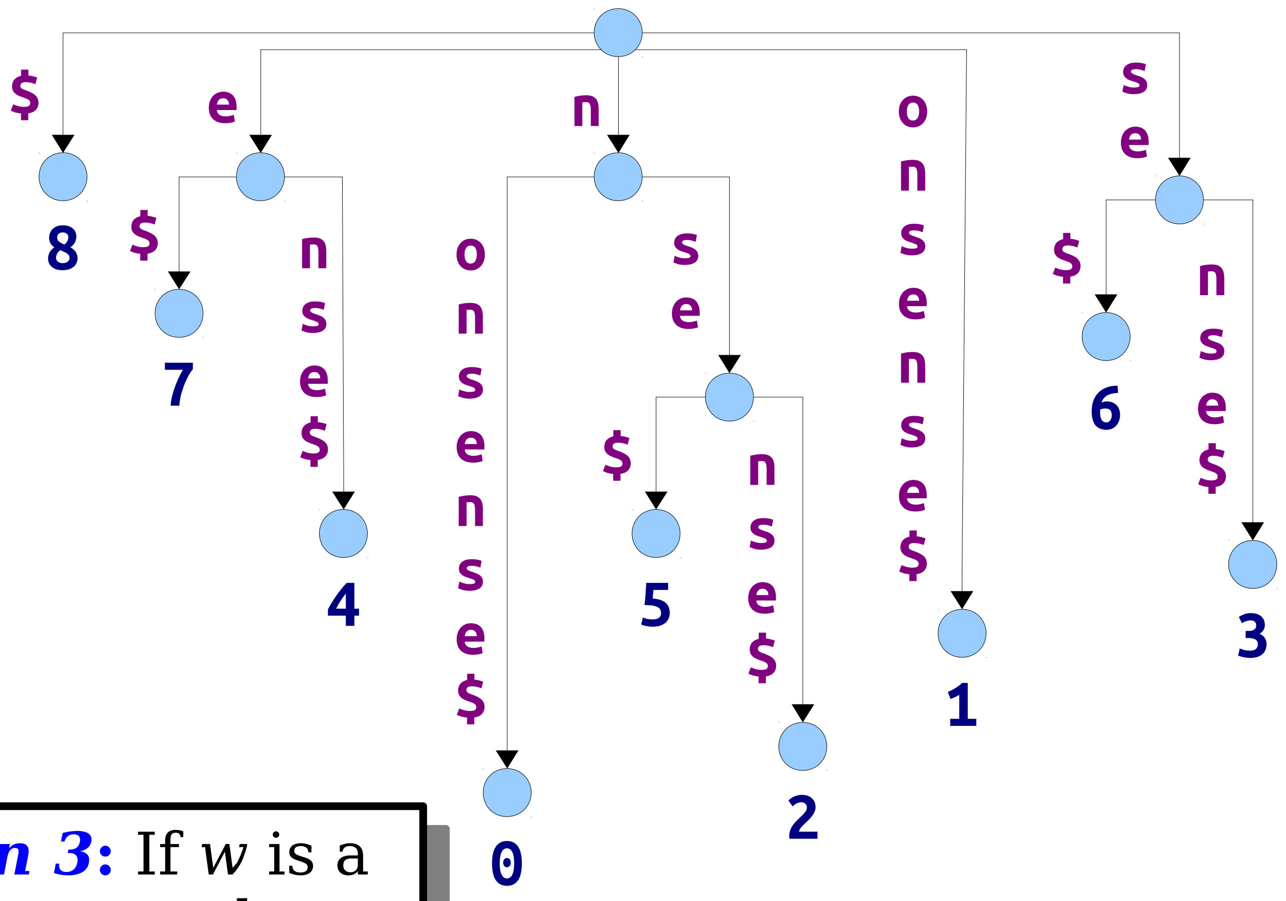
Longest Repeated Substring



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

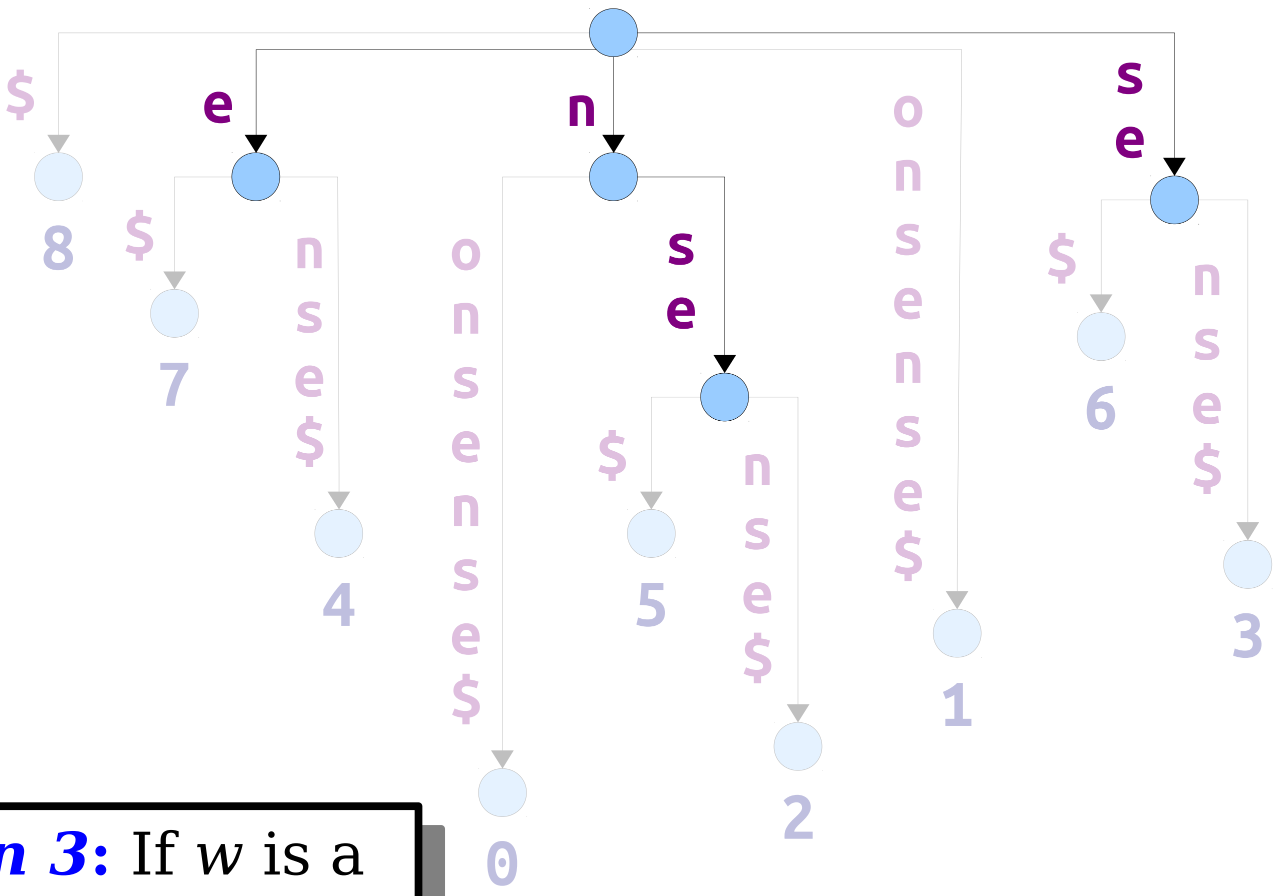
Longest Repeated Substring



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 the suffix tree, tracking the string depth as you go, to find the internal node of maximum string depth.
  - Recover the string that node 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 Sets

- Problem Set 0 solutions will be up on the course website later today.
  - We’ll try to get it graded and returned as soon as possible.
- Problem Set 1 is due on Tuesday at 2:30PM.
  - Stop by office hours with questions!
  - Ask questions on Piazza!

WICS PRESENTS

DISTINGUISHED SPEAKER SERIES

FEATURING

TRACY YOUNG

Come hear from co-founders of Plangrid, Tracy Young and Ralph Gootee, about their journey building Plangrid, a company that creates software for the $8 trillion a year construction industry. Dinner will be provided.

04.17.18
6:30-8:00 PM
HEWLETT 102

PlanGrid

Back to CS166!

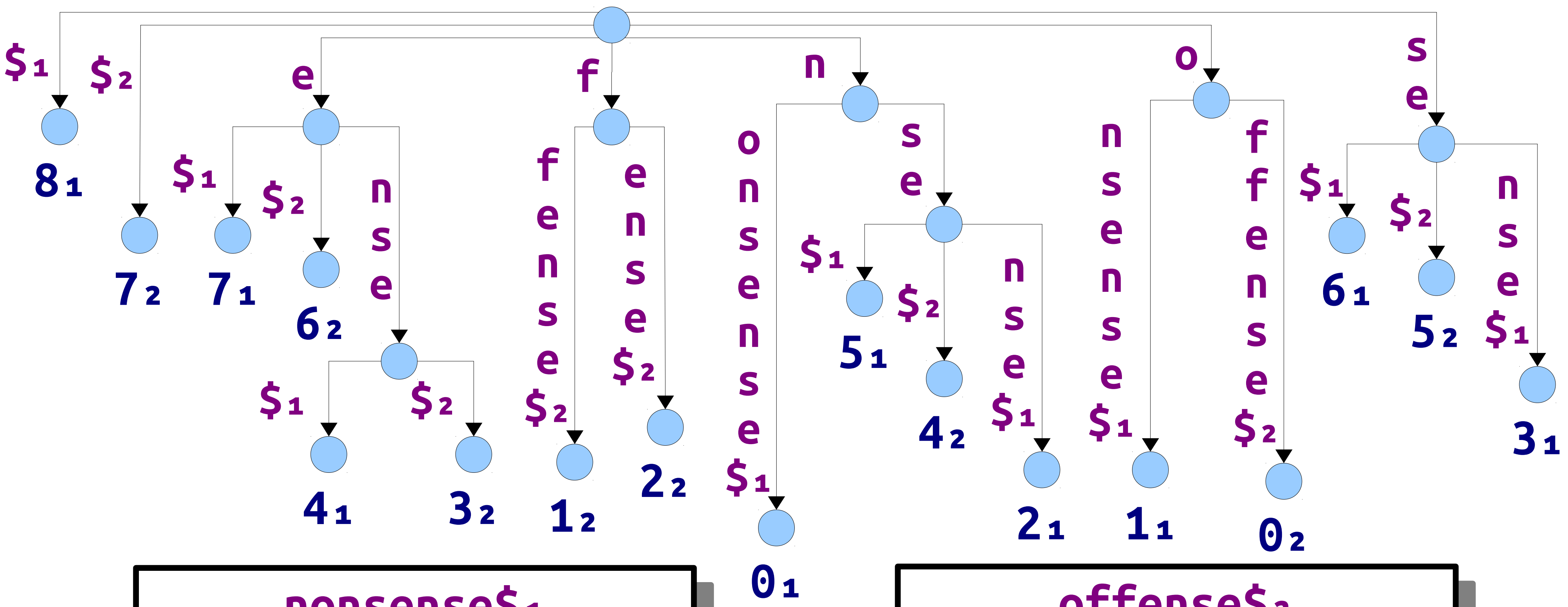
Generalized Suffix Trees

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 $T_1, \ldots, T_k$ is a Patricia trie of all suffixes of $T_1\$_1, \ldots, T_k\$_k$. Each T_i has a unique end marker.
- Leaves are tagged with $\boldsymbol{i_j}$, meaning "ith suffix of string T_j"



nonsense$\$_1$
012345678

offense$\$_2$
01234567

Generalized Suffix Trees

- ***Claim:*** A generalized suffix tree for strings $T_1, \ldots, T_k$ of total length m can be constructed in time $\Theta(m)$.
- Use a two-phase algorithm:
  - Construct a suffix tree for the single string $T_1\$_1T_2\$_2 \ldots T_k\$_k$ 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

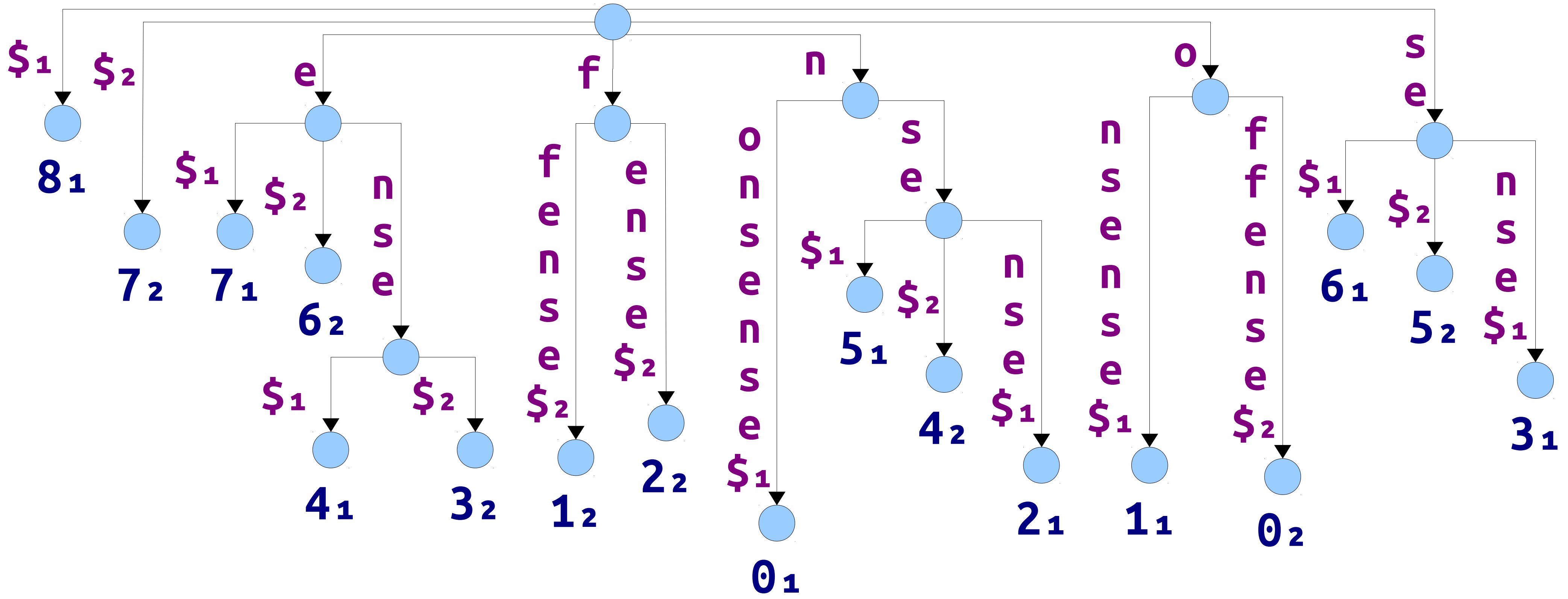
Longest Common Substring

- 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

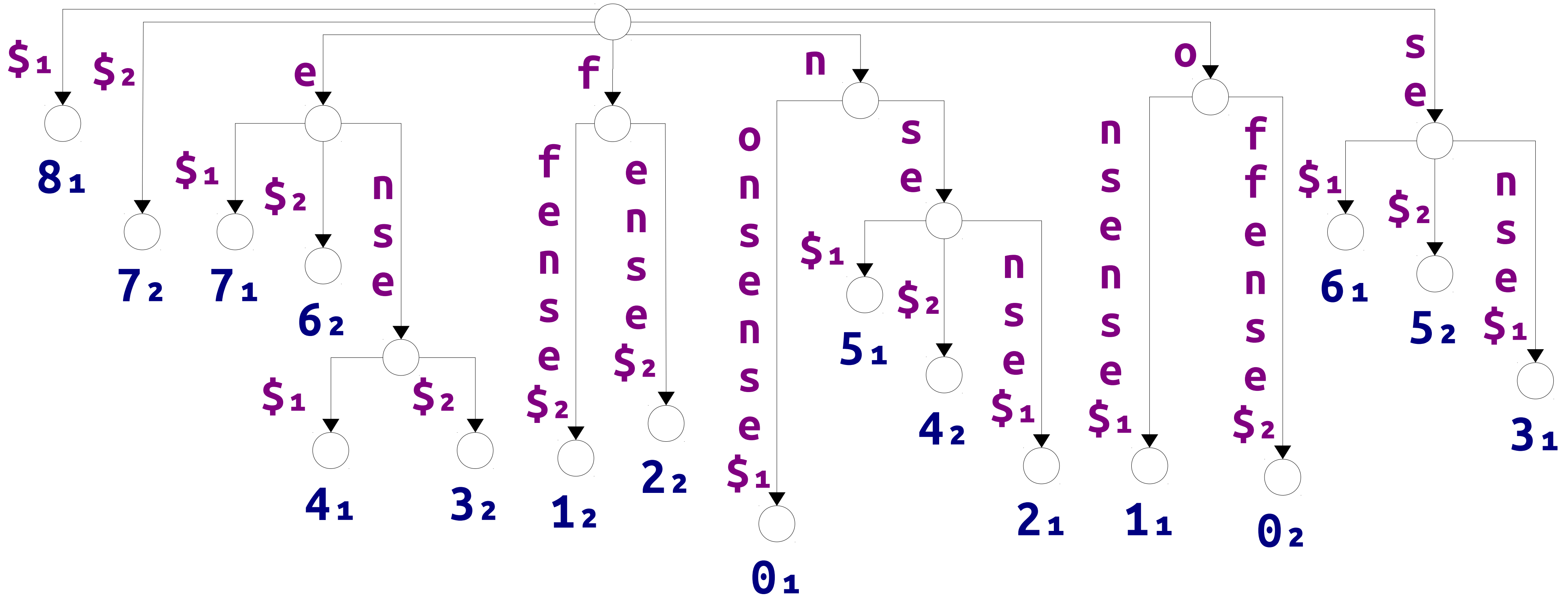


Observation: Any common substring of T_1 and T_2 will be a prefix of a suffix of T_1 and a prefix of a suffix of T_2.

```
nonsense$₁
012345678
```

```
offense$₂
01234567
```

Longest Common Substring

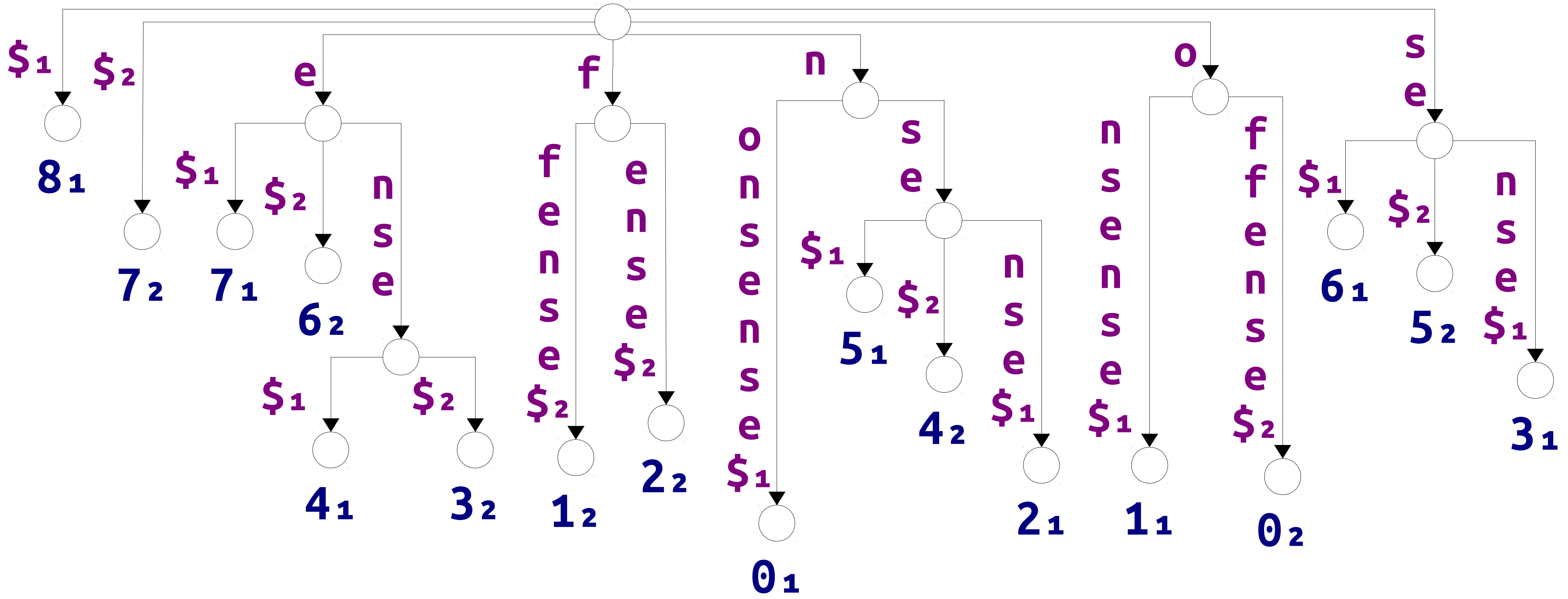


Observation: Any common substring of T_1 and T_2 will be a prefix of a suffix of T_1 and a prefix of a suffix of T_2.

```
nonsense$₁
012345678
```

```
offense$₂
01234567
```

Longest Common Substring

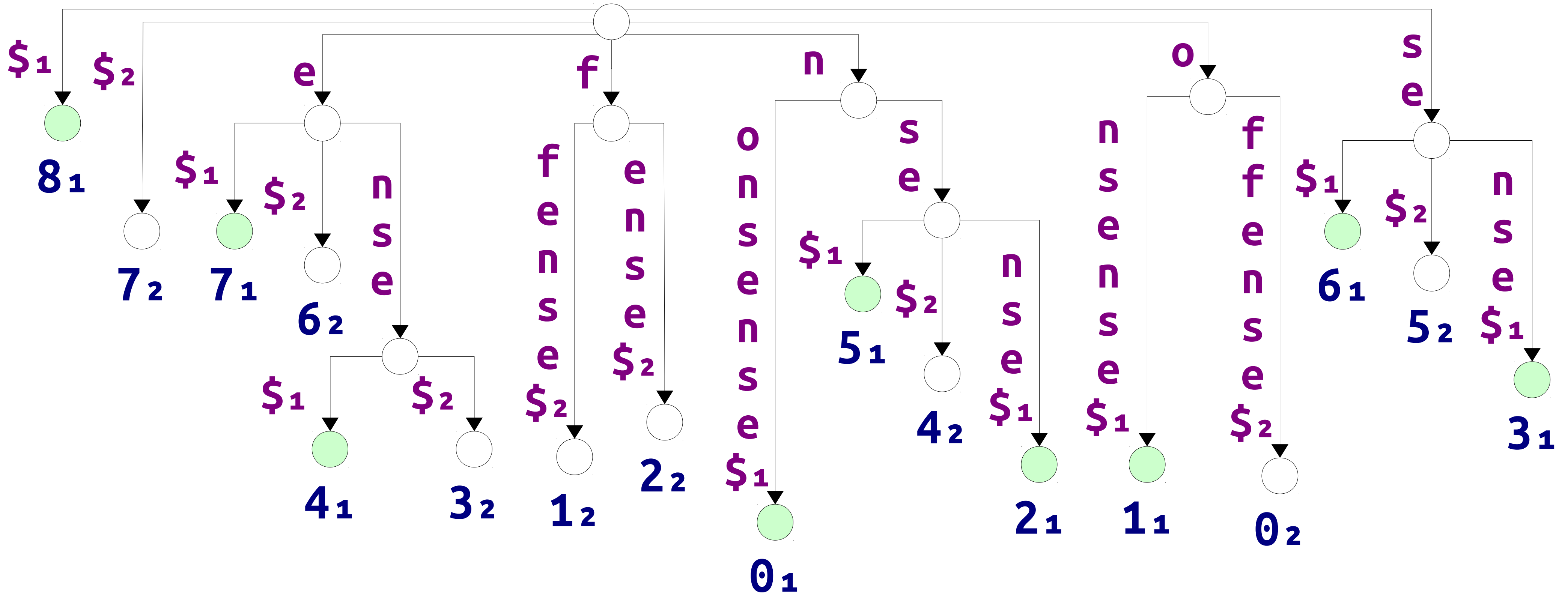


nonsense$₁
012345678

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.

