Suffix Trees

String Data Structures

- Our next three lectures are all on the wonderful world of string data structures.
- Why are they worth studying?
  - ***They're practical***. These data structures were developed to meet practical needs in data processing.
  - ***They're algorithmically interesting***. The techniques that power these data structures involve some truly beautiful connections and observations.
- We aren't going to be able to cover everything worth covering. Think of this as a launching point, not a high water mark.

Where We're Going

- Today, we'll cover ***tries*** and ***suffix trees***, two powerful data structures for exposing shared structures in strings.
- On Thursday, we'll see the ***suffix array*** and ***LCP array***, which are a more space-efficient way of encoding suffix trees.
- Next Tuesday, we'll see the ***SA-IS algorithm***, which quickly builds suffix trees and suffix arrays, and is probably the most beautiful divide and conquer algorithm ever invented.

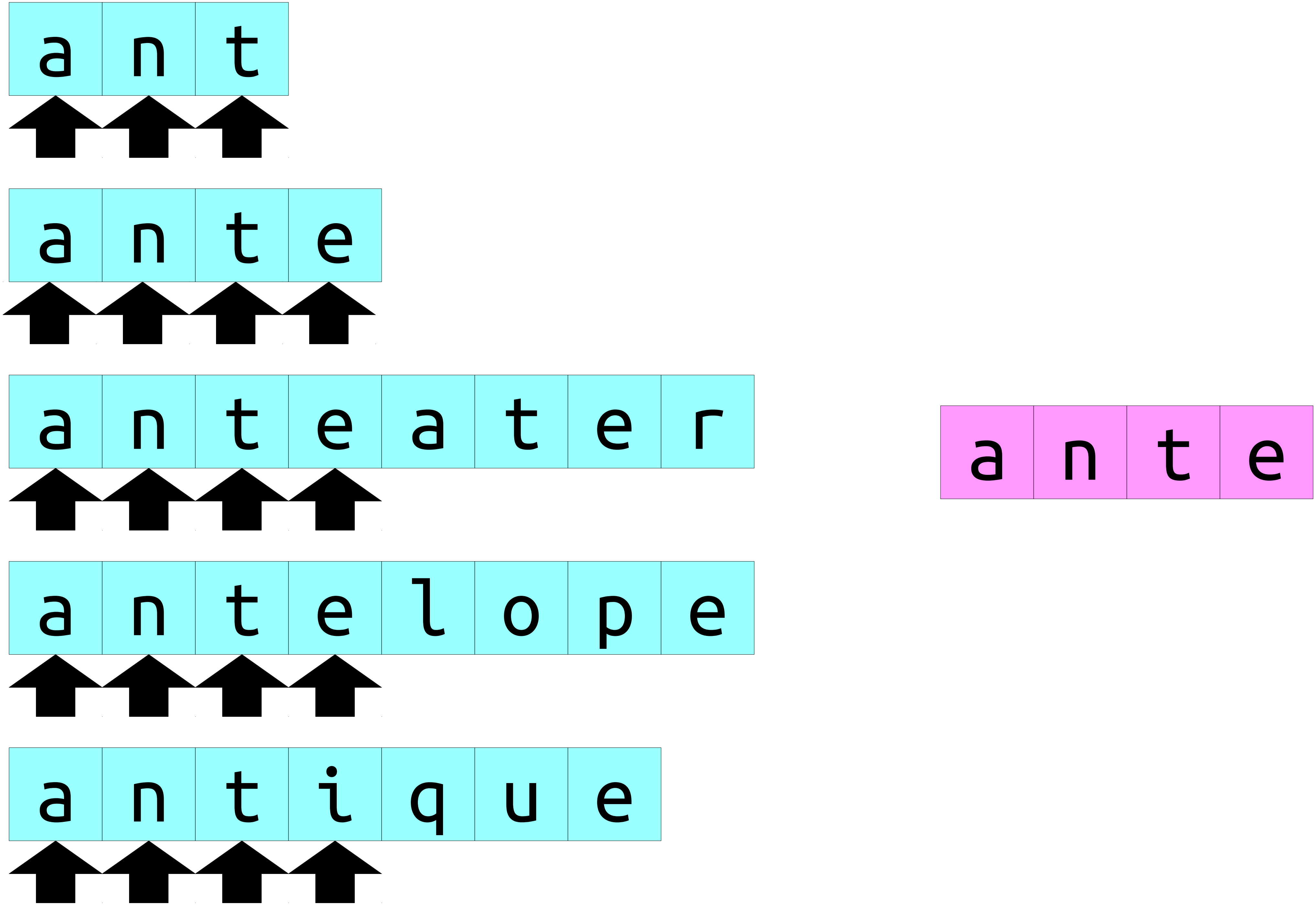
A Motivating Problem

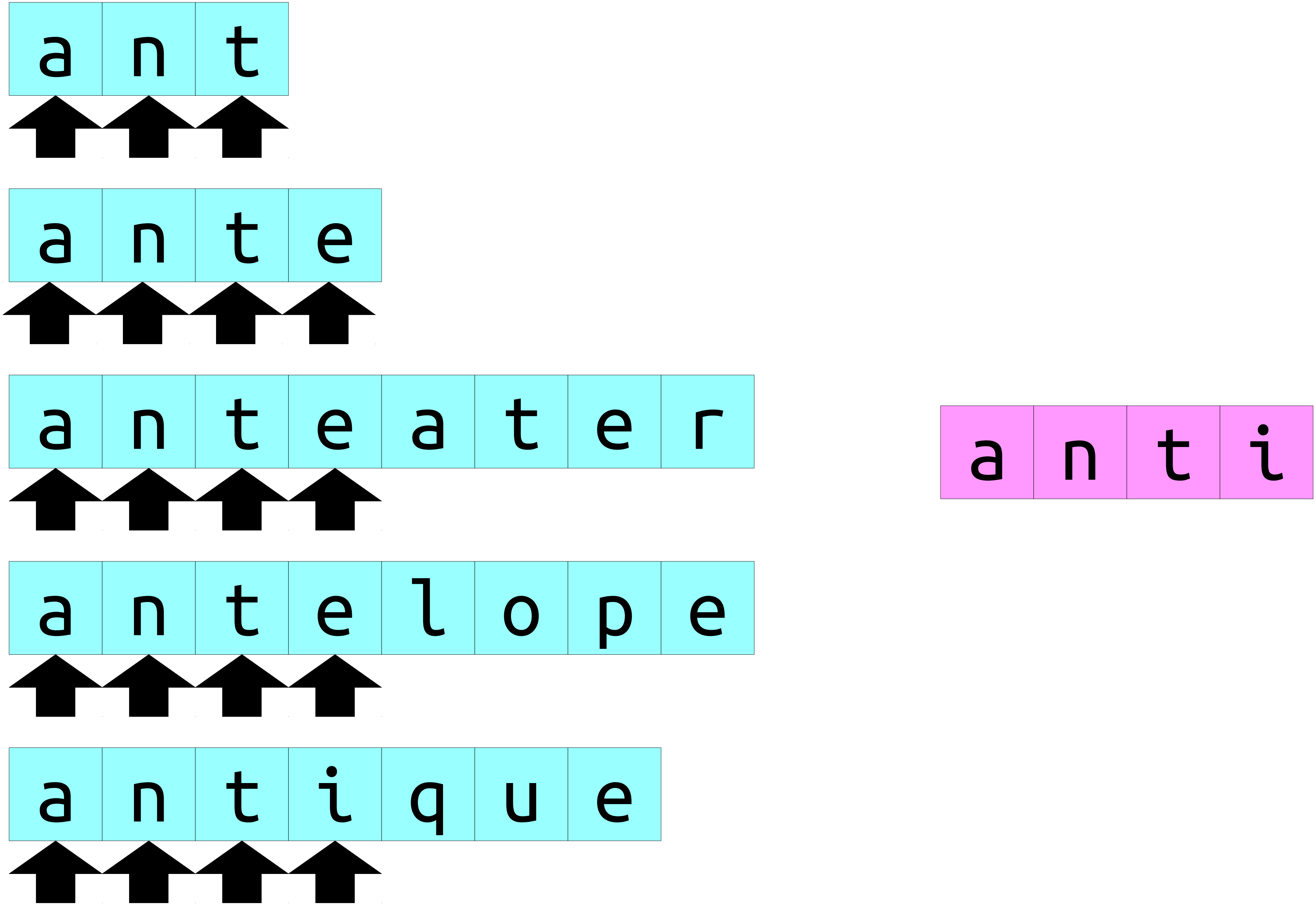
How is this done so quickly?