offense$₂
01234567

Longest Common Substring

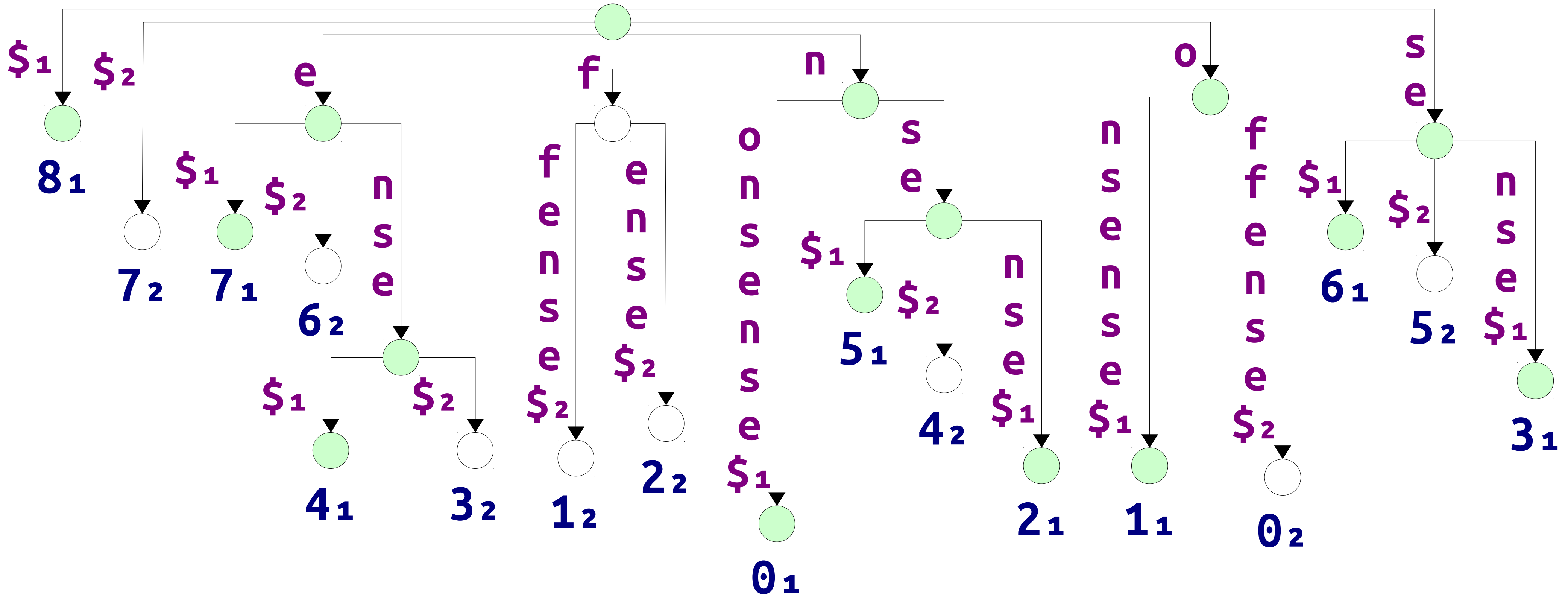


Observation: Any common substring of T_1 and T_2 will be a prefix of a suffix of T_1 and a prefix of a suffix of T_2.

```
nonsense$₁
012345678
```

```
offense$₂
01234567
```

Longest Common Substring

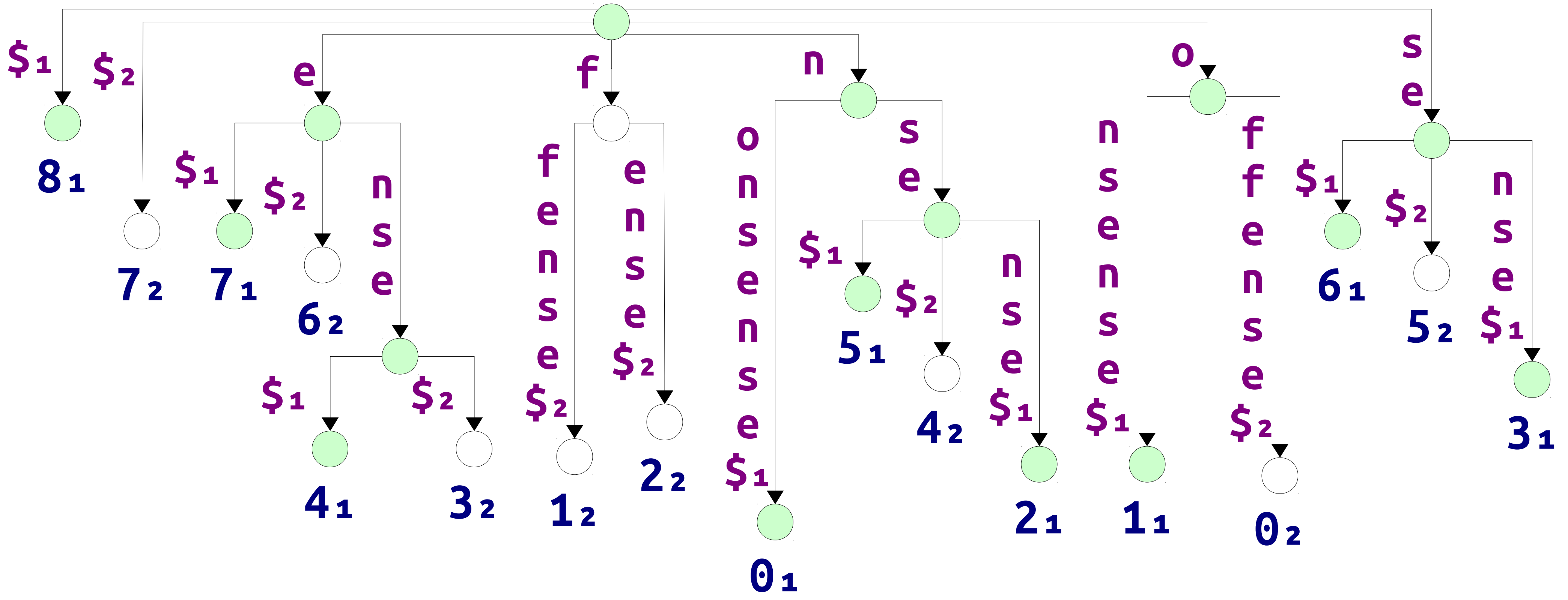


nonsense$_1$
012345678

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.

offense$_2$
01234567

Longest Common Substring

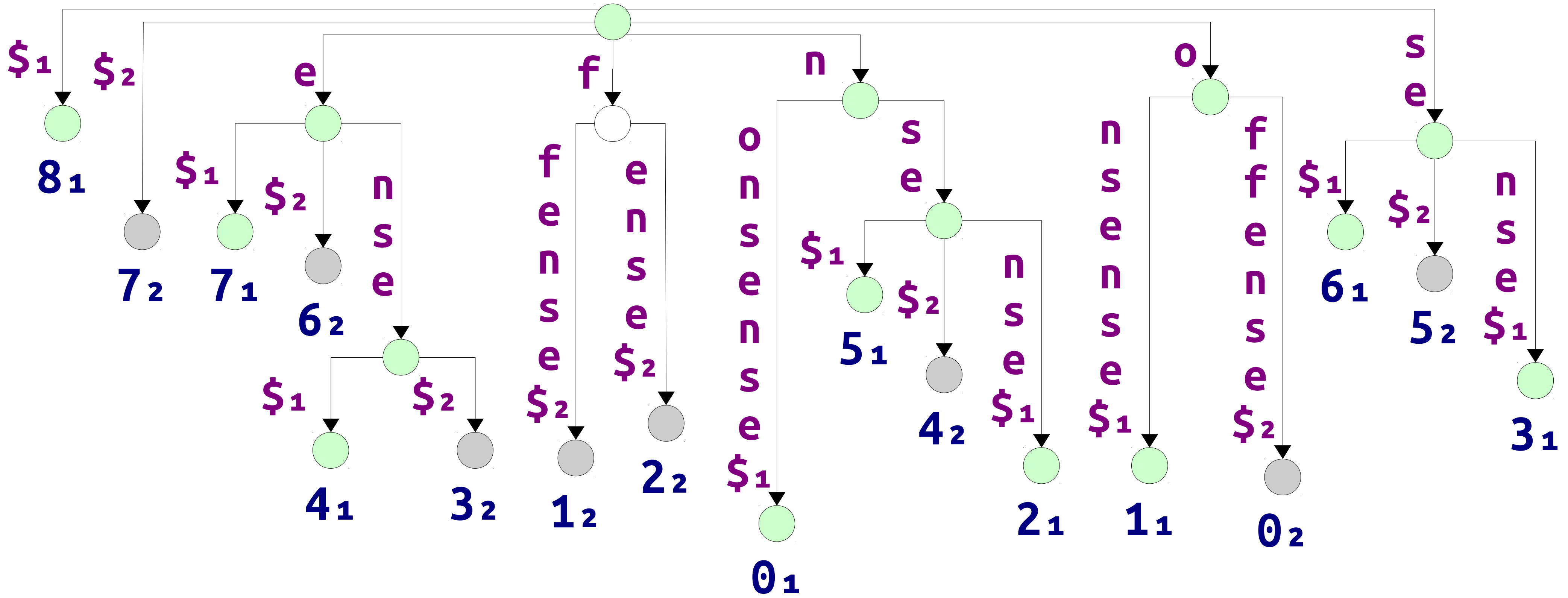


Observation: Any common substring of T_1 and T_2 will be a prefix of a suffix of T_1 and a prefix of a suffix of T_2.

```
nonsense$₁
012345678
```

```
offense$₂
01234567
```

Longest Common Substring

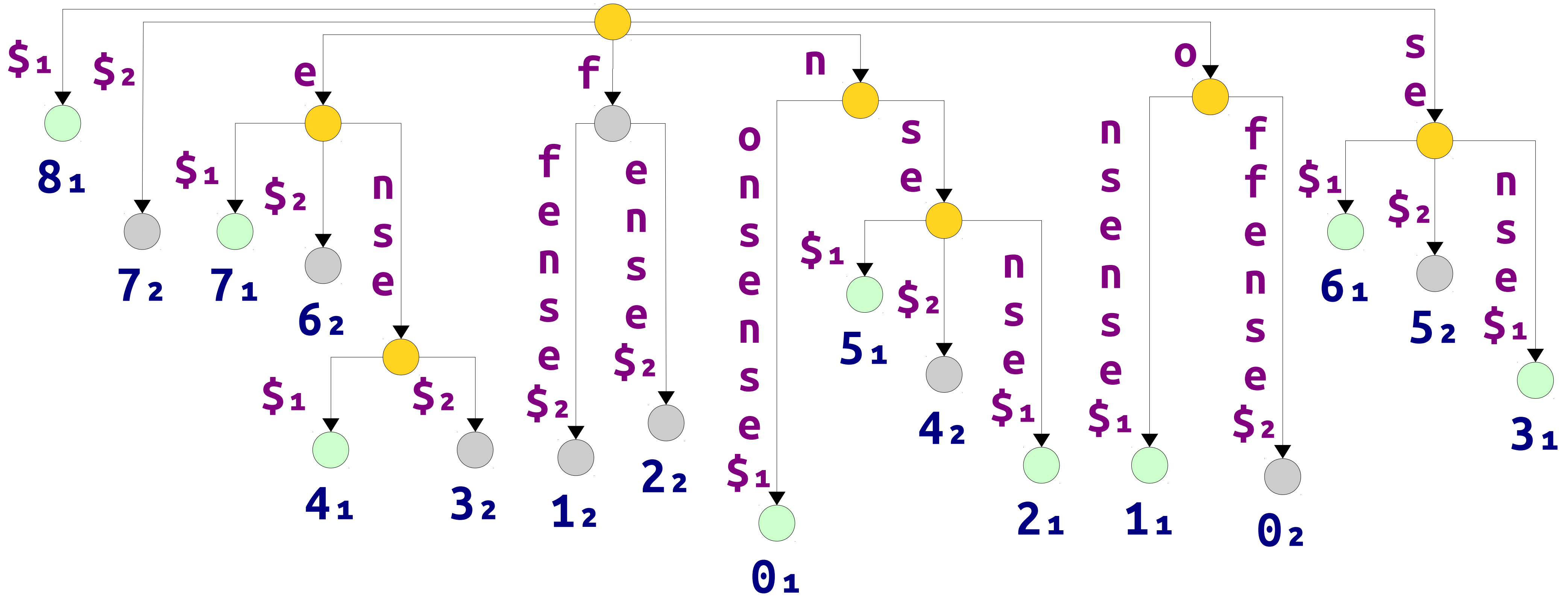


Observation: Any common substring of T_1 and T_2 will be a prefix of a suffix of T_1 and a prefix of a suffix of T_2.

```
nonsense$₁
012345678
```

```
offense$₂
01234567
```

Longest Common Substring

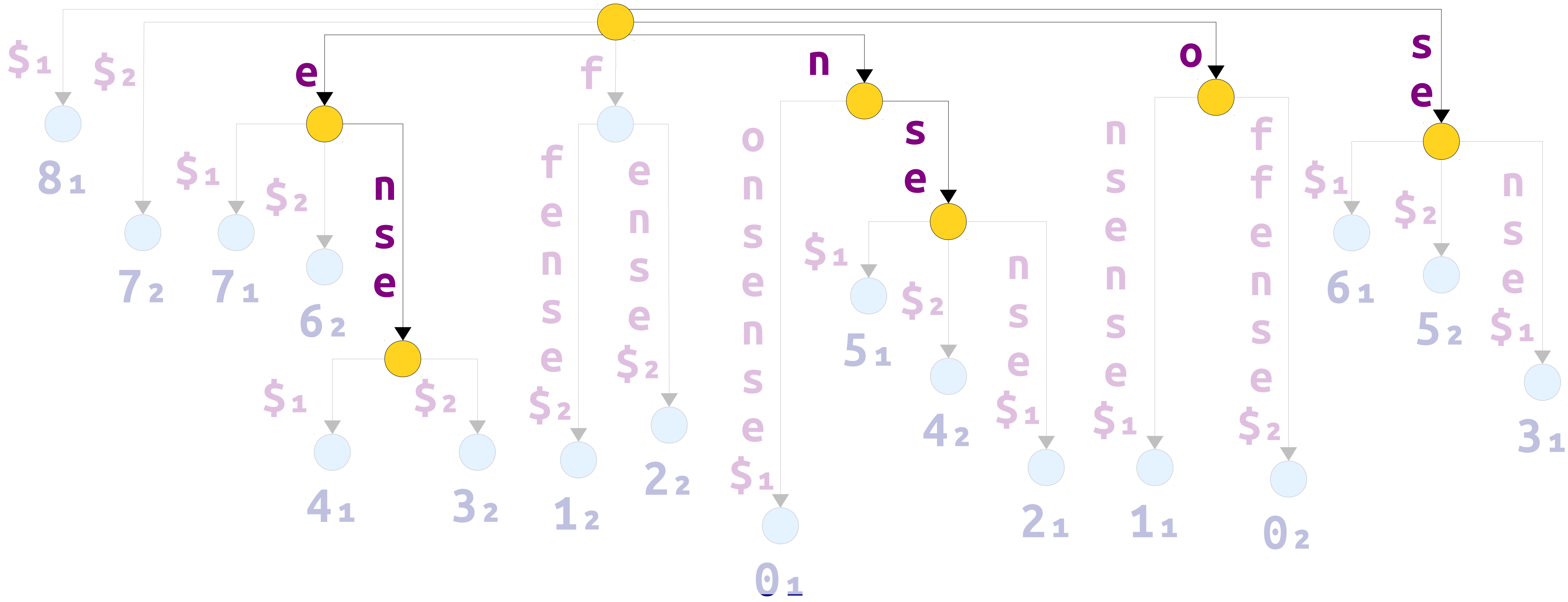


nonsense$\$_1$
012345678

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.

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.

offense$₂
01234567

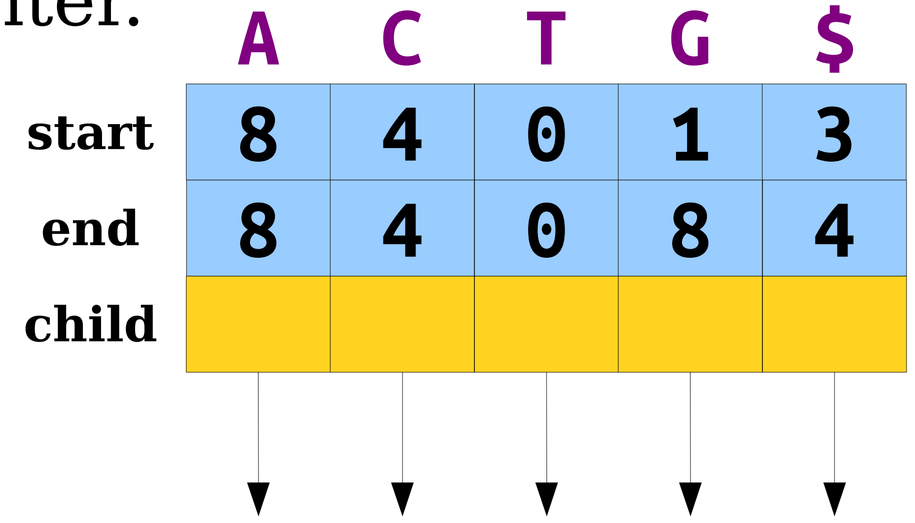
Longest Common Substring

- 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)**.

Suffix Trees: The Catch

Space Usage

- Suffix trees are memory hogs.
- Suppose Σ = {`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!

Can we get the flexibility of a suffix tree without the memory costs?

Yes… kinda!

Suffix Arrays

- A ***suffix array*** for a string *T* is an array of the suffixes of *T*$, stored in sorted order.
- By convention, $ precedes all other characters.

0	nonsense$
1	onsense$
2	nsense$
3	sense$
4	ense$
5	nse$
6	se$
7	e$
8	$

Suffix Arrays

- A ***suffix array*** for a string *T* is an array of the suffixes of *T*$, stored in sorted order.
- By convention, $ precedes all other characters.

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

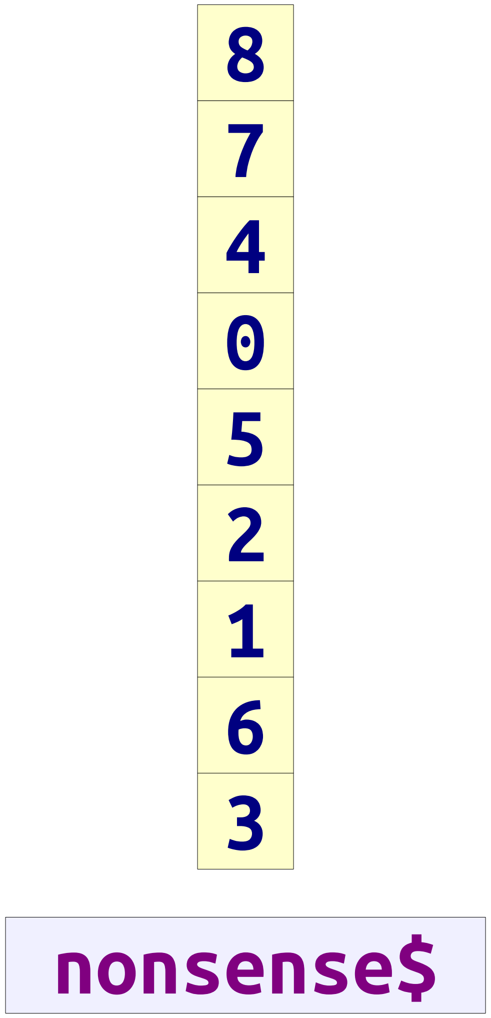
Representing Suffix Arrays

- Suffix arrays are typically represented implicitly by just storing the indices of the suffixes in sorted order rather than the suffixes themselves.
- Space required: $\Theta(m)$.
- More precisely, space for $T\$$, plus one extra word for each character.

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

Representing Suffix Arrays

- Suffix arrays are typically represented implicitly by just storing the indices of the suffixes in sorted order rather than the suffixes themselves.
- Space required: $\Theta(m)$.
- More precisely, space for $T\$$, plus one extra word for each character.



Searching a Suffix Array

- ***Recall:*** *P* is a substring of *T* iff it's a prefix of a suffix of *T*.
- All matches of *P* in *T* have a common prefix, so they'll be stored consecutively.
- Can find all matches of *P* in *T* by doing a binary search over the suffix array.

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

Analyzing the Runtime

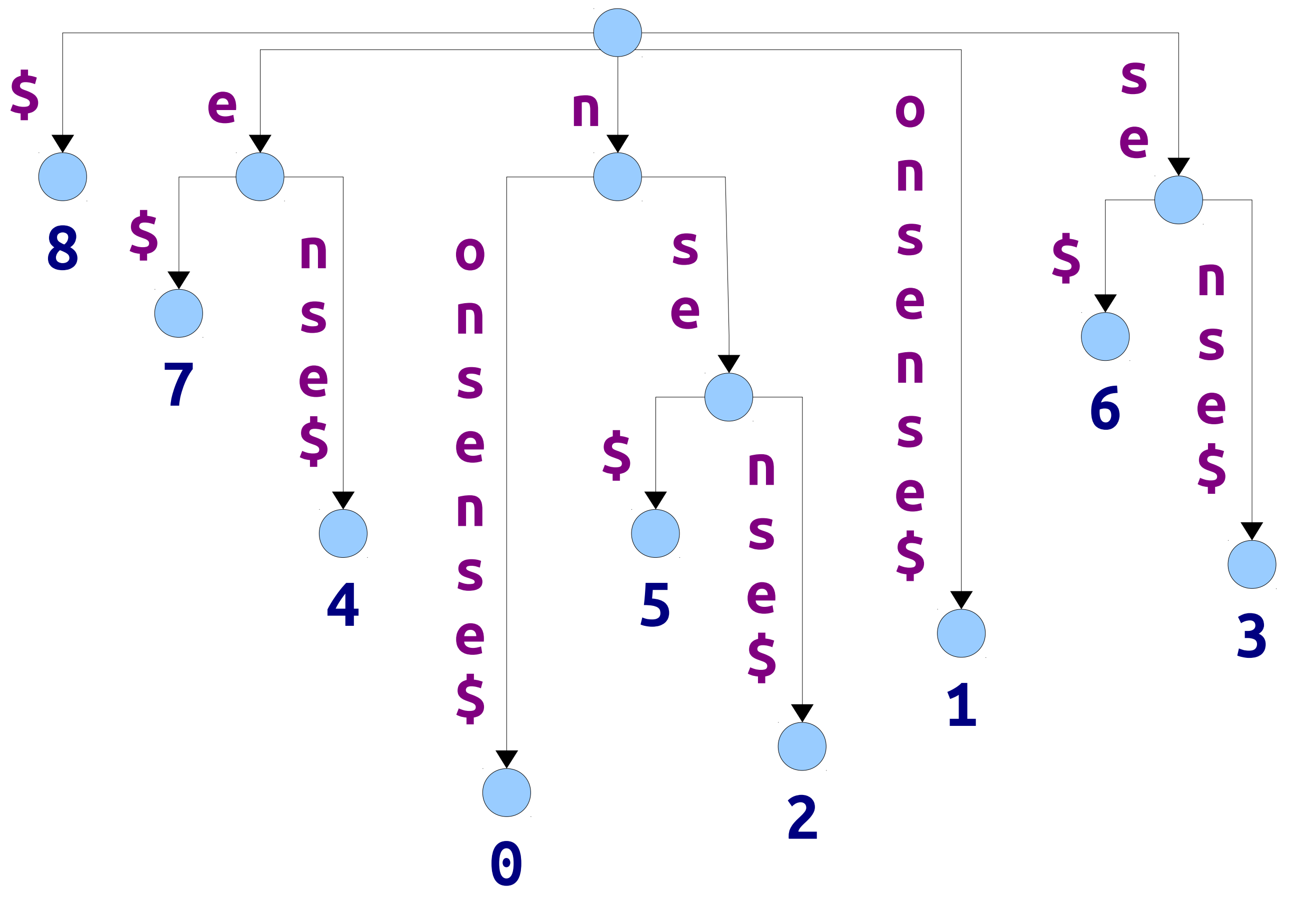
- The binary search will require O(log m) probes into the suffix array.
- Each comparison takes time O(n): have to compare P against the current suffix.
- Time for binary searching: O(n log m).
- Time to report all matches after that point: O(z).
- Total time: **O(n log m + z)**.

Why the Slowdown?

A Loss of Structure

- Many algorithms on suffix trees involve looking for internal nodes with various properties:
  - Longest repeated substring: internal node with largest string depth.
  - Longest common substring: internal node with largest string depth that has a child from each string.
- Because suffix arrays do not store the tree structure, we lose access to this information.

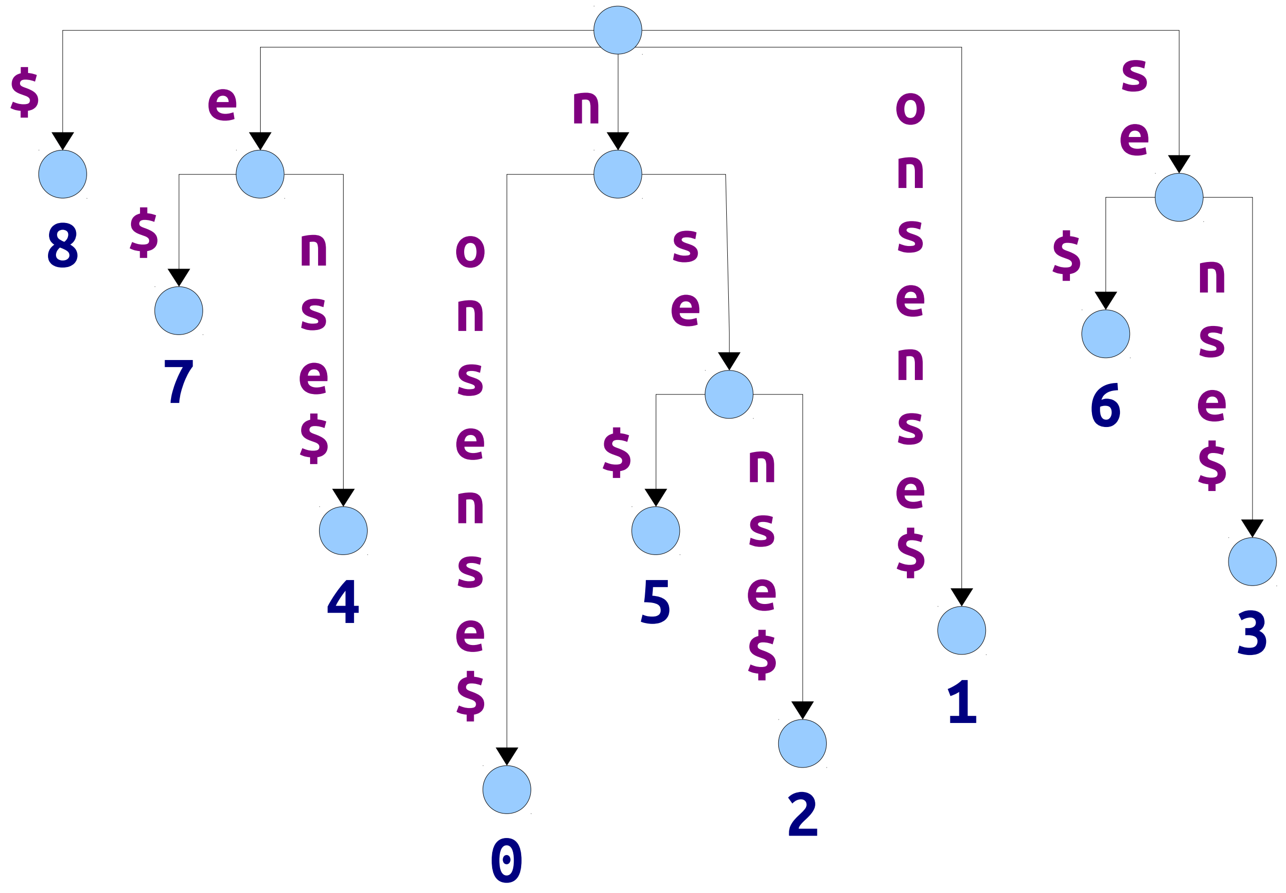
Suffix Trees and Suffix Arrays



nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

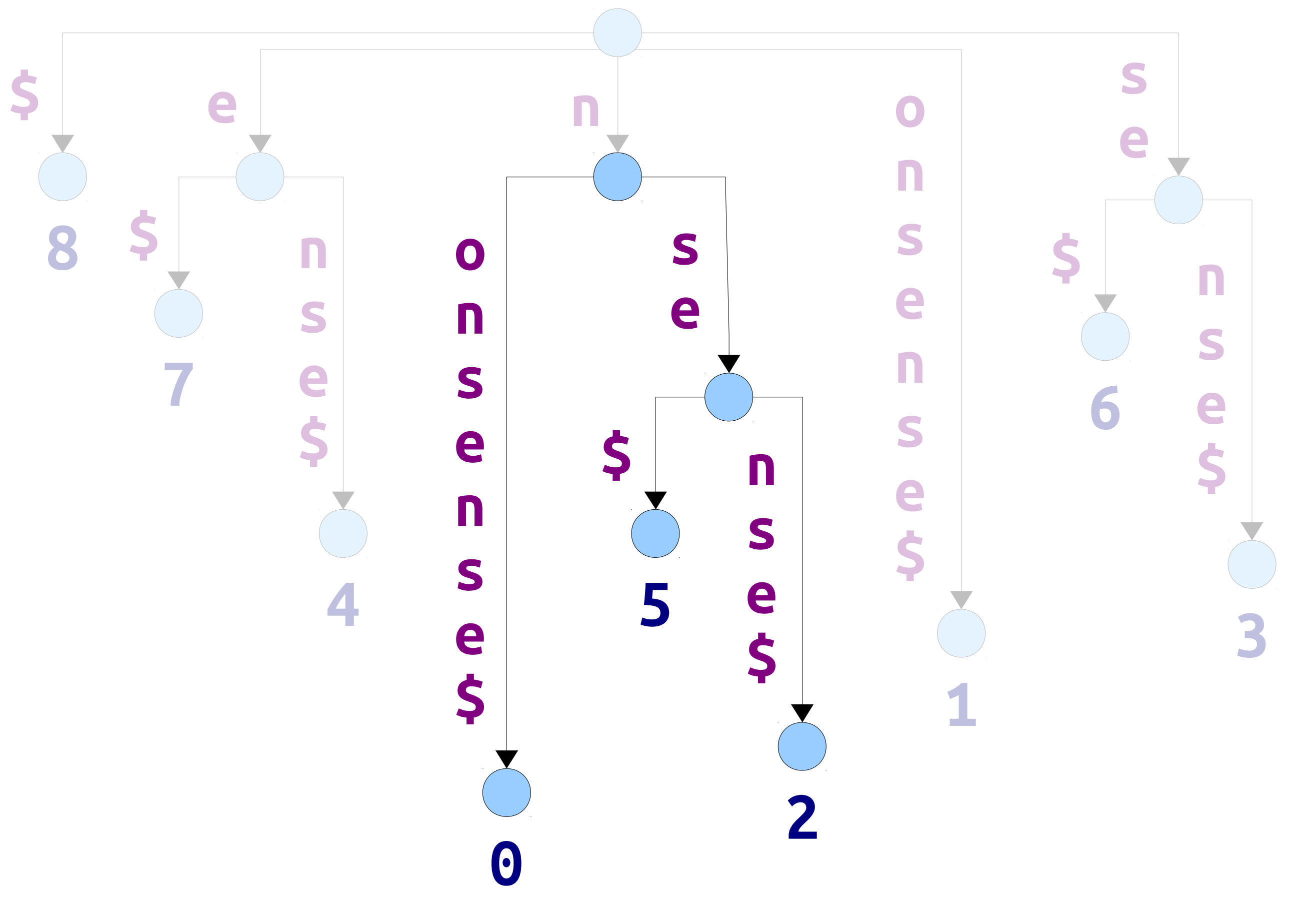
Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

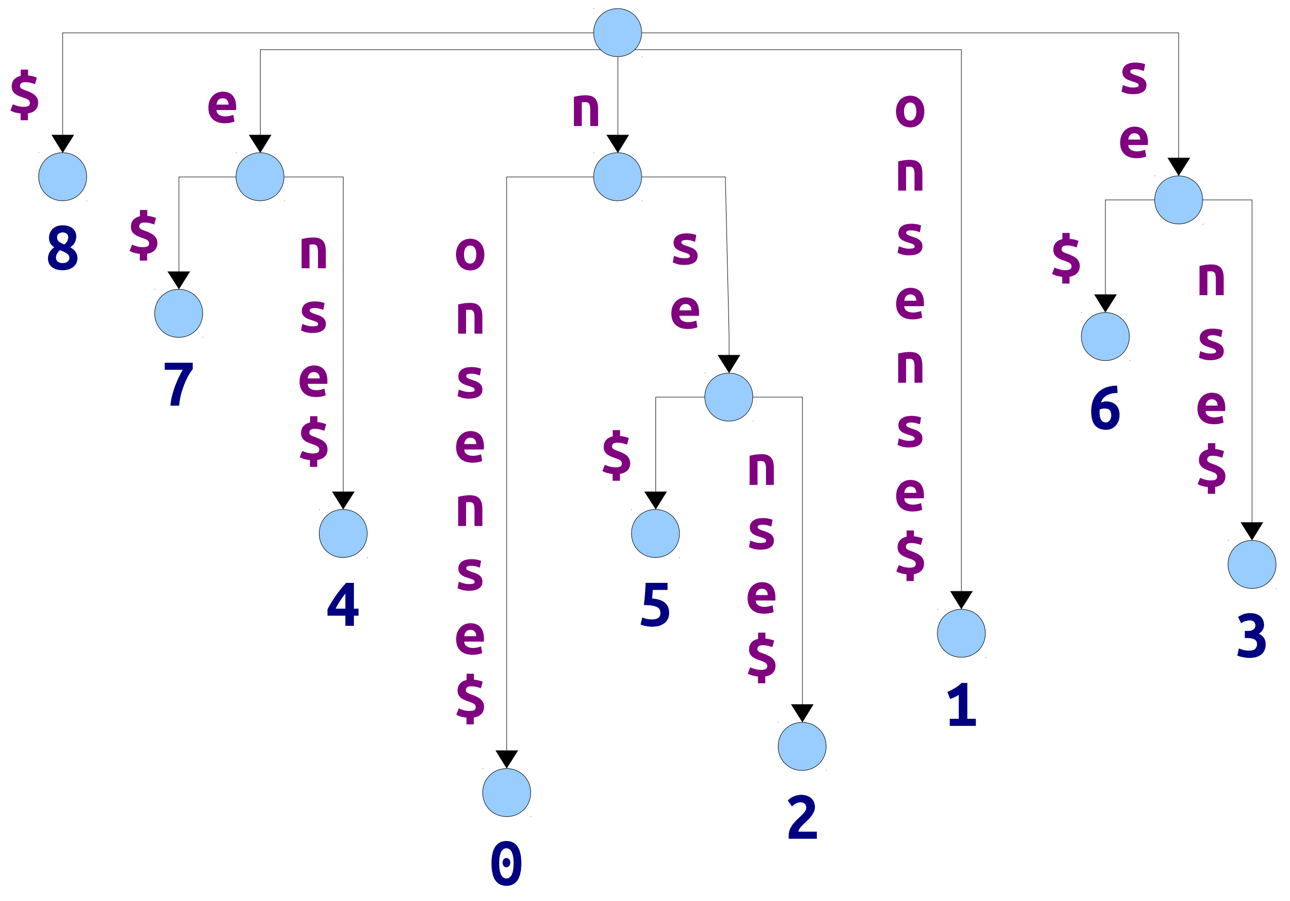
Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

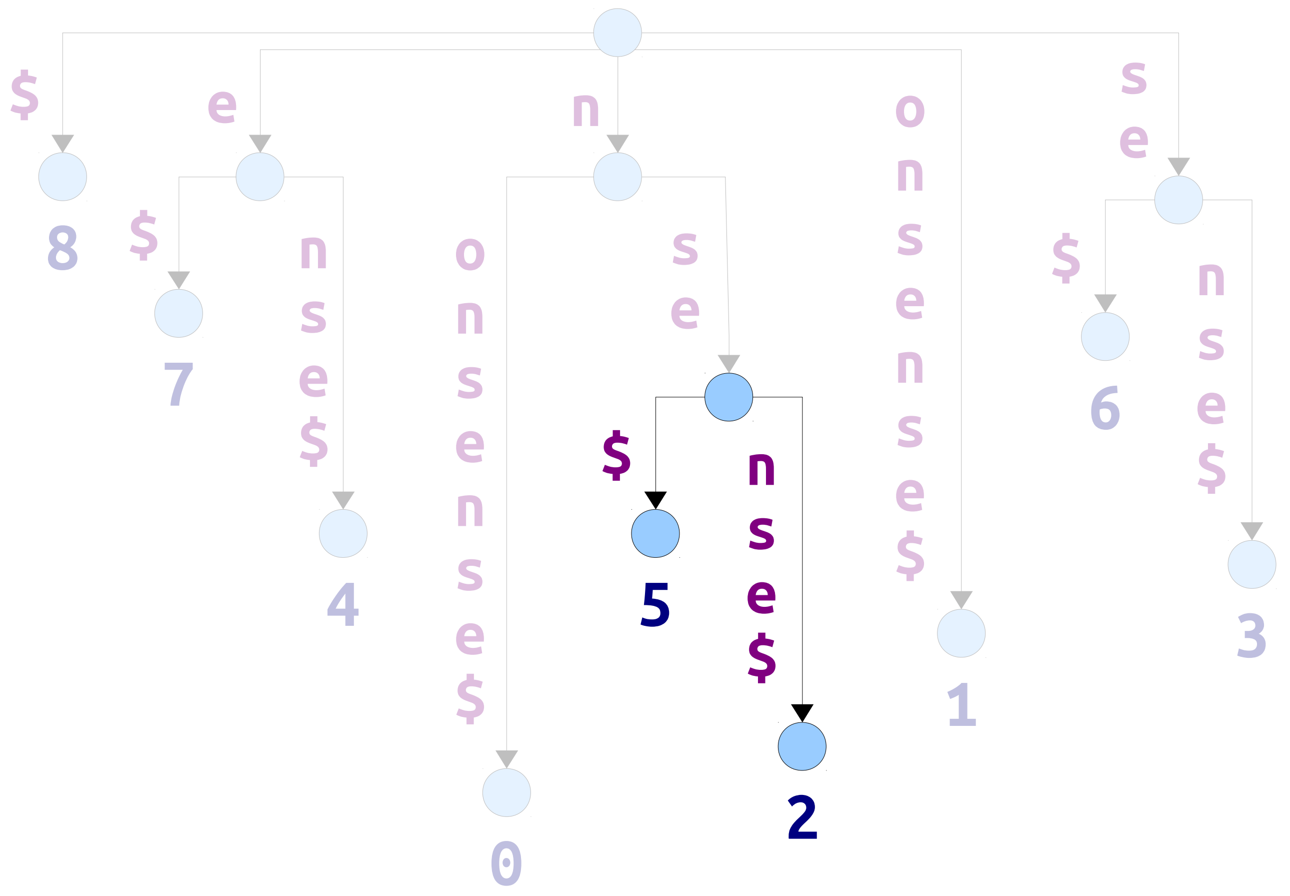
Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

Suffix Trees and Suffix Arrays



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nonsense$
012345678

The longest common prefix of a range of strings in a suffix array corresponds to the lowest common ancestor of those suffixes in the suffix tree.

Longest Common Prefixes

- Given two strings *x* and *y*, the ***longest common prefix*** or (***LCP***) of *x* and *y* is the longest prefix of *x* that is also a prefix of *y*.
- The LCP of *x* and *y* is denoted **lcp(*x*, *y*)**.
- ***Fun fact:*** There is an O(*m*)-time algorithm for computing LCP information on a suffix array.
- Let's see how it works.

Pairwise LCP

- ***Fact:*** There is an algorithm (due to Kasai et al.) that constructs, in time O(m), an array of the LCPs of adjacent suffix array entries.
- The algorithm isn't that complex, but the correctness argument is a bit nontrivial.

	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

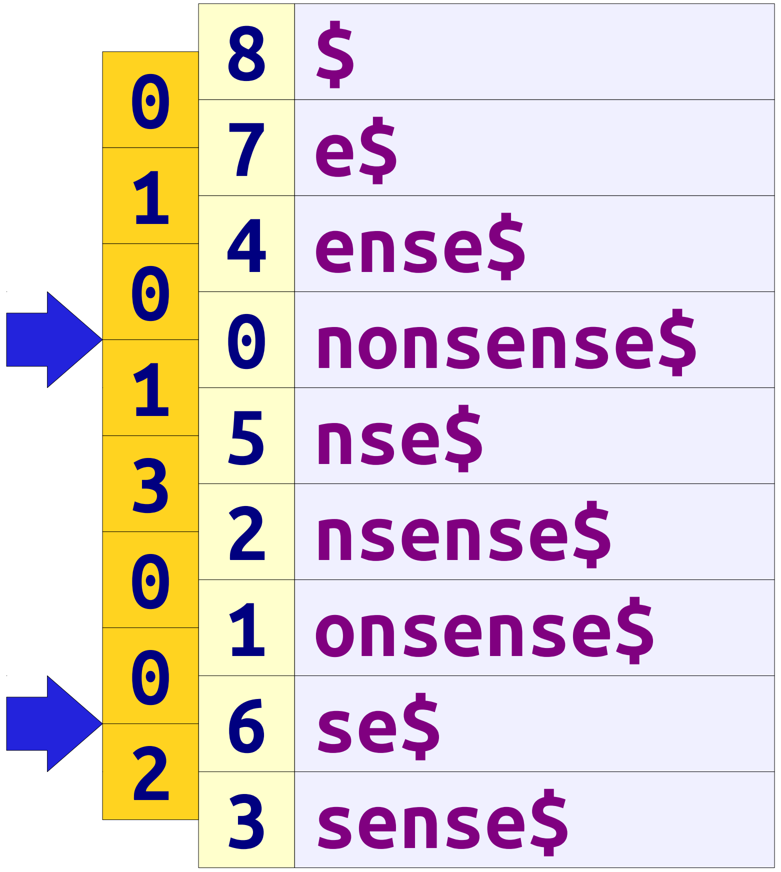
Pairwise LCP

- ***Claim:*** This information is enough for us to figure out the longest common prefix of a range of elements in the suffix array.

	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

Pairwise LCP

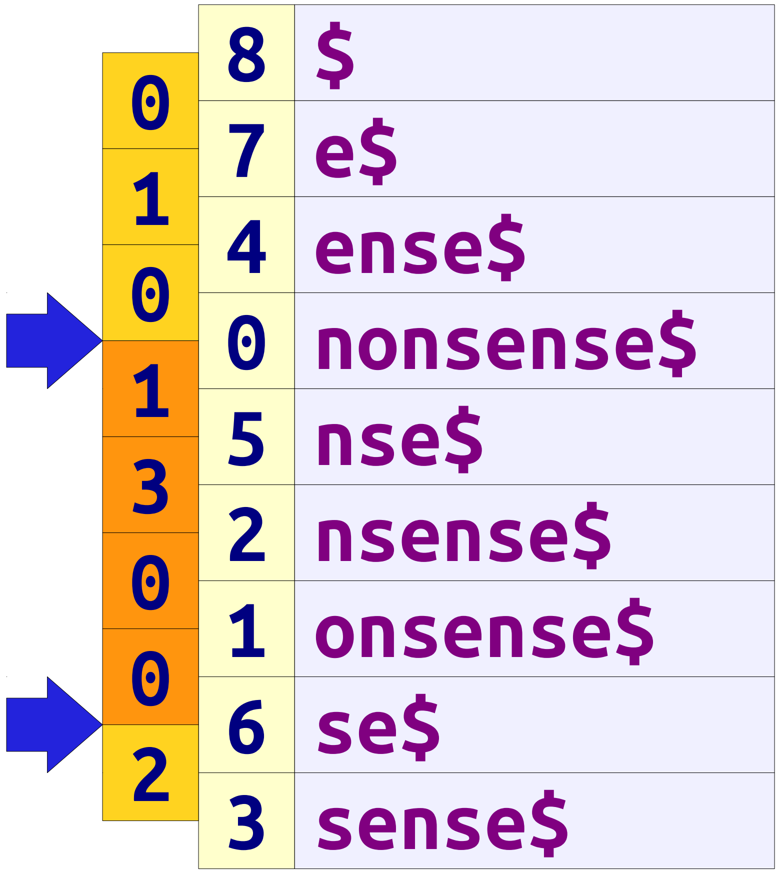
- ***Claim:*** This information is enough for us to figure out the longest common prefix of a range of elements in the suffix array.



	8	$
0	7	e$
1	4	ense$
0	0	nonsense$
1	5	nse$
3	2	nsense$
0	1	onsense$
0	6	se$
2	3	sense$

Pairwise LCP

- ***Claim:*** This information is enough for us to figure out the longest common prefix of a range of elements in the suffix array.



Hey, look! It's a range minimum query problem!

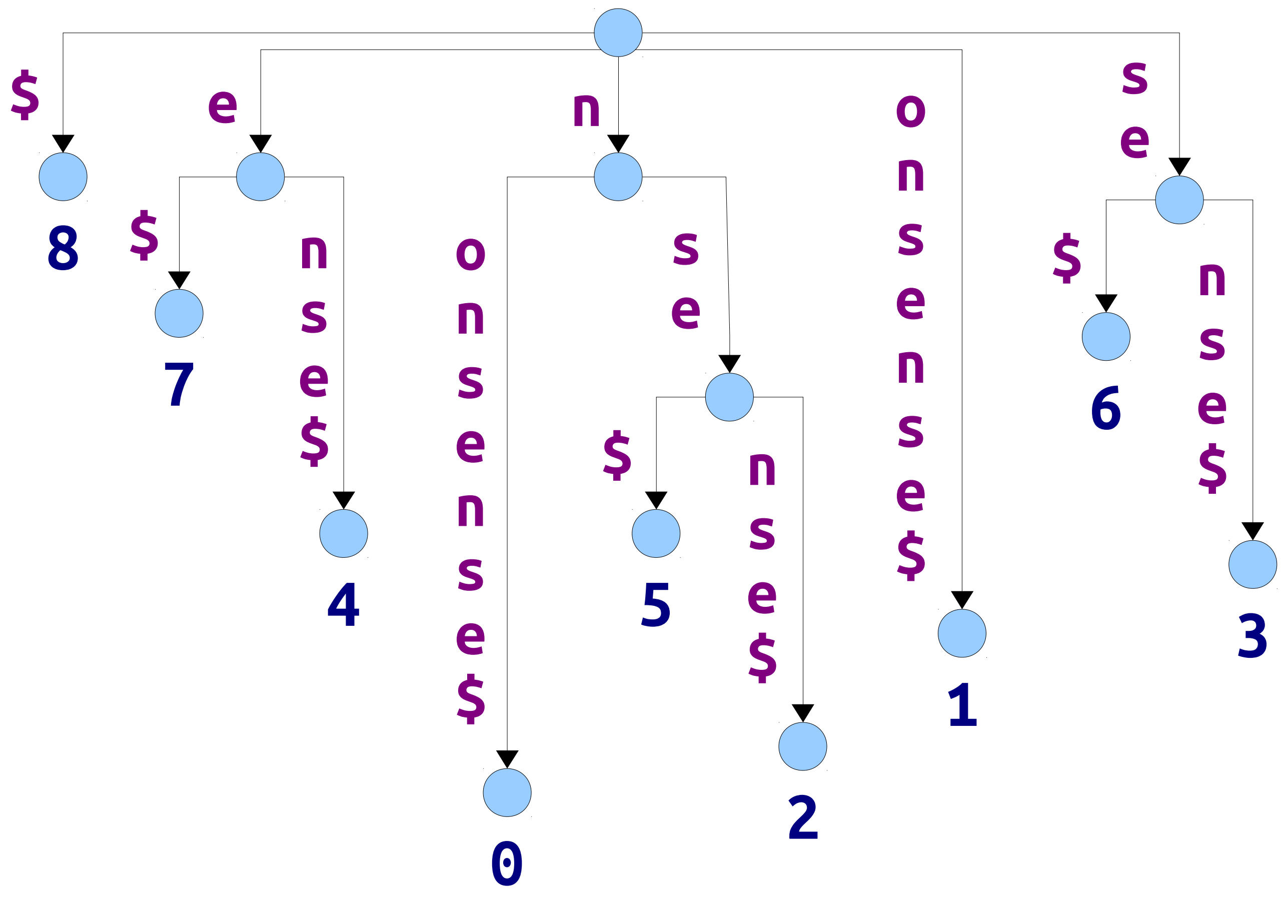
Computing LCPs

- To preprocess a suffix array to support O(1) LCP queries:
  - Use Kasai's O(m)-time algorithm to build the LCP array.
  - Build an RMQ structure over that array in time O(m) using Fischer-Heun.
  - Use the precomputed RMQ structure to answer LCP queries over ranges.
- Requires O(m) preprocessing time and only O(1) query time.

Searching a Suffix Array

- ***Recall:*** Can search a suffix array of T for all matches of a pattern P in time O(n log m + z).
- If we've done O(m) preprocessing to build the LCP information, we can speed this up.