The Autocomplete Problem

- We have a series of text strings T_1, T_2, …, T_k of total length m.
- We have a pattern string P of length n.
- ***Goal:*** Find all text strings that start with P.
- If we just do a single query, then we can solve this pretty easily.
  - Just scan over all the strings and see which ones start with m.
- ***Question:*** If we have a set of fixed text strings and varying patterns, can we speed this up?

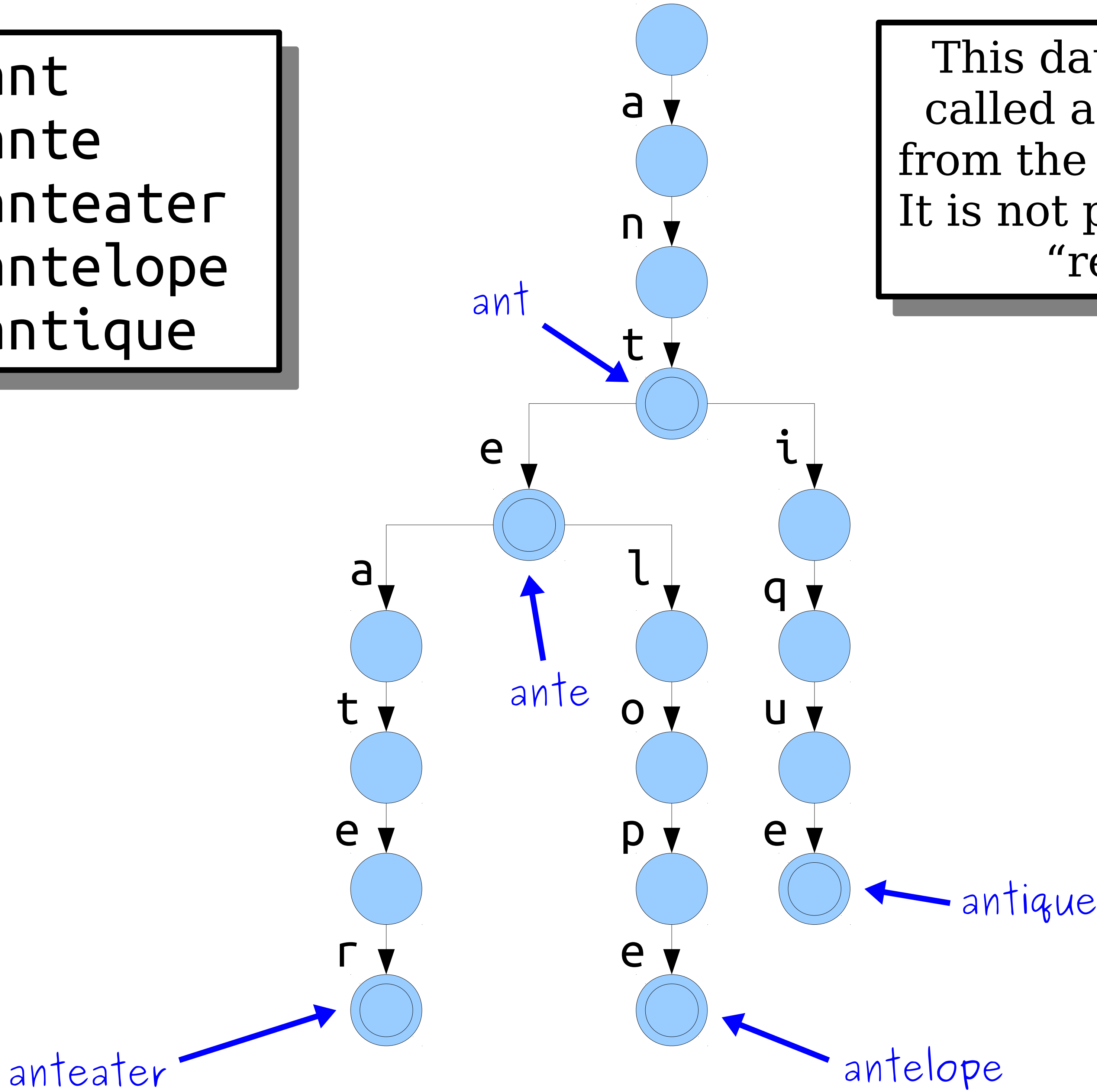
A Naive Solution


ant
ante
anteater
antelope
antique
ante



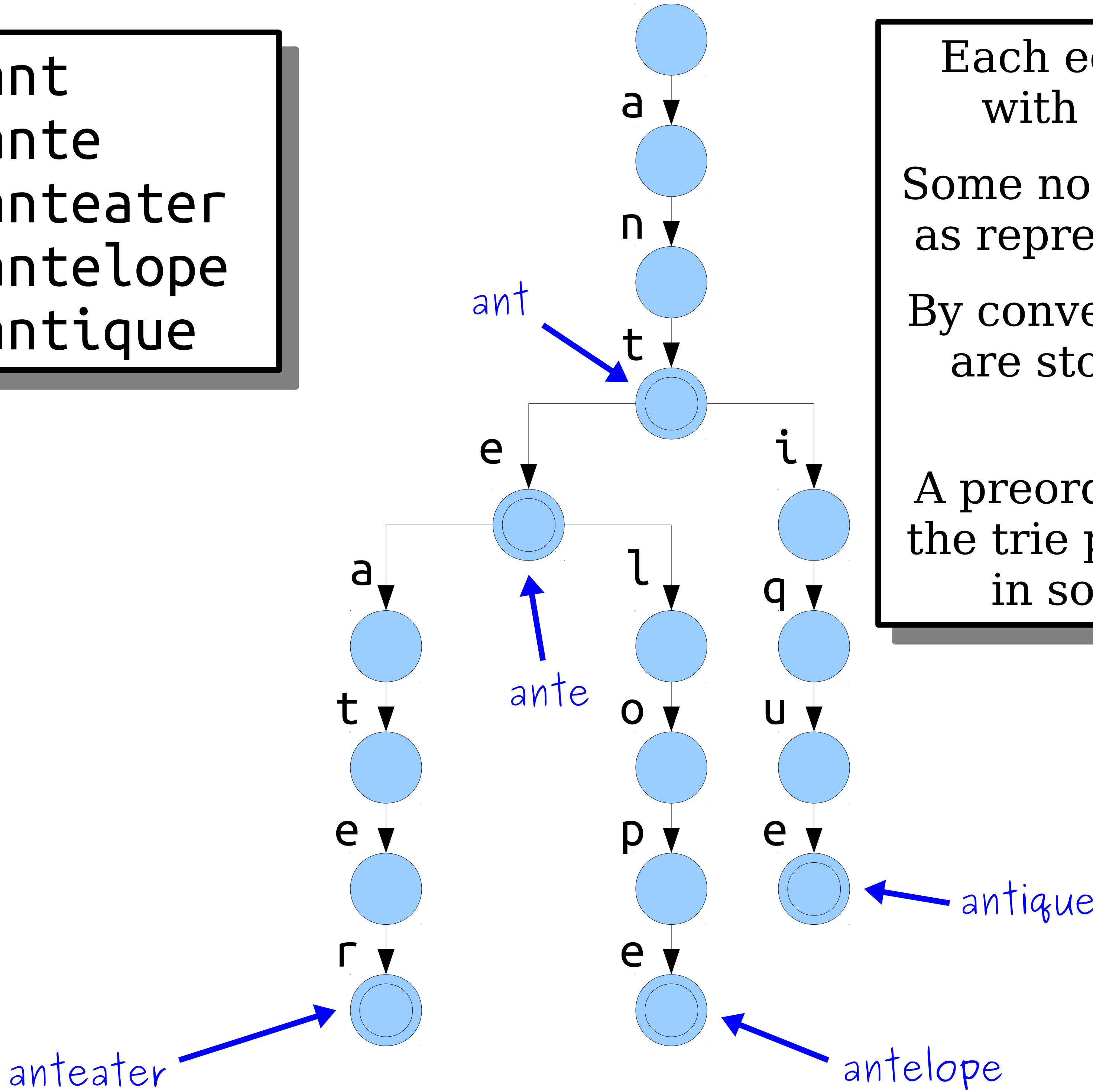
a n t
a n t e
a n t e a t e r
a n t e l o p e
a n t i q u e
a n t i