Searching a Suffix Array

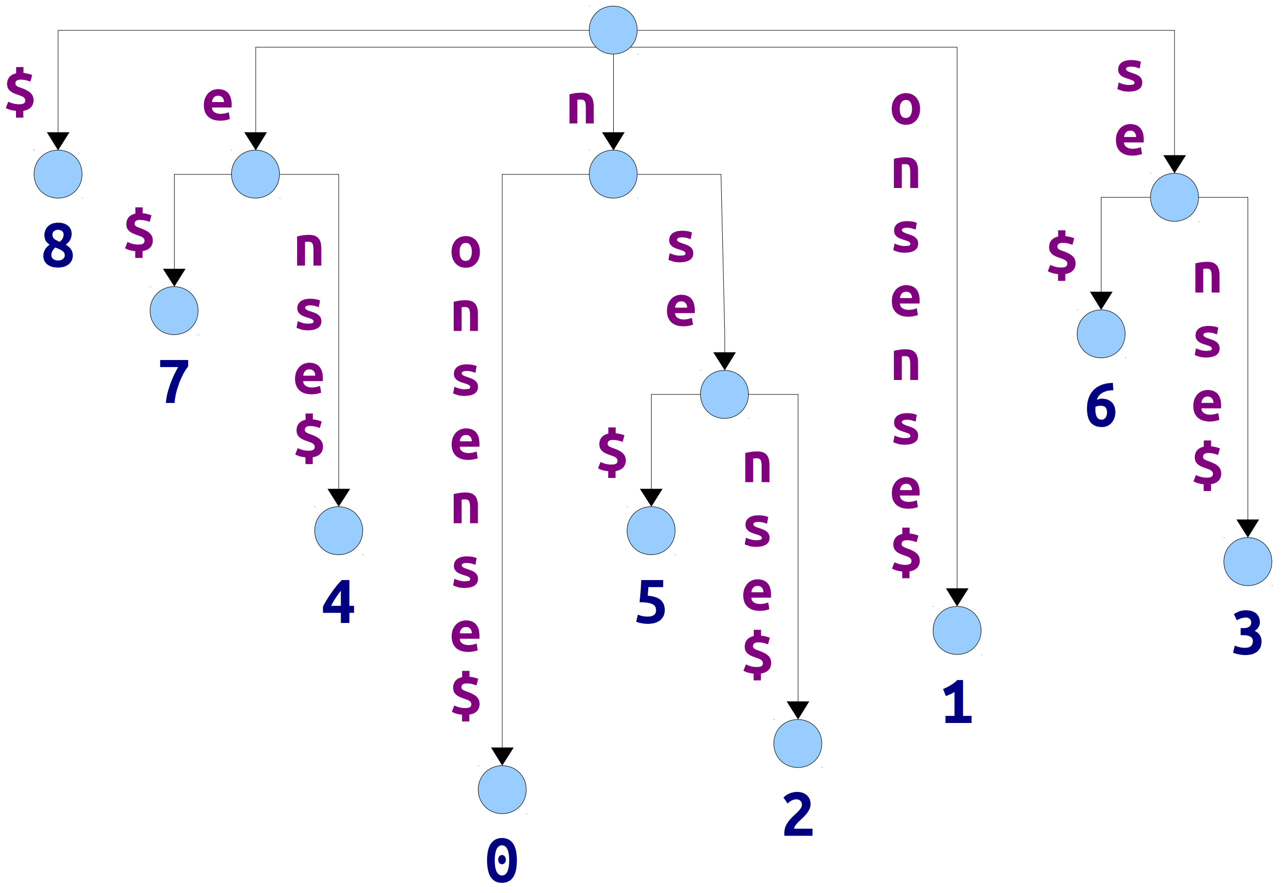


nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

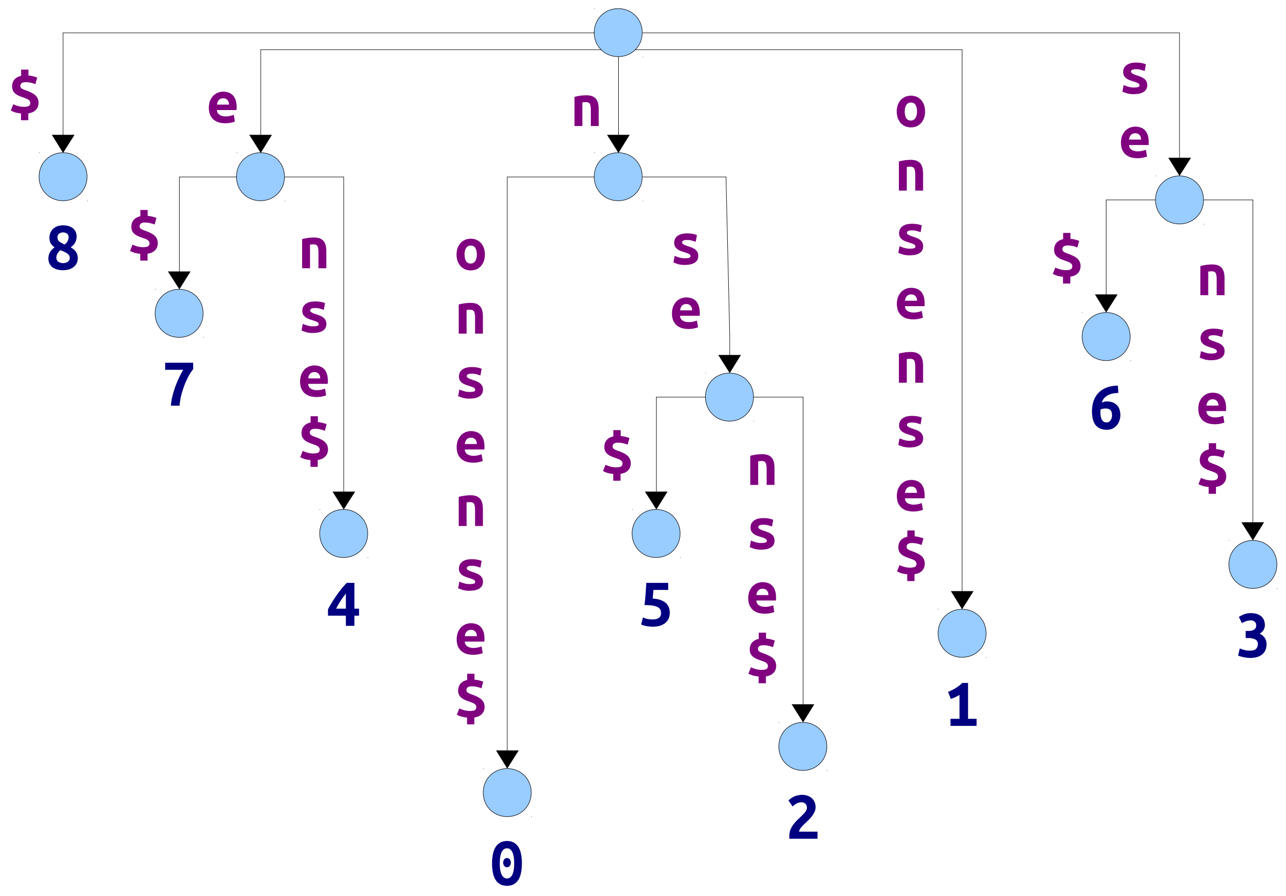


nonsense$
012345678

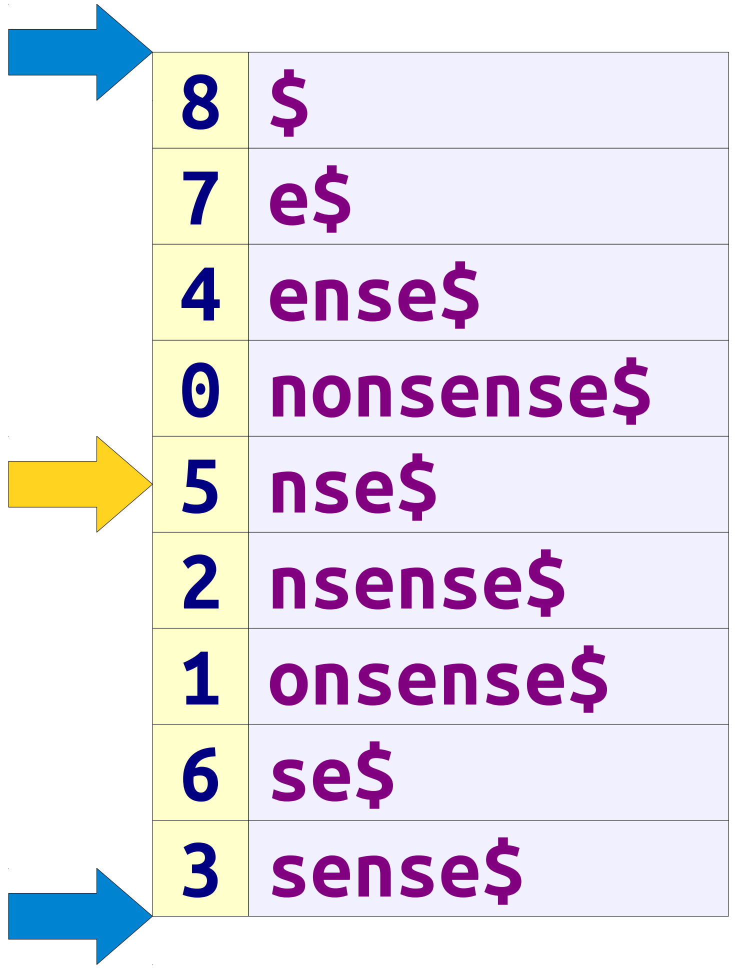
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



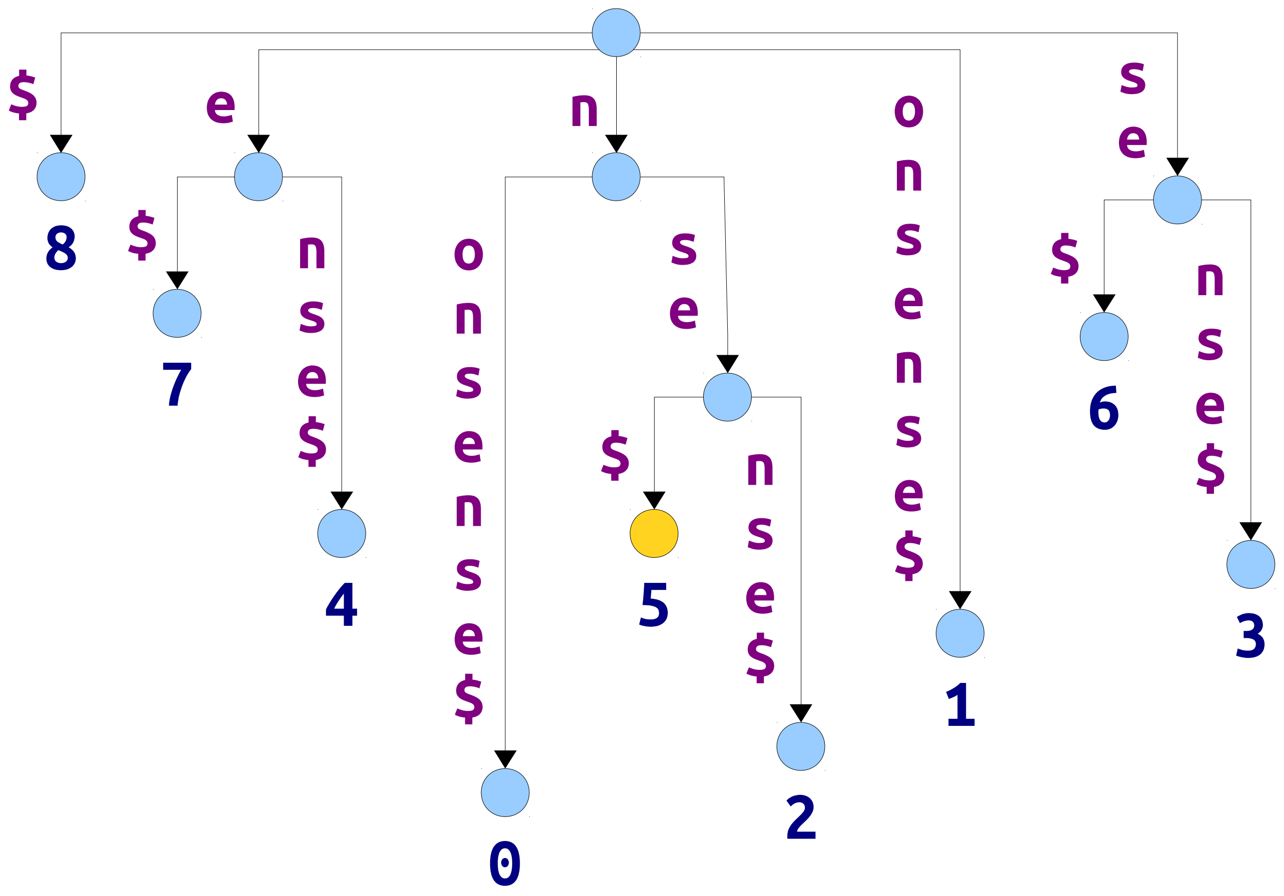
nonsense$
012345678



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

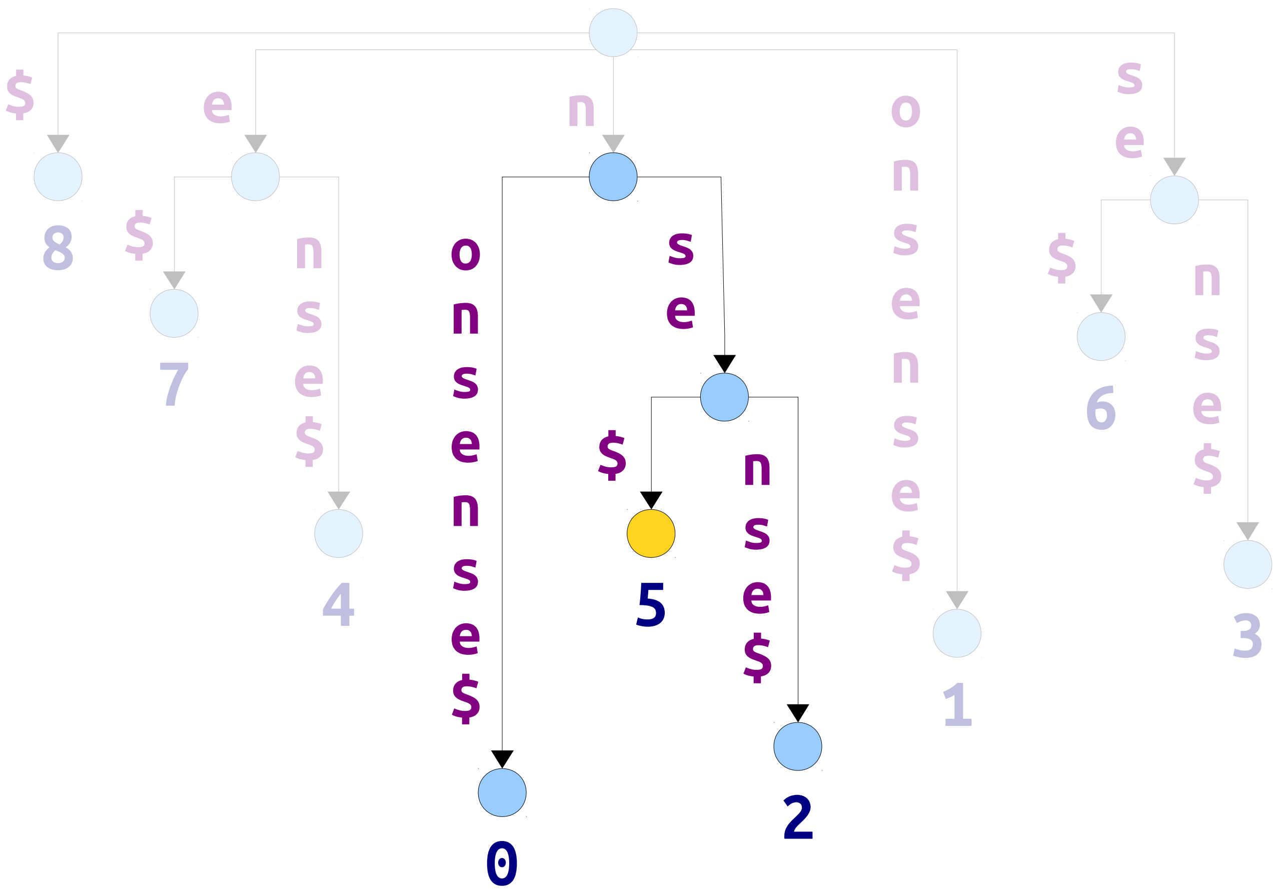


nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

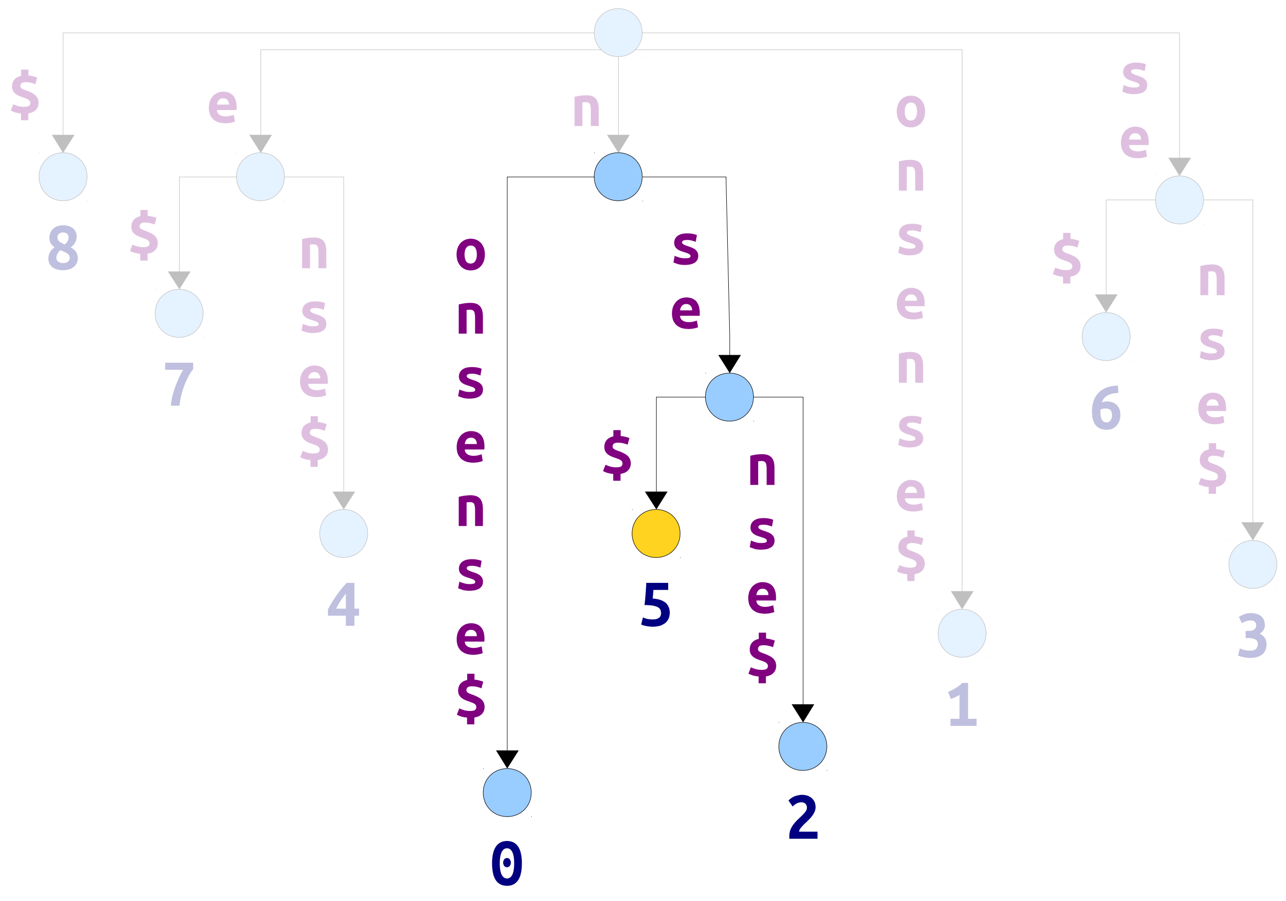


nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

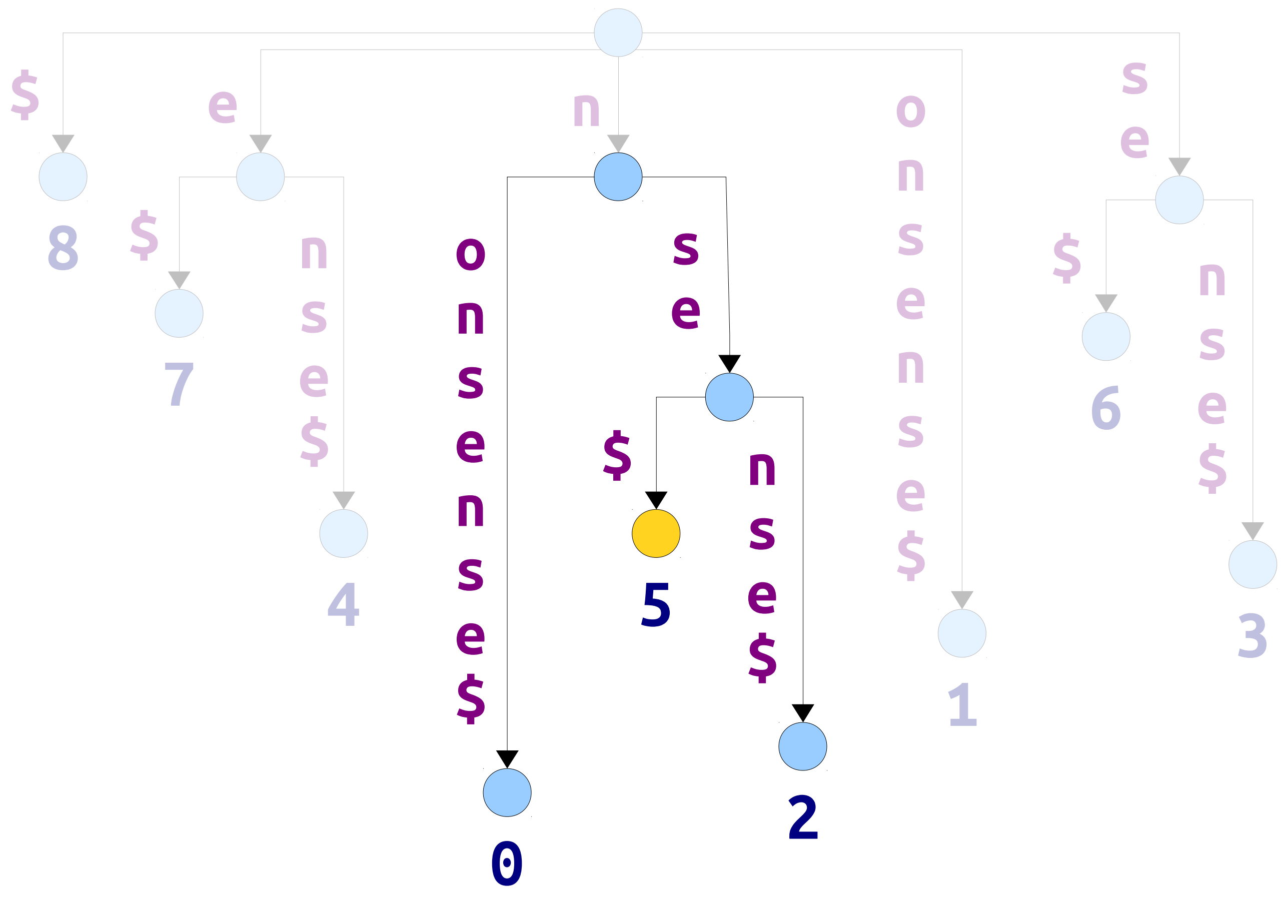


nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>n</u>ons

Searching a Suffix Array

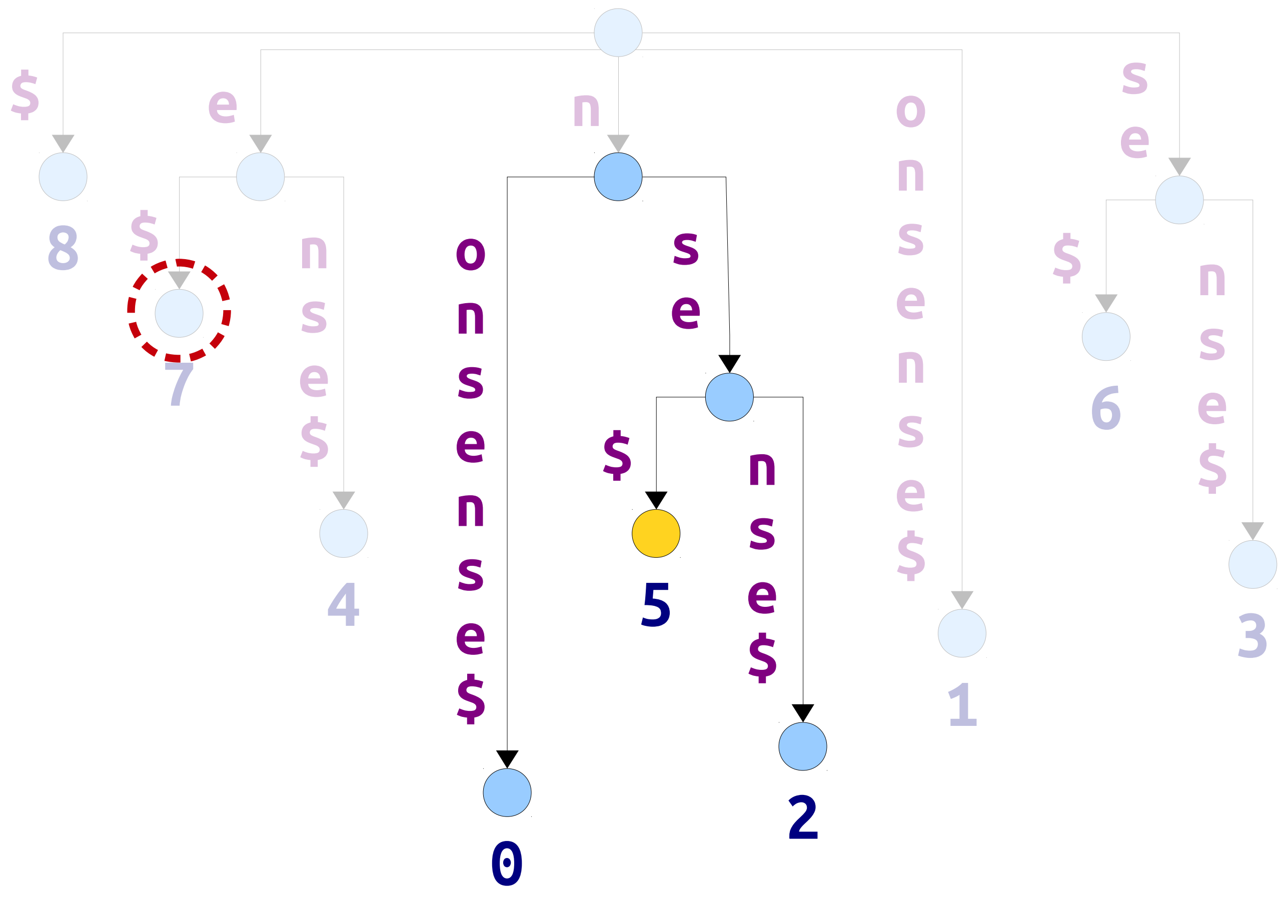


nonsense$
012345678

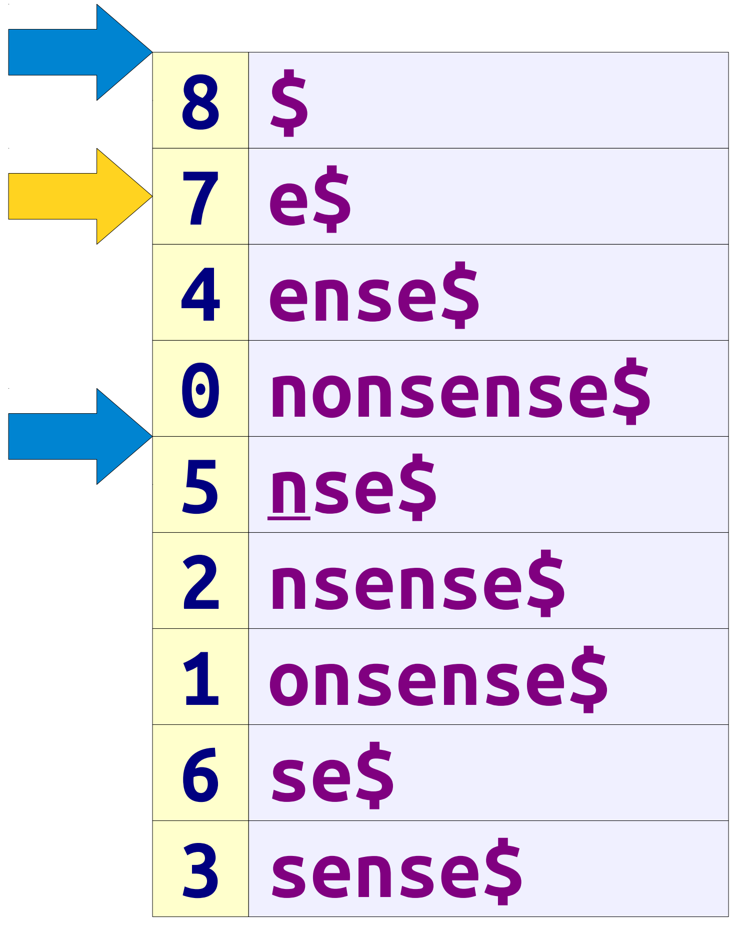
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



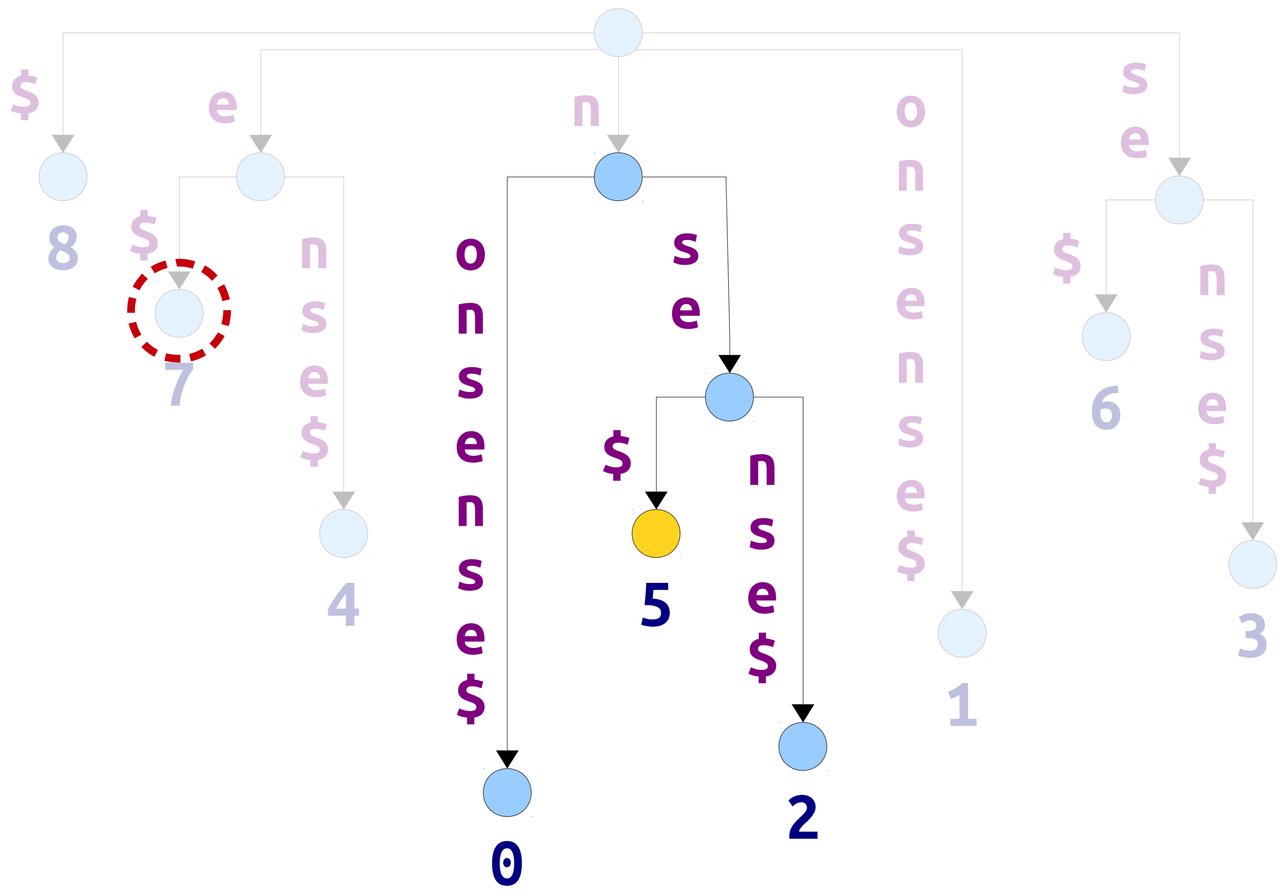
nonsense$
012345678



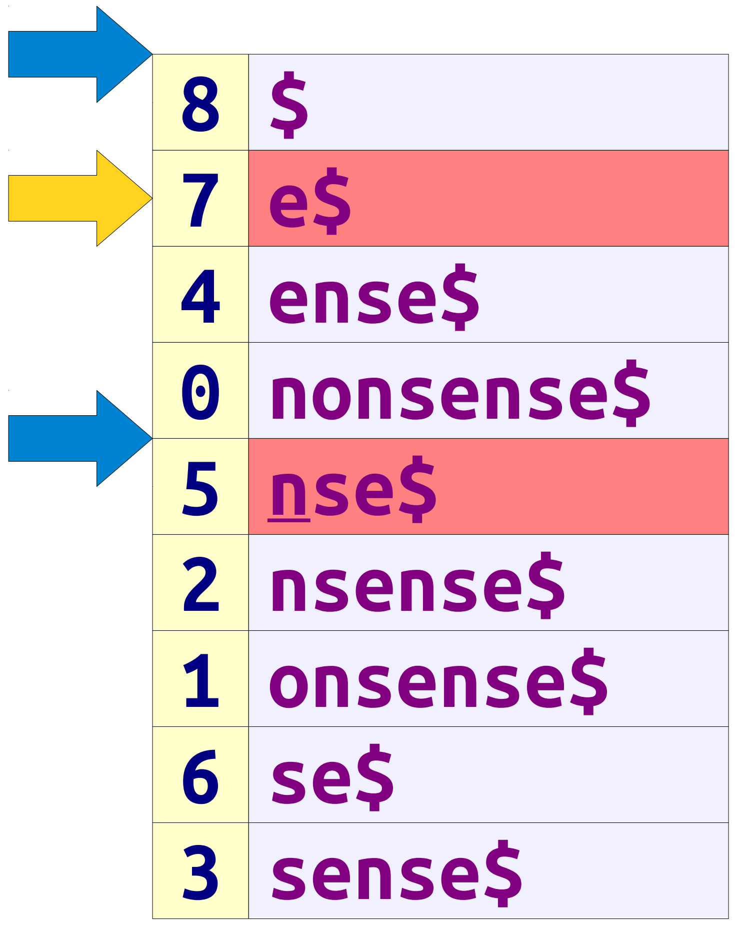
8	$
7	e$
4	ense$
0	nonsense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>n</u>ons

Searching a Suffix Array



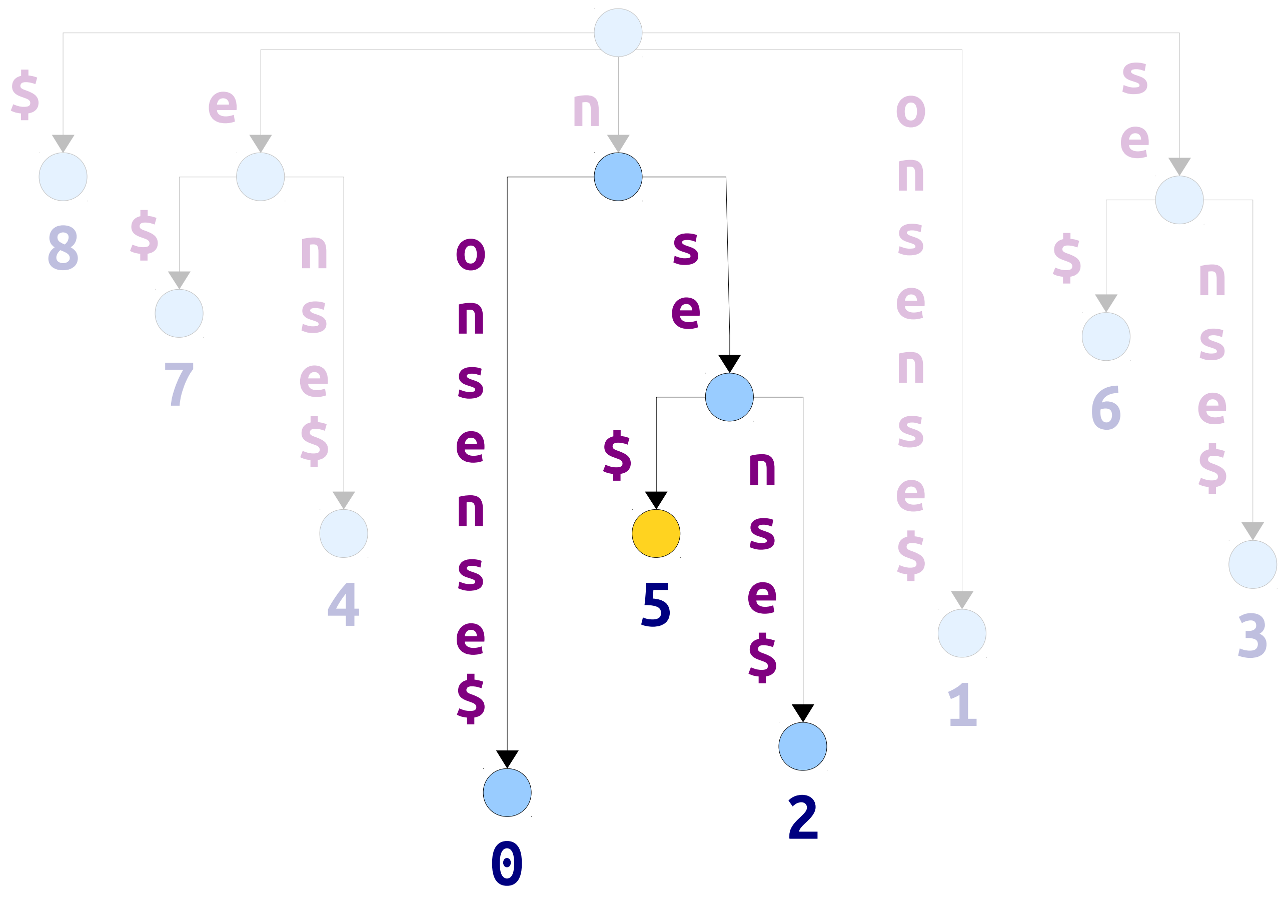
nonsense$
012345678



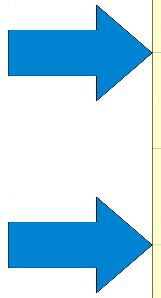
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



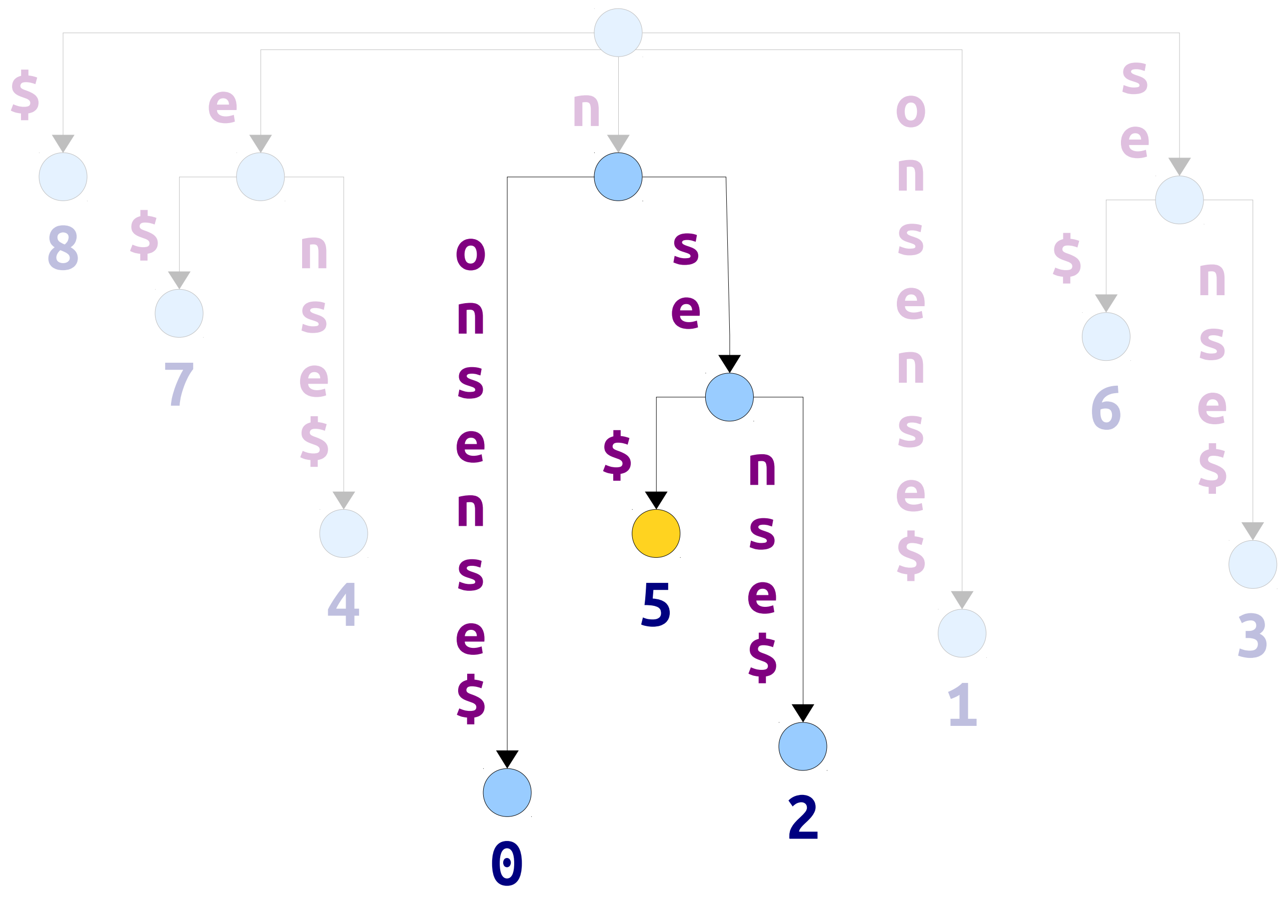
nonsense$
012345678



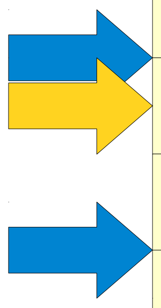
8	$
7	e$
4	ense$
0	nonsense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>n</u>ons

Searching a Suffix Array



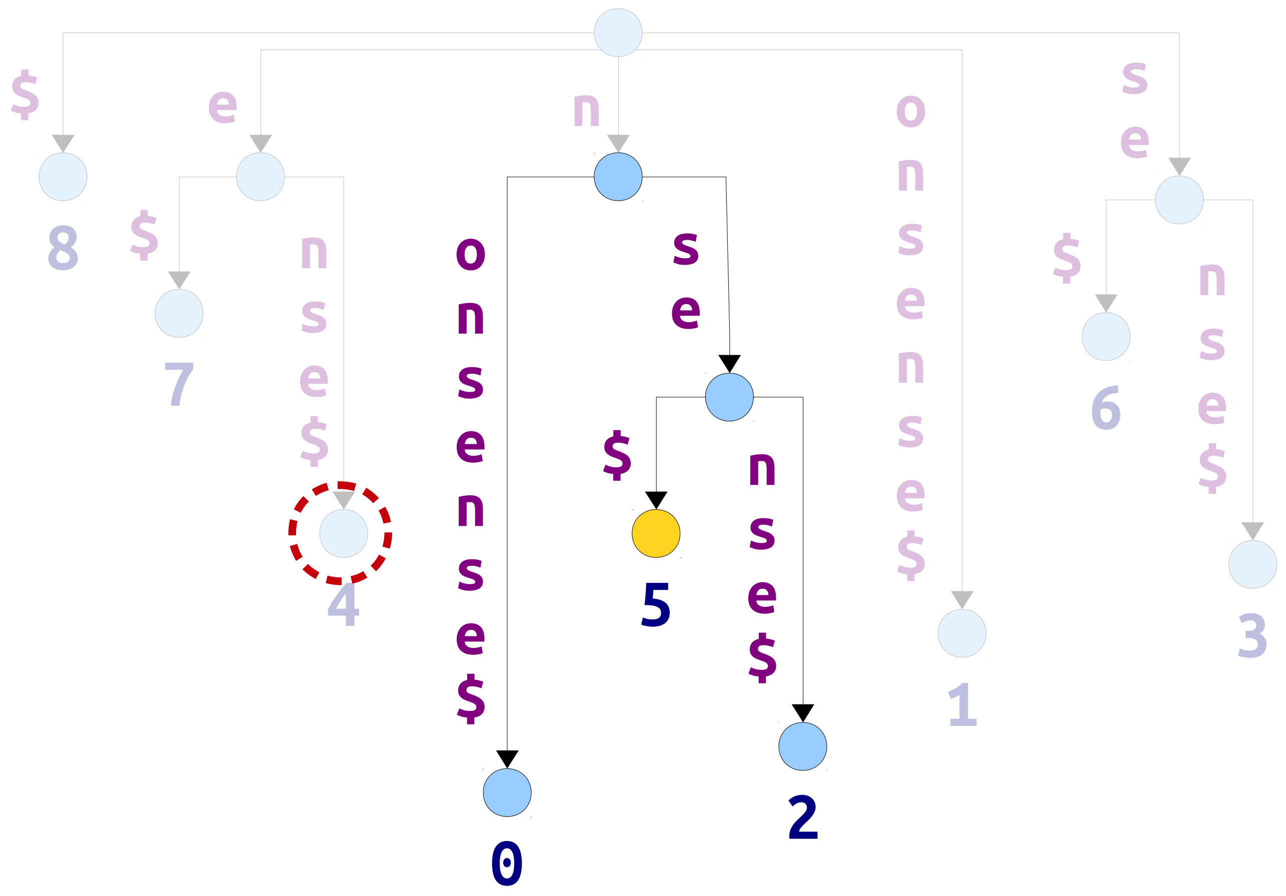
nonsense$
012345678



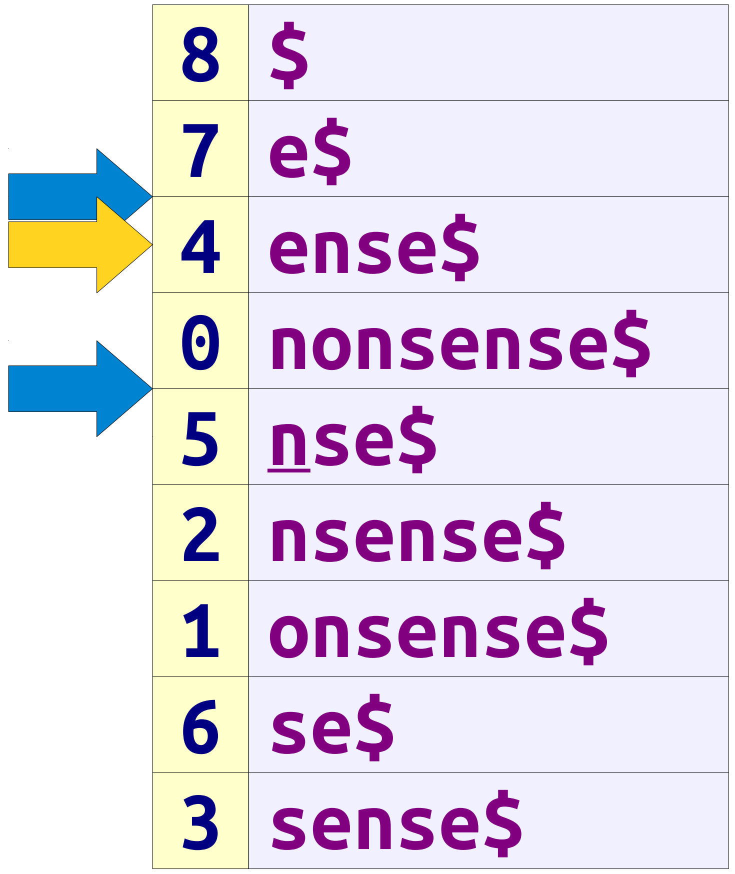
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