ant
ante
anteater
antelope
antique



This data structure is called a ***trie***. It comes from the word re***trie***val. It is not pronounced like "retrieval."

ant
ante
anteater
antelope
antique



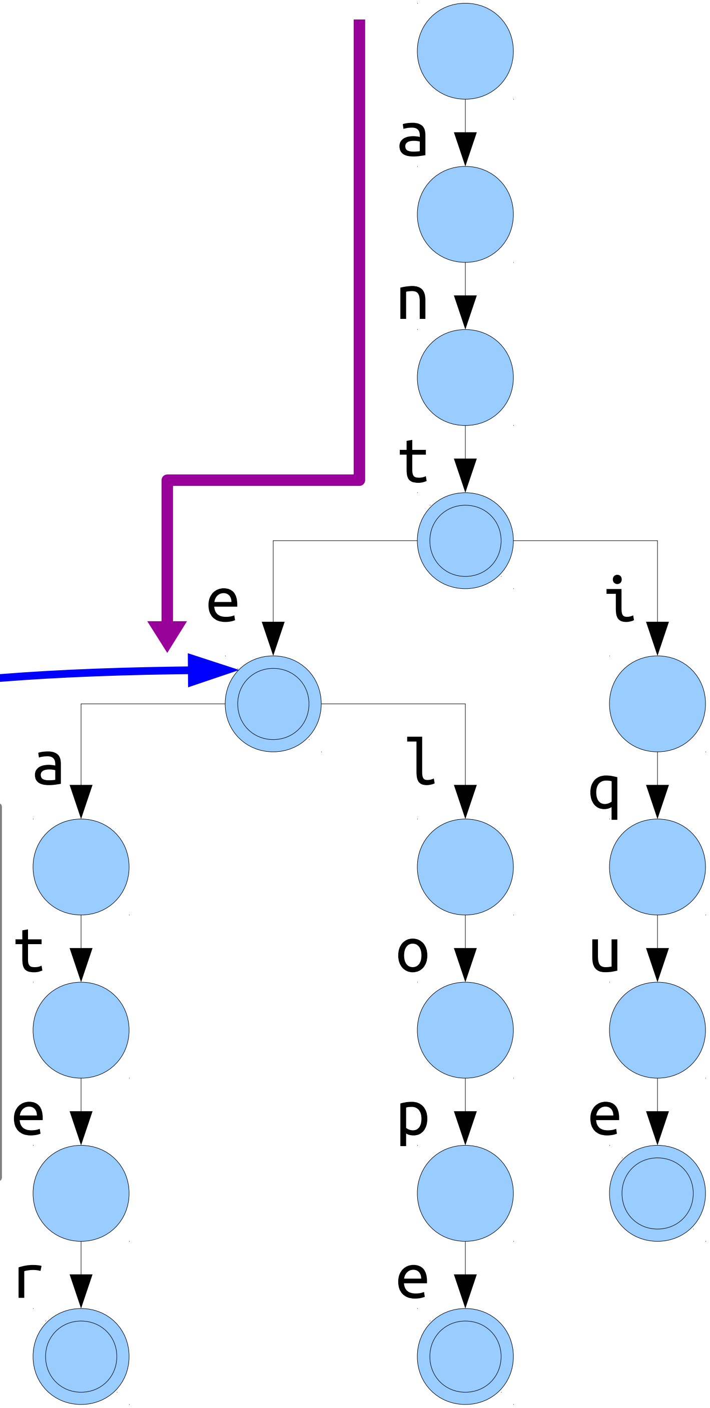
Each edge is labeled with a character.