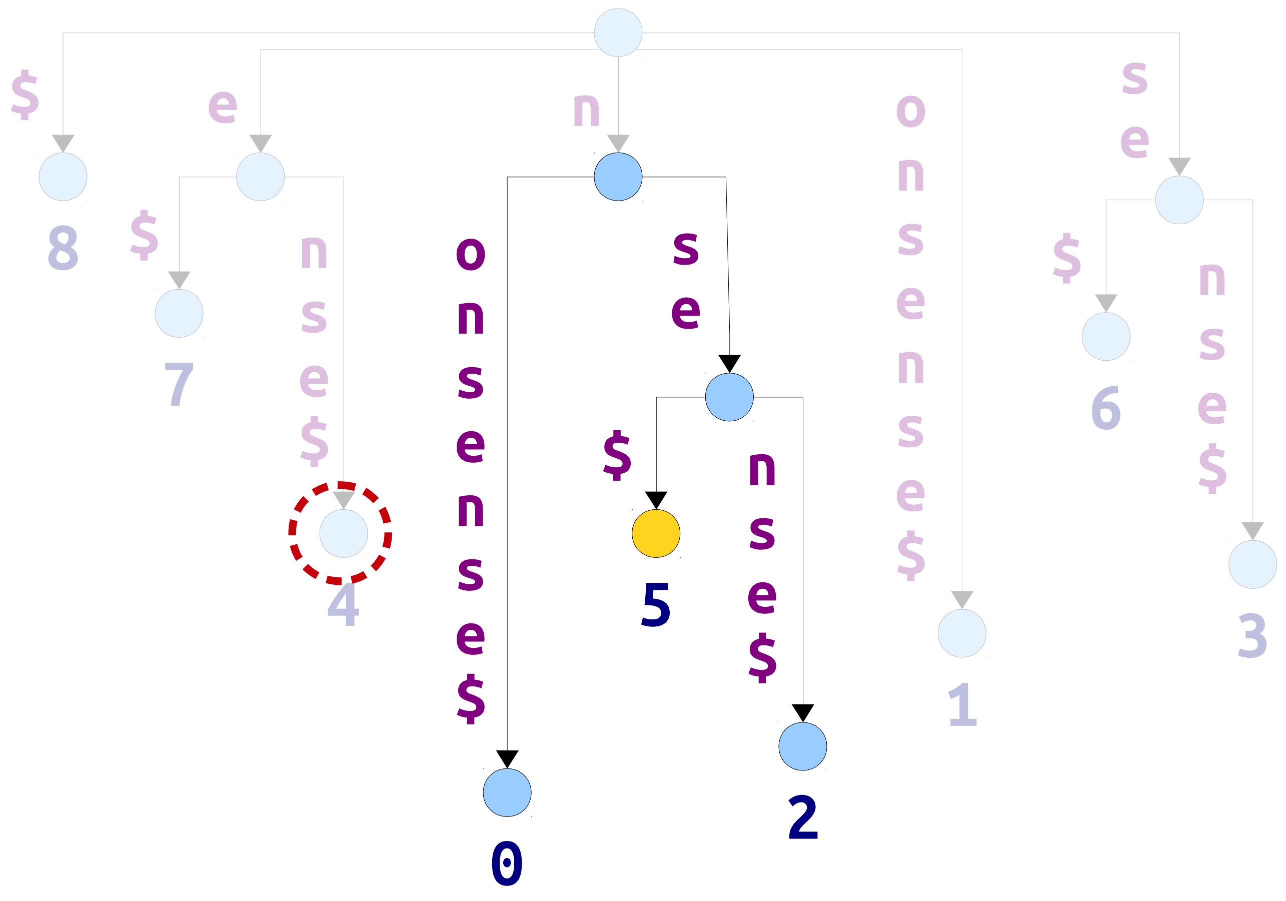
nonsense$
012345678



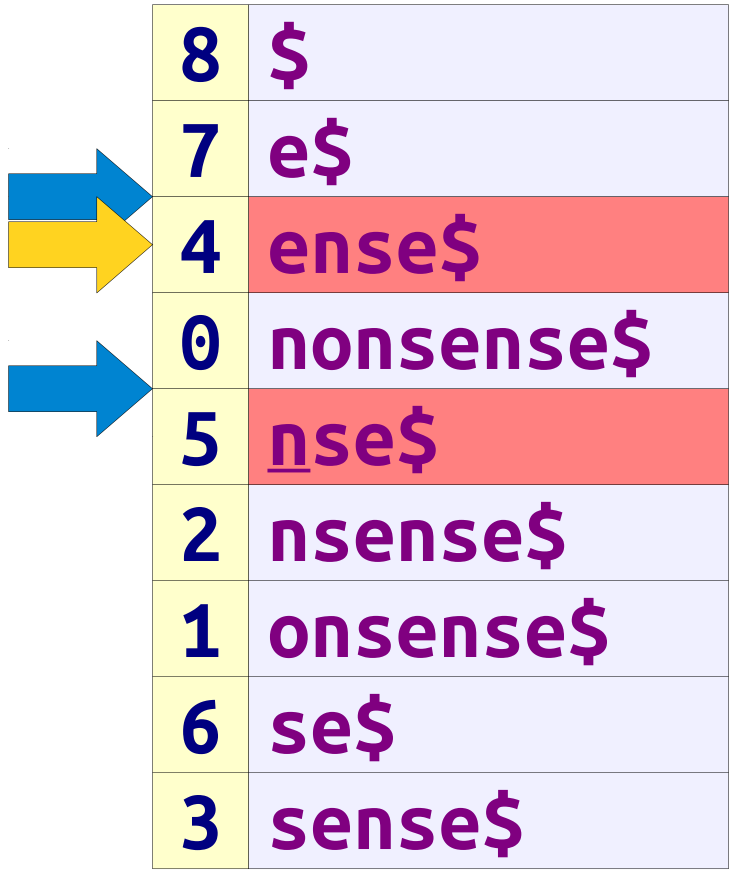
8	$
7	e$
4	ense$
0	nonsense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>n</u>ons

Searching a Suffix Array



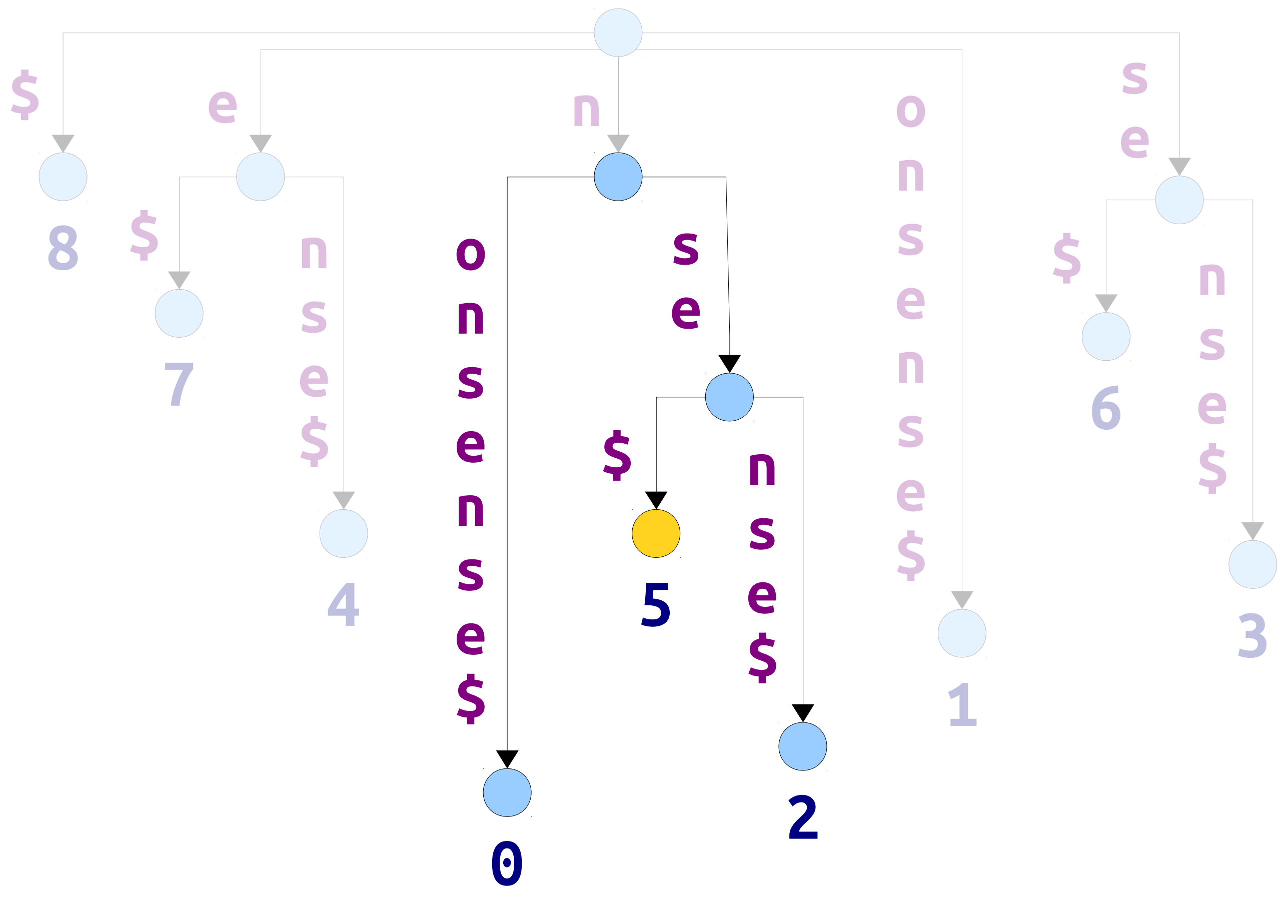
nonsense$
012345678



8	$
7	e$
4	ense$
0	nonsense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>n</u>ons

Searching a Suffix Array

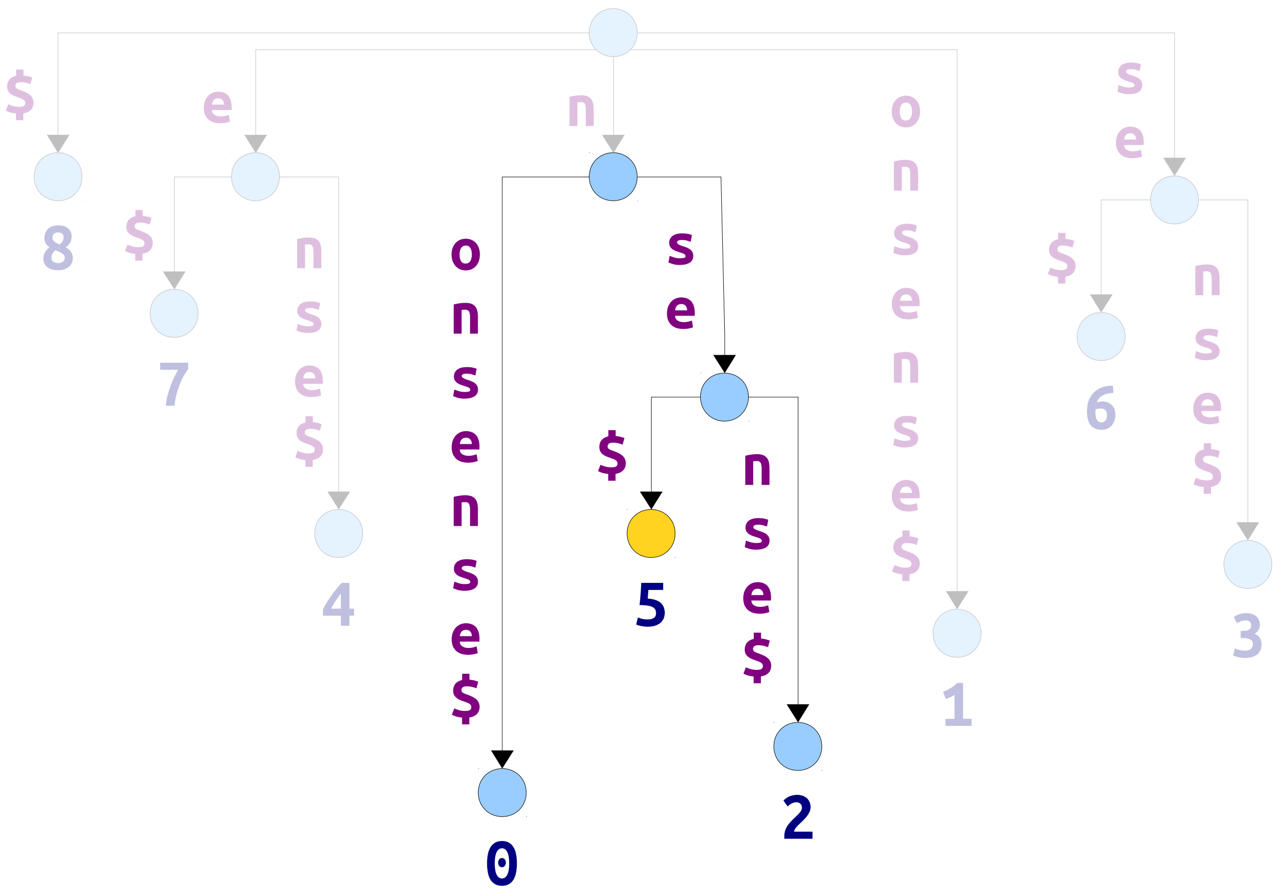


nonsense$
012345678

8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

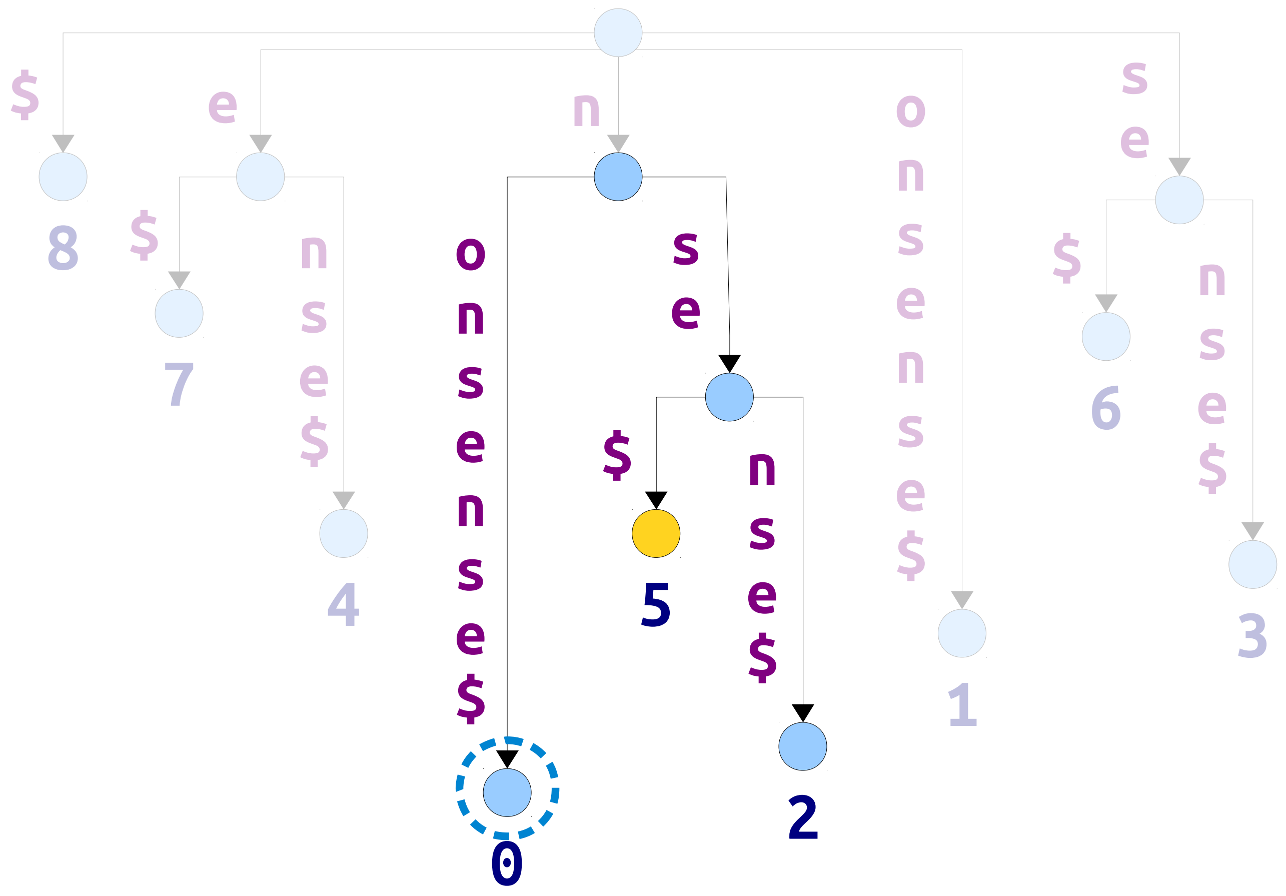


nonsense$
012345678

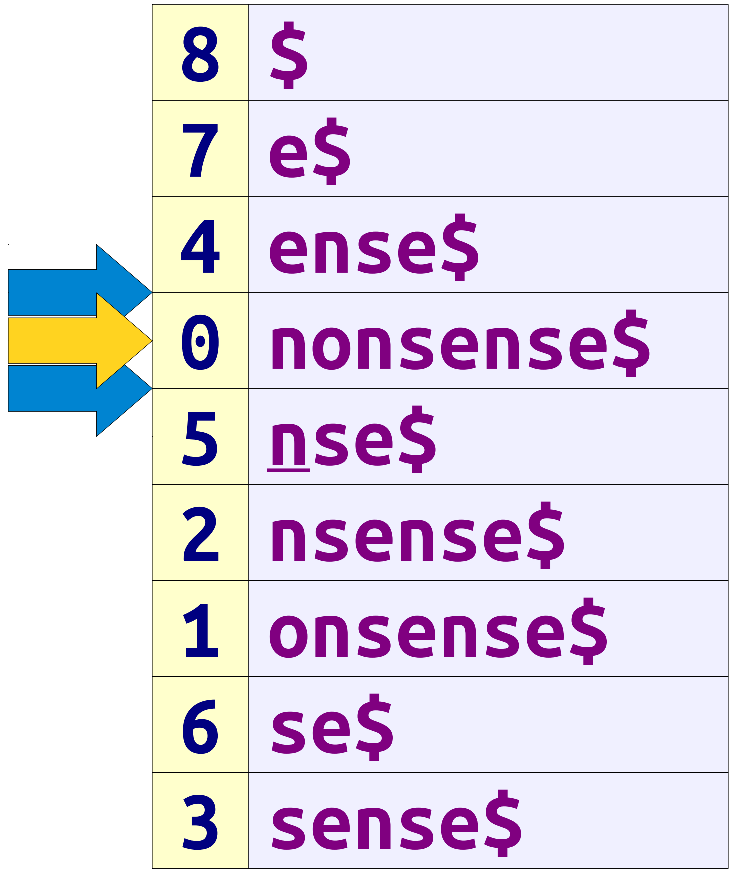
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



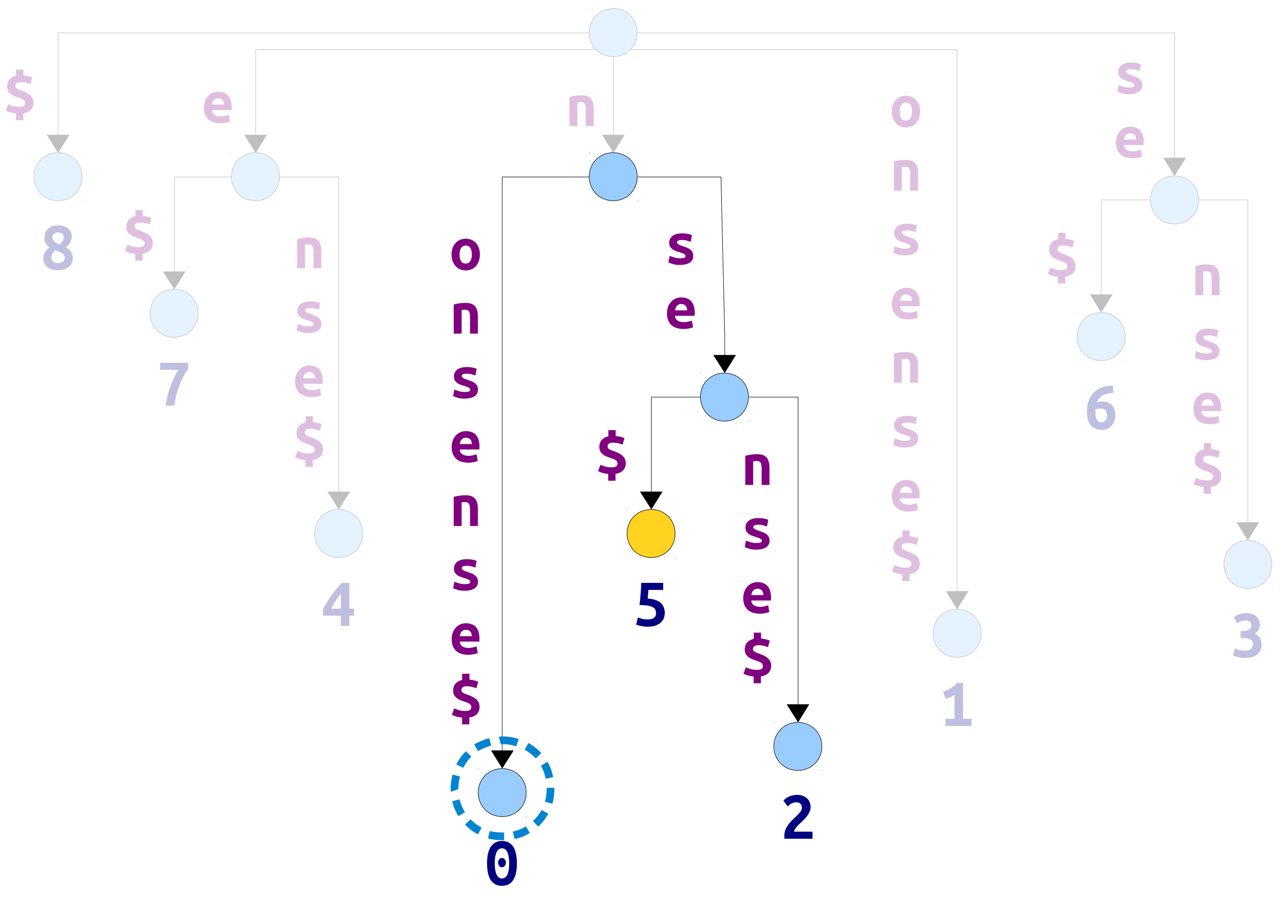
nonsense$
012345678



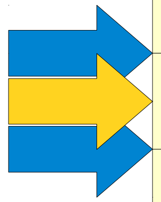
8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array



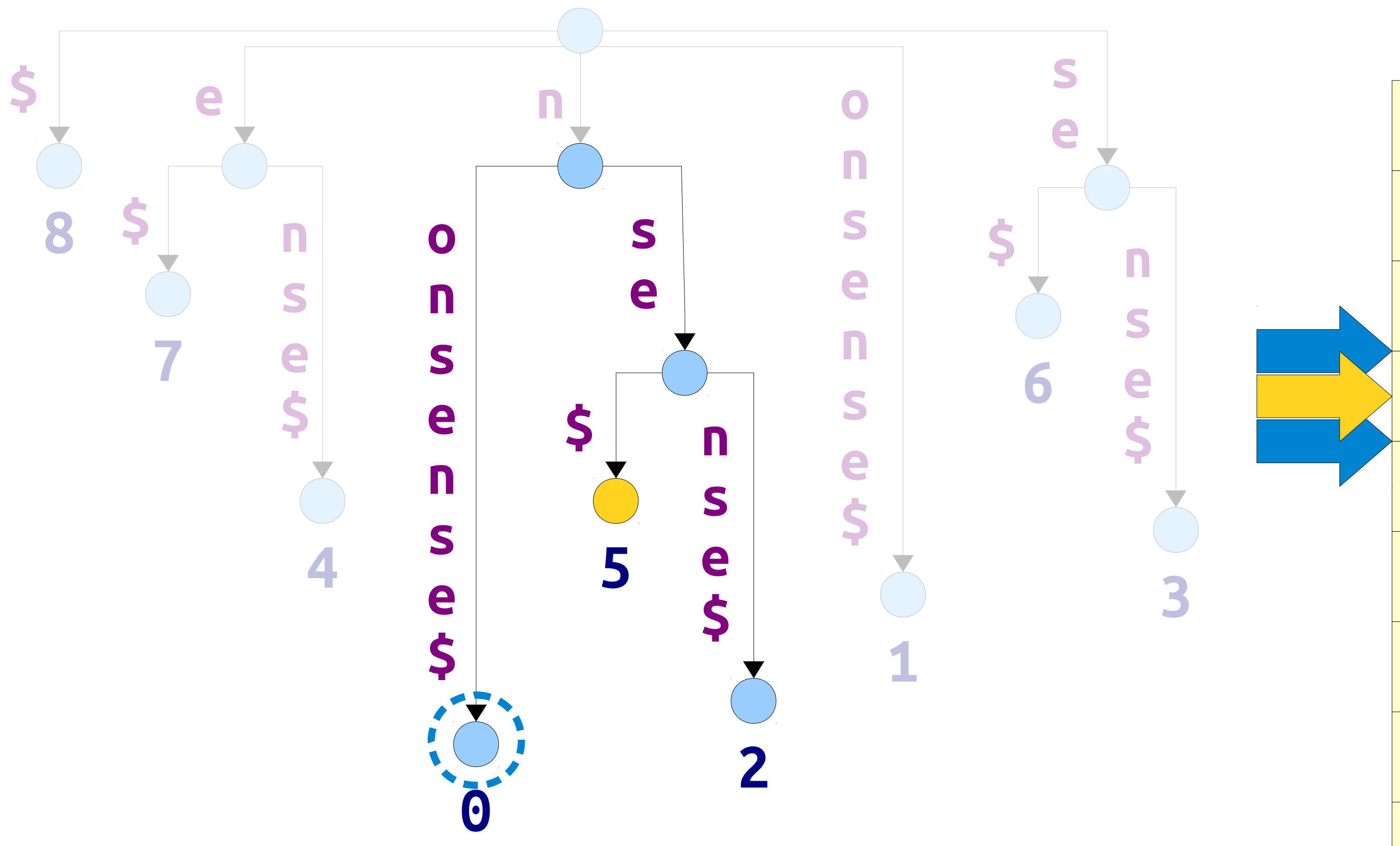
nonsense$
012345678



8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

nons

Searching a Suffix Array

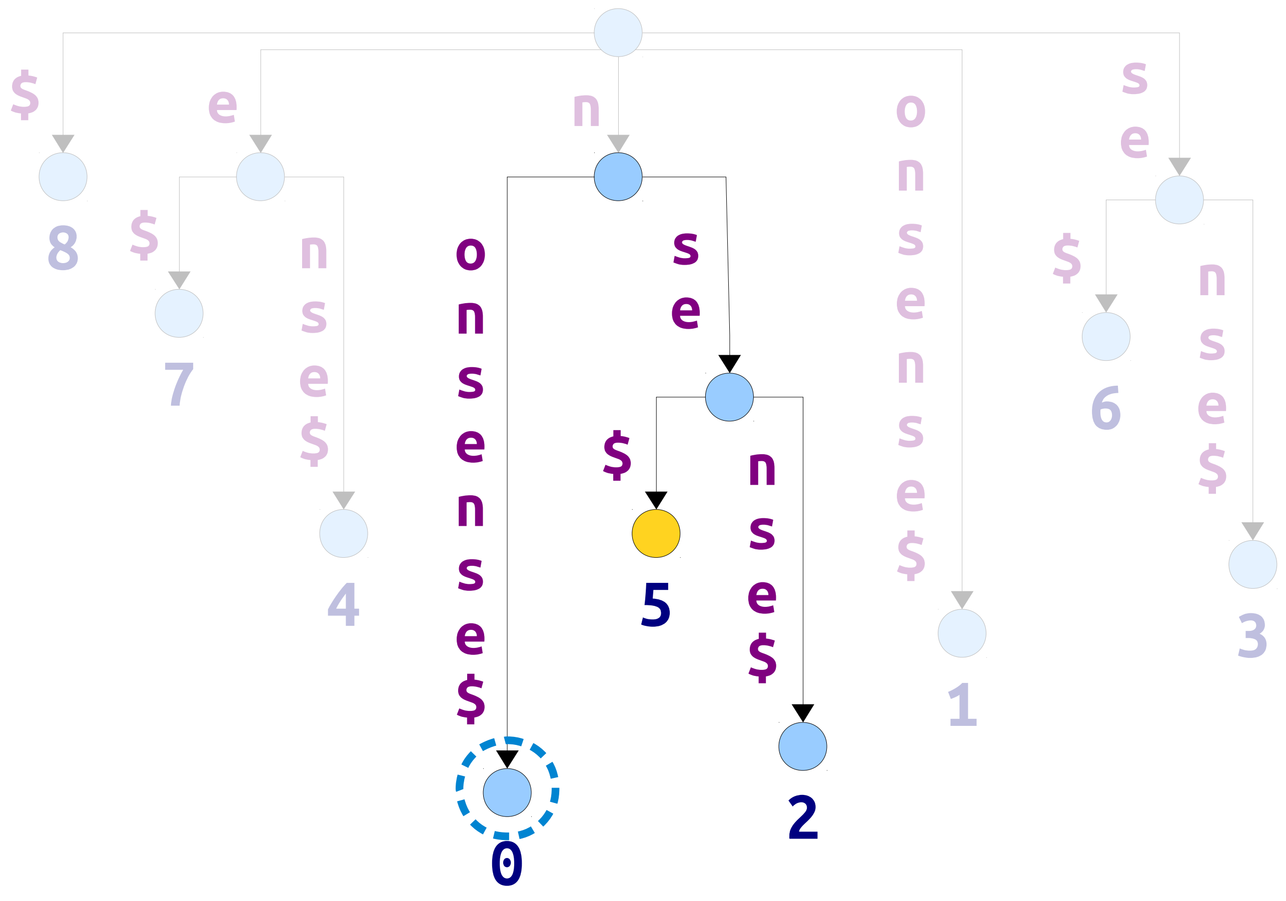


8	$
7	e$
4	ense$
0	nonsense$
5	nse$
2	nsense$
1	onsense$
6	se$
3	sense$

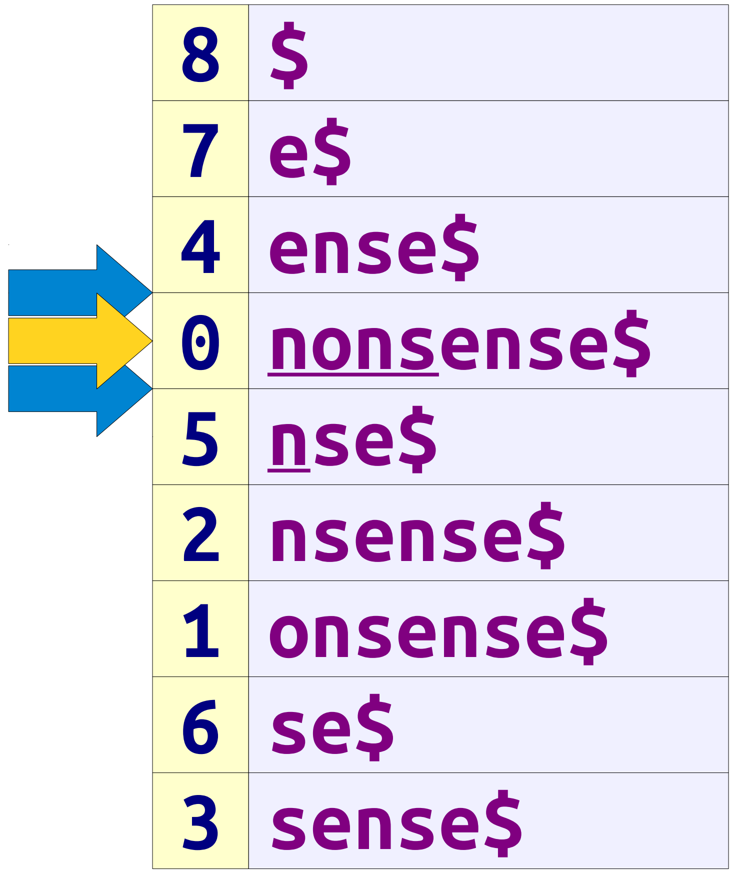
nonsense$
012345678

nons

Searching a Suffix Array



nonsense$
012345678



8	$
7	e$
4	ense$
0	<u>nons</u>ense$
5	<u>n</u>se$
2	nsense$
1	onsense$
6	se$
3	sense$

<u>nons</u>

Searching a Suffix Array

- Intuitively, simulate doing a binary search of the leaves of a suffix tree, remembering the deepest subtree you've matched so far.
- At each point, if the binary search probes a leaf outside of the current subtree, skip it and continue the binary search in the direction of the current subtree.
- To implement this on an actual suffix array, we use LCP information to implicitly keep track of where the bounds on the current subtree are.

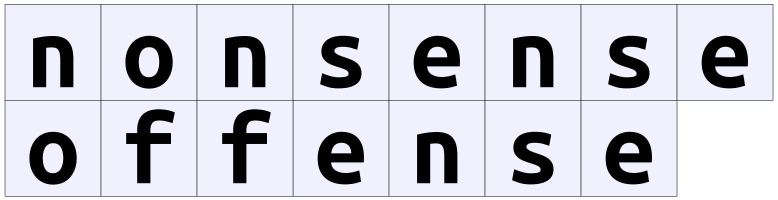
Searching a Suffix Array

- ***Claim:*** The algorithm we just sketched runs in time O(n + log m + z).
- ***Proof Sketch:*** The O(log m) term comes from the binary search over the leaves of the suffix tree. The O(n) term corresponds to descending deeper into the suffix tree one character at a time. Finally, we have to spend O(z) time reporting matches. ■

Applications:
Longest Common Extensions

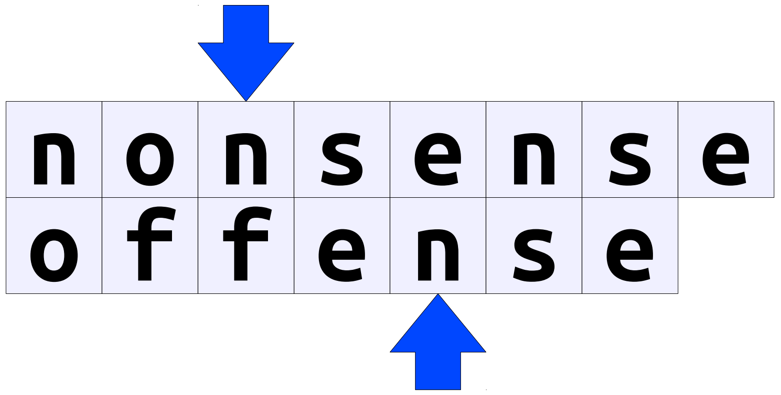
Longest Common Extensions

- 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.



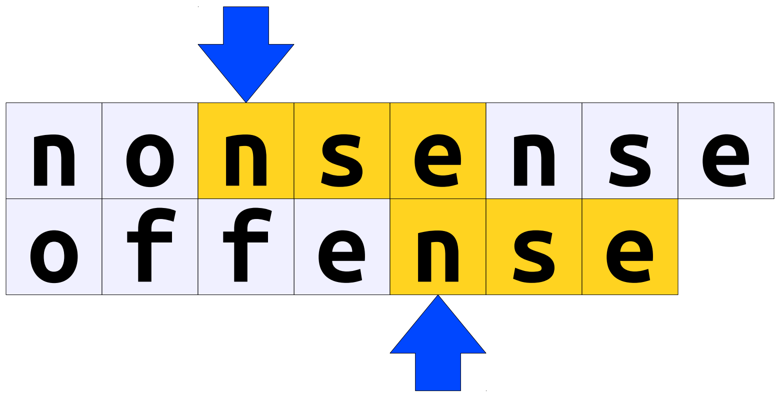
Longest Common Extensions

- 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.



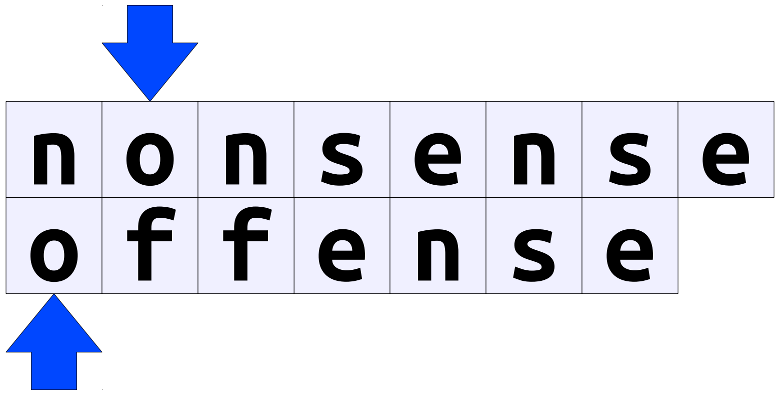
Longest Common Extensions

- 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.



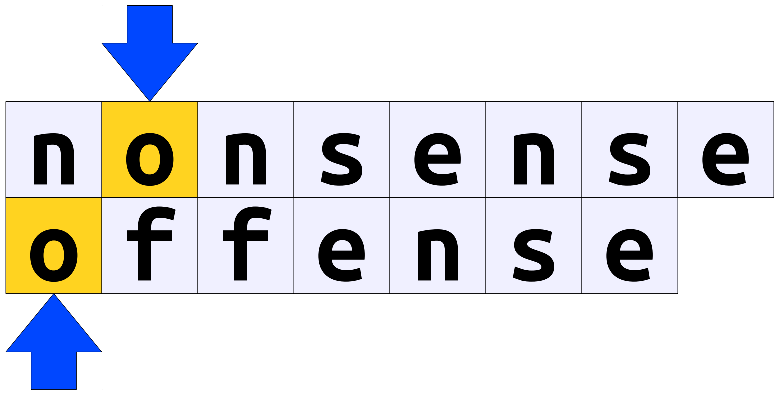
Longest Common Extensions

- 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.



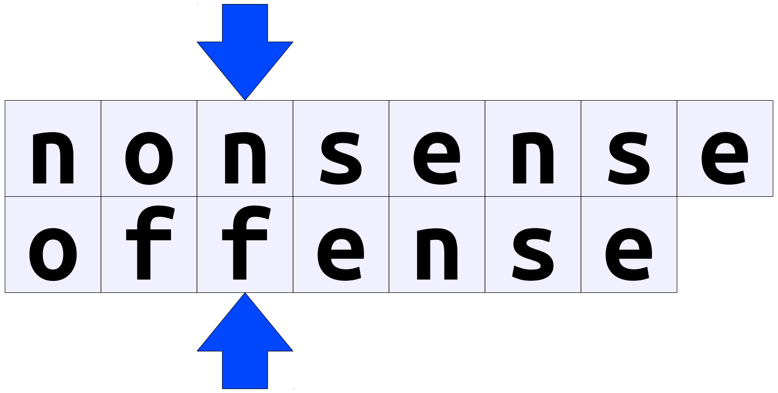
Longest Common Extensions

- 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 $\mathrm{LCE}_{T_1, T_2}(i, j)$.
- Typically, T_1 and T_2 are fixed and multiple (i, j) queries are specified.



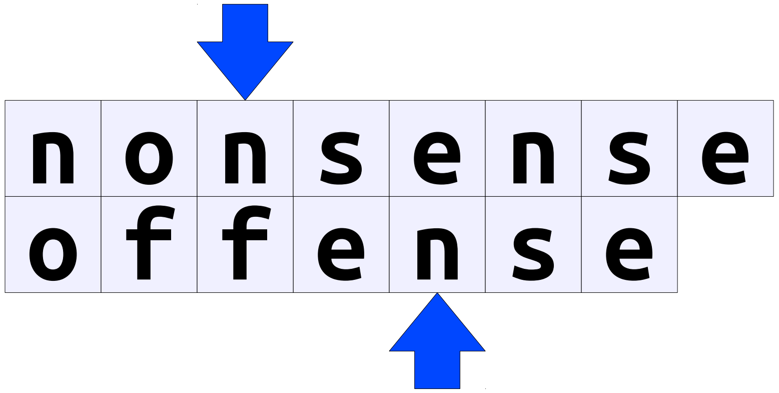
Longest Common Extensions

- 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.



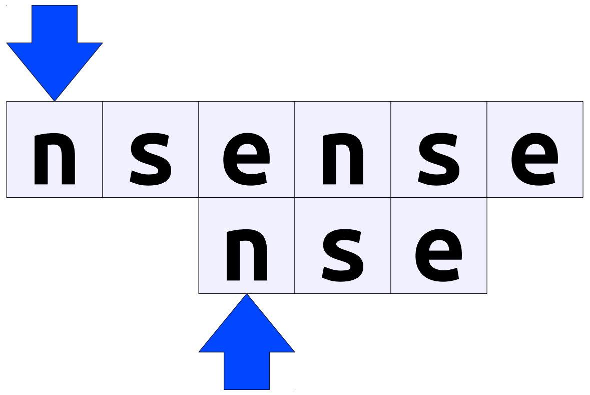
Longest Common Extensions

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



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.



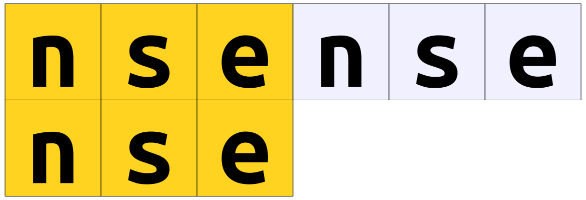
Longest Common Extensions

- ***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			

Longest Common Extensions

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



Suffix Arrays and LCE

- ***Claim:*** There is an ⟨O(m), O(1)⟩ data structure for LCE.
- Preprocessing:
  - Construct a ***generalized suffix array*** for T_1 and T_2 augmented with LCP information.
    - (Just build a suffix array for $T_1\$_1T_2\$_2$.)
  - Then build a table mapping each index in the string to its index in the suffix array.
- Query:
  - Do an RMQ over the LCP array at the appropriate indices.

0
0
1
1
4
0
1
3
3
0
0
2
2
0

8_1	$\$_1$
5_2	$\$_2$
7_1	e$\$_1$
4_2	e$\$_2$
4_1	ense$\$_1$
1_2	ense$\$_2$
0_1	nonsense$\$_1$
5_1	nse$\$_1$
2_2	nse$\$_2$
2_1	nsense$\$_1$
1_1	onsense$\$_1$
6_1	se$\$_1$
3_2	se$\$_2$
3_1	sense$\$_1$
0_2	tense$\$_2$

1	nonsense$\$_2$
2	tense$\$_2$

Suffix Arrays and LCE

- ***Claim:*** There is an ⟨O(m), O(1)⟩ data structure for LCE.
- Preprocessing:
  - Construct a ***generalized suffix array*** for T_1 and T_2 augmented with LCP information.
    - (Just build a suffix array for $T_1\$_1T_2\$_2$.)
  - Then build a table mapping each index in the string to its index in the suffix array.
- Query:
  - Do an RMQ over the LCP array at the appropriate indices.

	8_1	$\$_1$
0	5_2	$\$_2$
0	7_1	$e\$_1$
1	4_2	$e\$_2$
1	4_1	$ense\$_1$
4	1_2	$ense\$_2$
0	0_1	$nonsense\$_1$
1	5_1	$nse\$_1$
3	2_2	$nse\$_2$
3	2_1	$nsense\$_1$
0	1_1	$onsense\$_1$
0	6_1	$se\$_1$
2	3_2	$se\$_2$
2	3_1	$sense\$_1$
0	0_2	$tense\$_2$

1	$nonsense\$_2$
2	$tense\$_2$

Suffix Arrays and LCE

- ***Claim:*** There is an ⟨O(m), O(1)⟩ data structure for LCE.
- Preprocessing:
  - Construct a ***generalized suffix array*** for T_1 and T_2 augmented with LCP information.
    - (Just build a suffix array for $T_1\$_1T_2\$_2$.)
  - Then build a table mapping each index in the string to its index in the suffix array.
- Query:
  - Do an RMQ over the LCP array at the appropriate indices.

	8_1	$\$_1$
0	5_2	$\$_2$
0	7_1	e$\$_1$
1	4_2	e$\$_2$
1	4_1	ense$\$_1$
4	1_2	ense$\$_2$
0	0_1	nonsense$\$_1$
1	5_1	nse$\$_1$
3	2_2	nse$\$_2$
3	2_1	nsense$\$_1$
0	1_1	onsense$\$_1$
0	6_1	se$\$_1$
2	3_2	se$\$_2$
2	3_1	sense$\$_1$
0	0_2	tense$\$_2$

1	nonsense$\$_2$
2	tense$\$_2$

Suffix Arrays and LCE

- ***Claim:*** There is an ⟨O(m), O(1)⟩ data structure for LCE.
- Preprocessing:
  - Construct a ***generalized suffix array*** for T_1 and T_2 augmented with LCP information.
    - (Just build a suffix array for $T_1\$_1T_2\$_2$.)
  - Then build a table mapping each index in the string to its index in the suffix array.
- Query:
  - Do an RMQ over the LCP array at the appropriate indices.

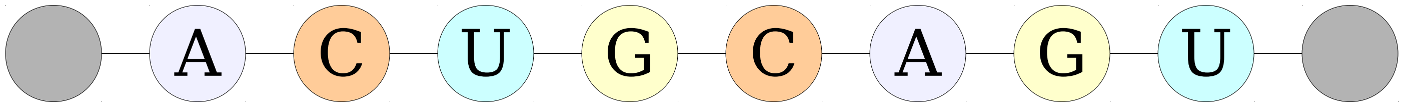
	8_1	$\$_1$
0	5_2	$\$_2$
0	7_1	e$\$_1$
1	4_2	e$\$_2$
1	4_1	ense$\$_1$
4	1_2	ense$\$_2$
0	0_1	nonsense$\$_1$
1	5_1	nse$\$_1$
3	2_2	nse$\$_2$
3	2_1	nsense$\$_1$
0	1_1	onsense$\$_1$
0	6_1	se$\$_1$
2	3_2	se$\$_2$
2	3_1	sense$\$_1$
0	0_2	tense$\$_2$

1	nonsense$\$_2$
2	tense$\$_2$

An Application: Longest Palindromic Substring

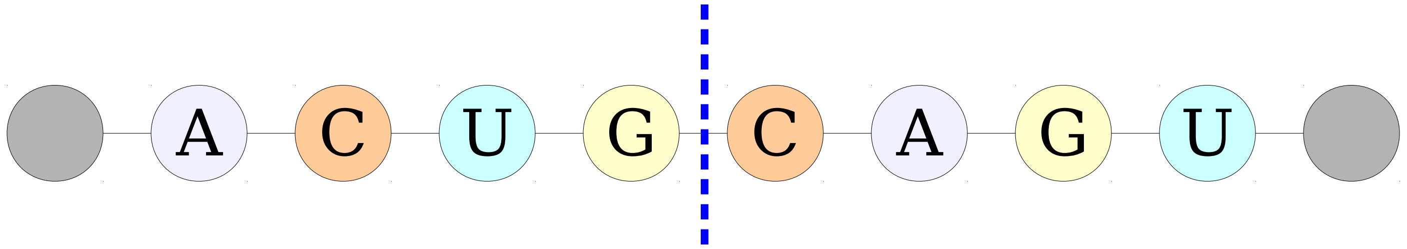
Palindromes

- 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.



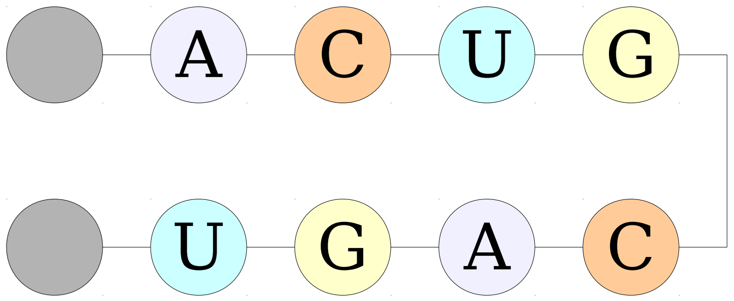
Palindromes

- 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.



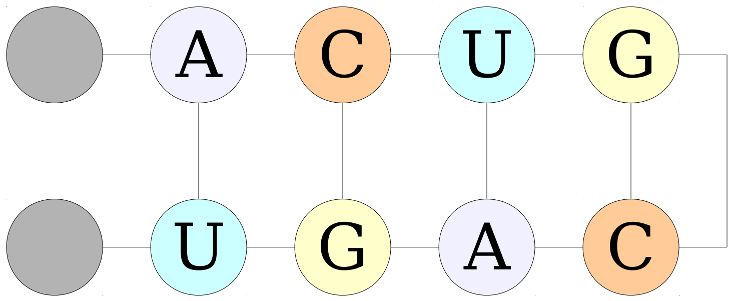
Palindromes

- 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:
  - $T =$ **abcdba**
  - $T^R =$ **abdcba**
  - Longest common substring: **ab** / **ba**
  - Longest palindromic substring: **a** / **b** / **c** / **d**

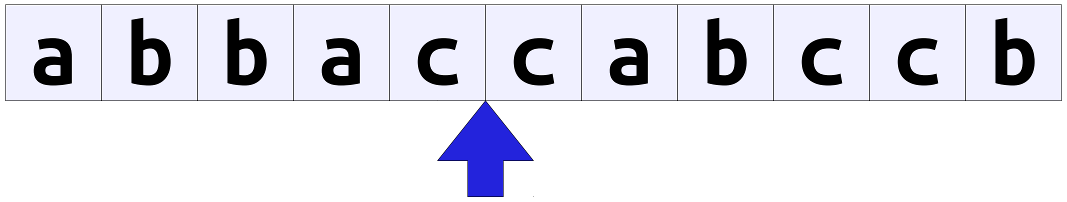
Palindrome Centers and Radii

- 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.

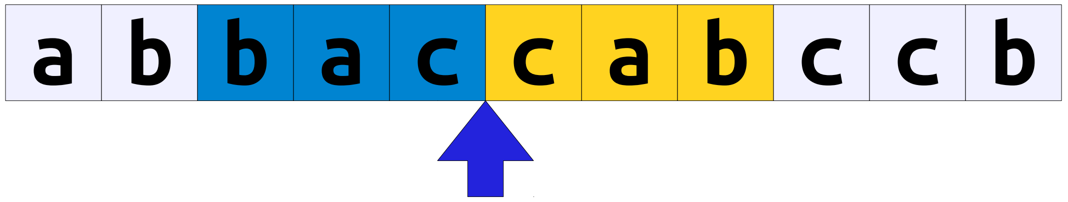
Palindrome Centers and Radii

a	b	b	a	c	c	a	b	c	c	b

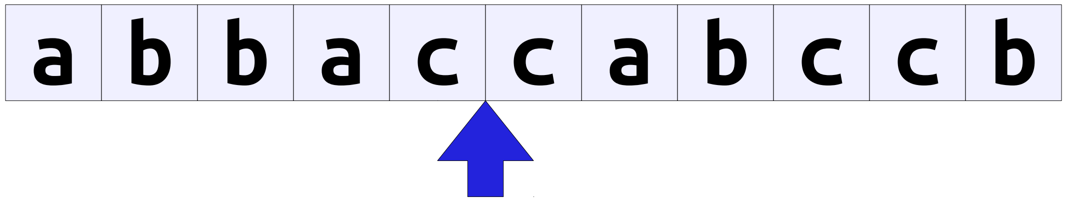
Palindrome Centers and Radii



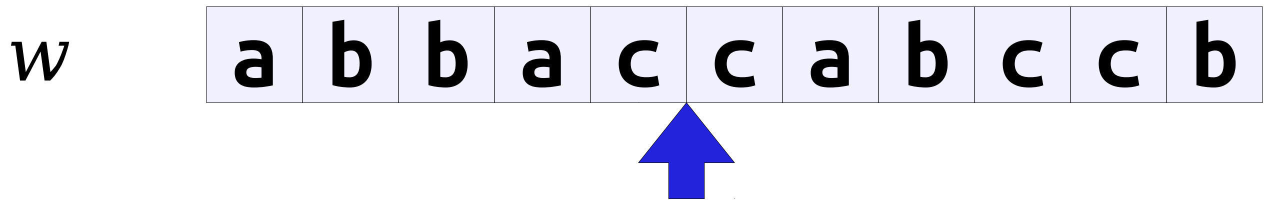
Palindrome Centers and Radii



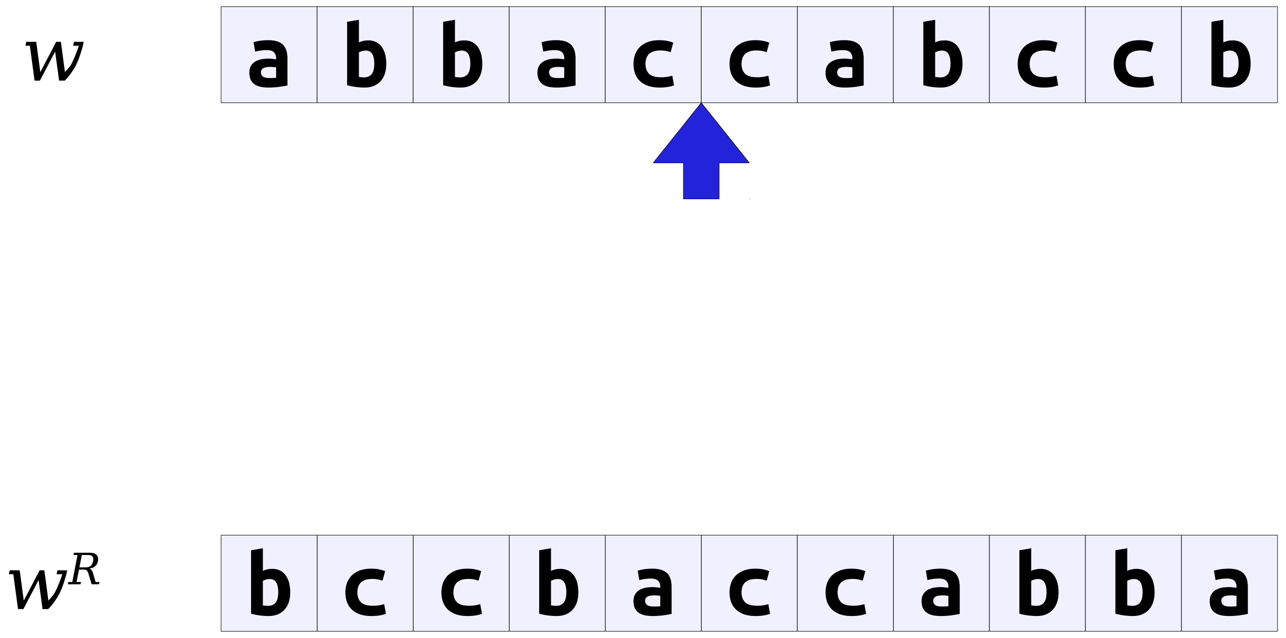
Palindrome Centers and Radii



Palindrome Centers and Radii



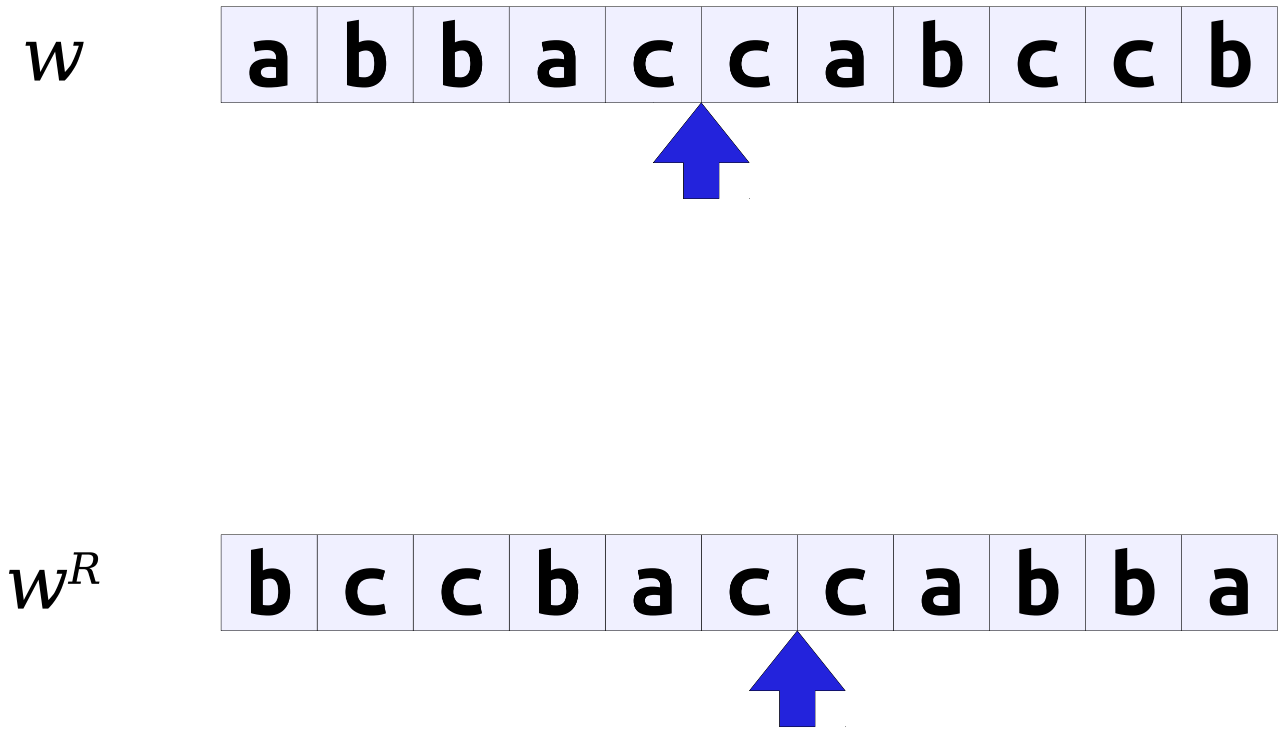
Palindrome Centers and Radii



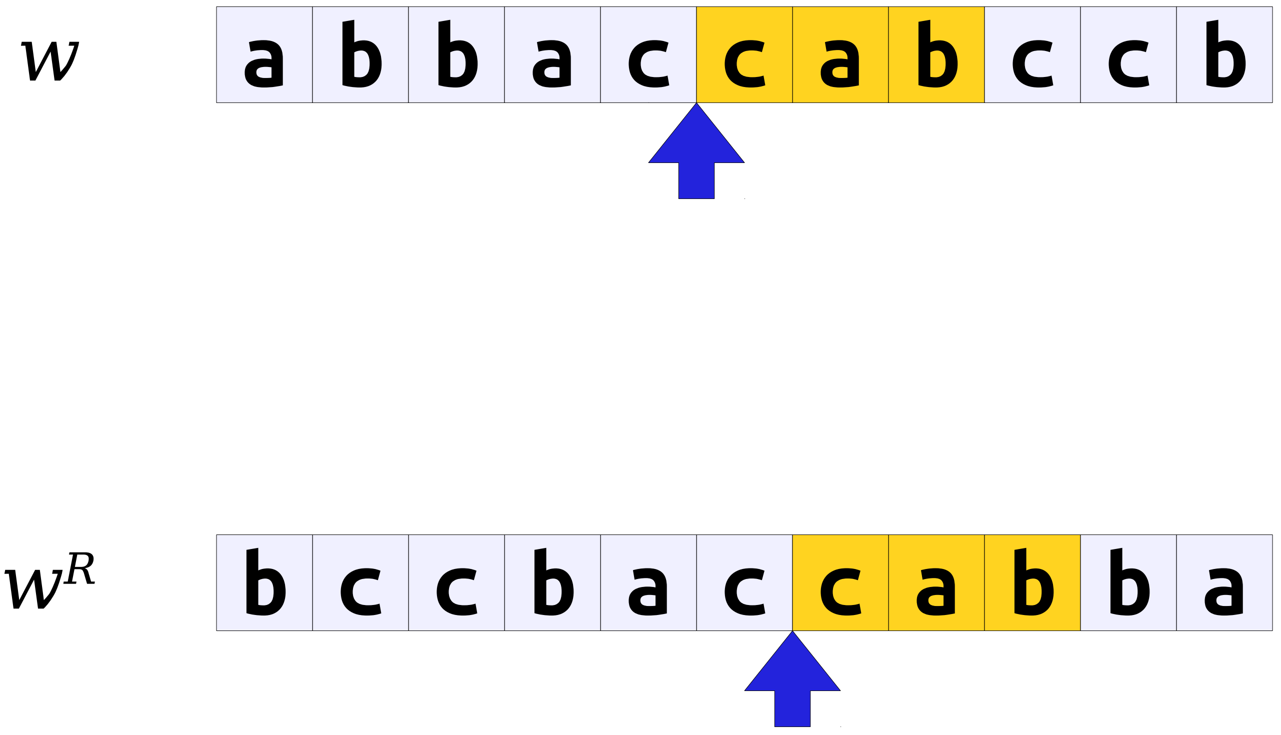
w	a	b	b	a	c	c	a	b	c	c	b

w^R	b	c	c	b	a	c	c	a	b	b	a

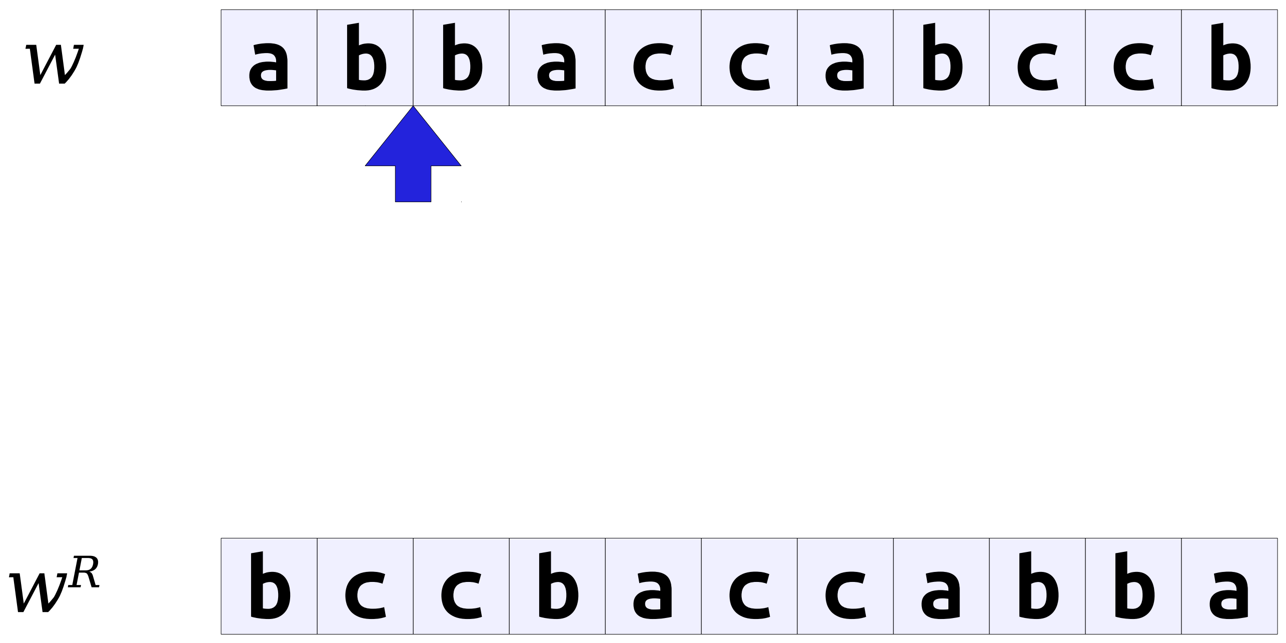
Palindrome Centers and Radii



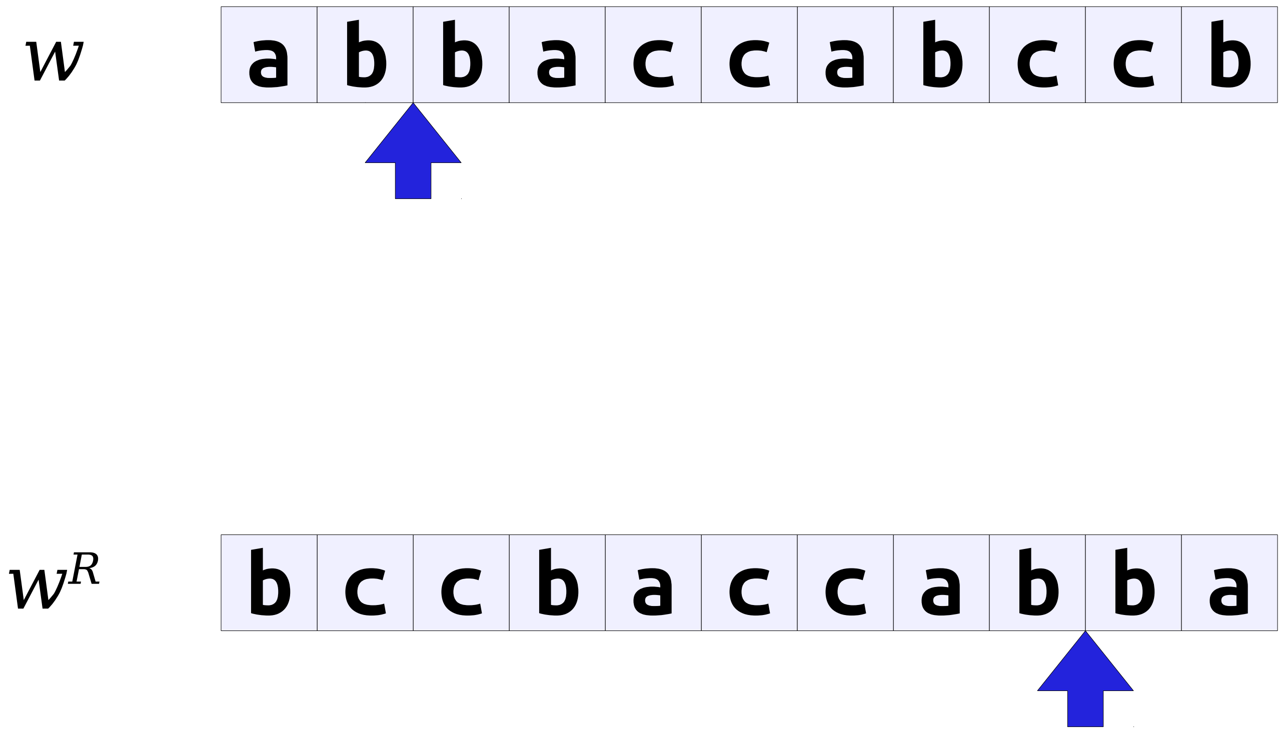
Palindrome Centers and Radii



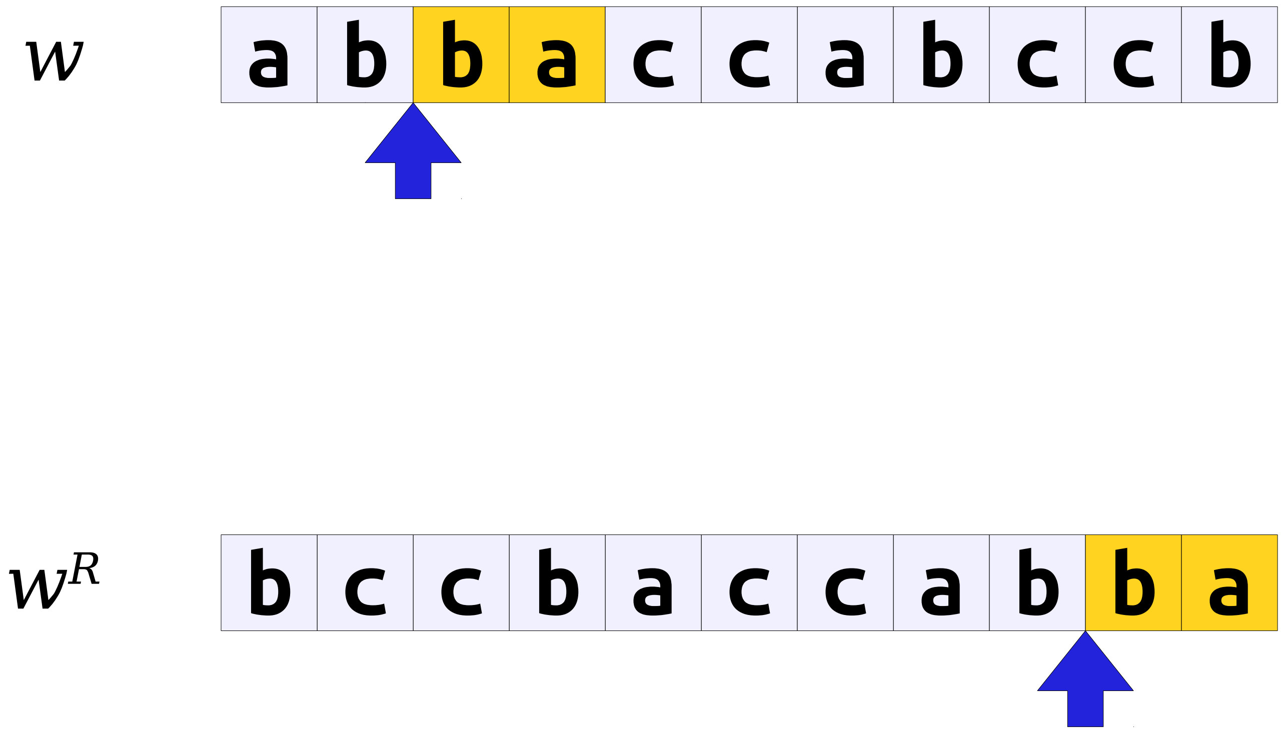
Palindrome Centers and Radii



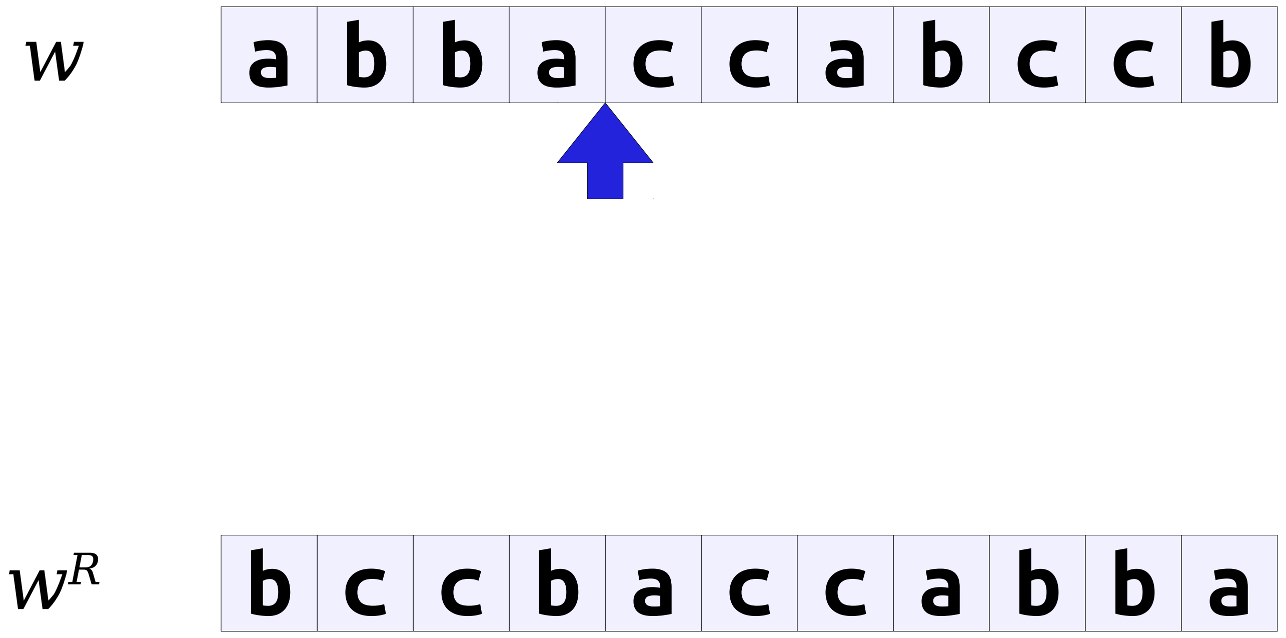
Palindrome Centers and Radii



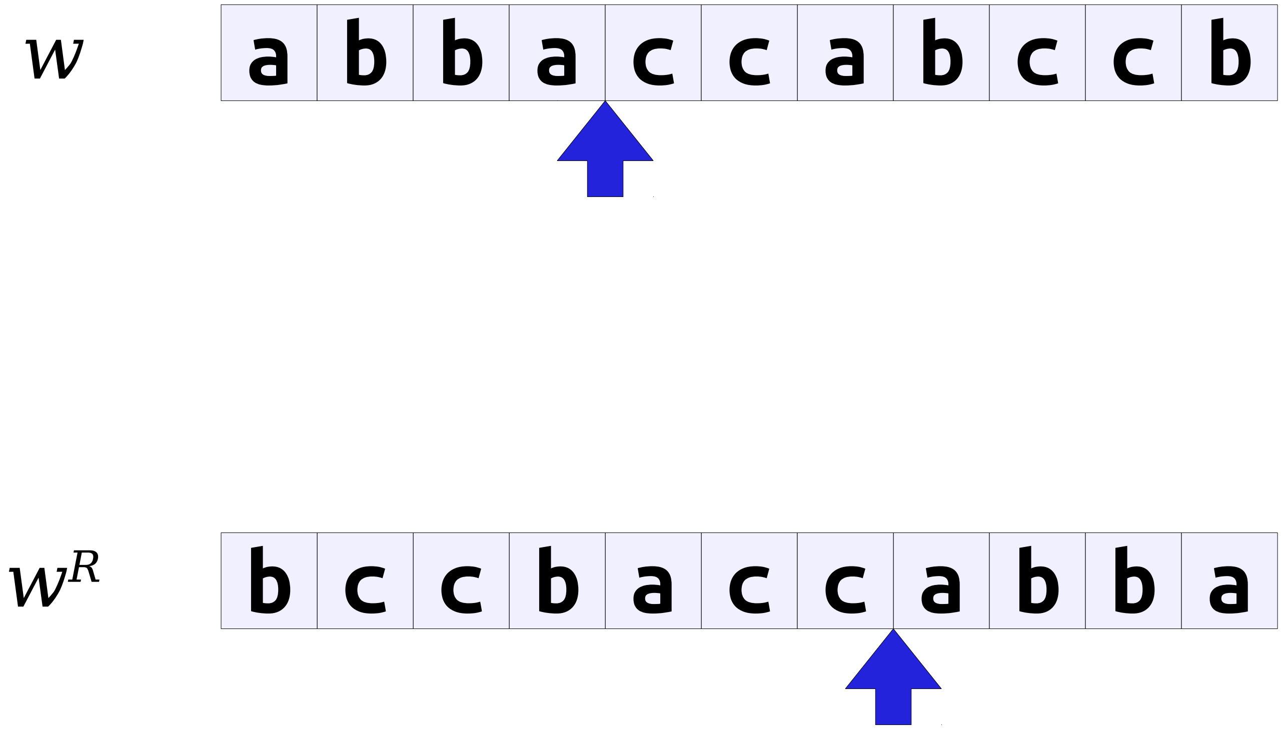
Palindrome Centers and Radii



Palindrome Centers and Radii



Palindrome Centers and Radii



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)**.

Next Time

- ***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).