Some nodes are marked as representing words.

By convention, children are stored in sorted order.

A preorder traversal of the trie prints all words in sorted order.

ant
ante
anteater
antelope
antique

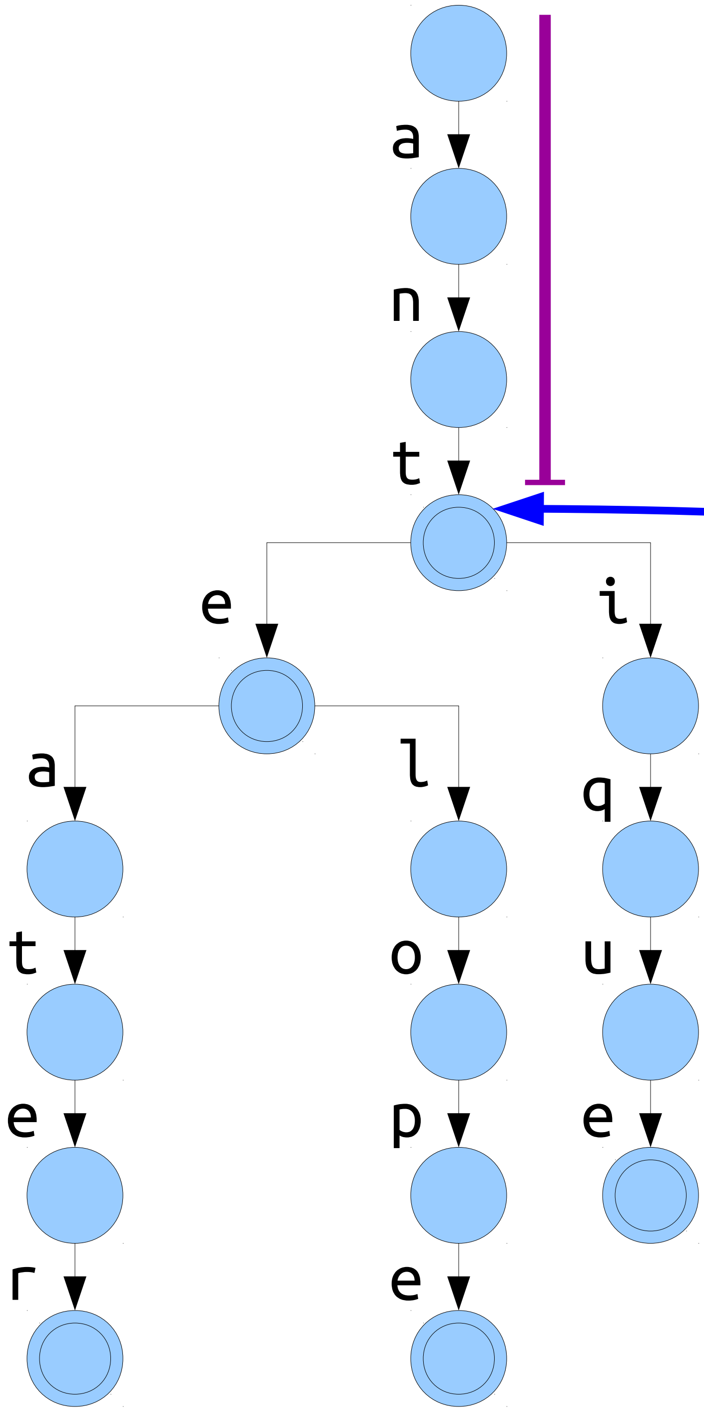


a	n	t	e

Now, do a DFS to find all words rooted here.

ant
ante
anteater
antelope
antique

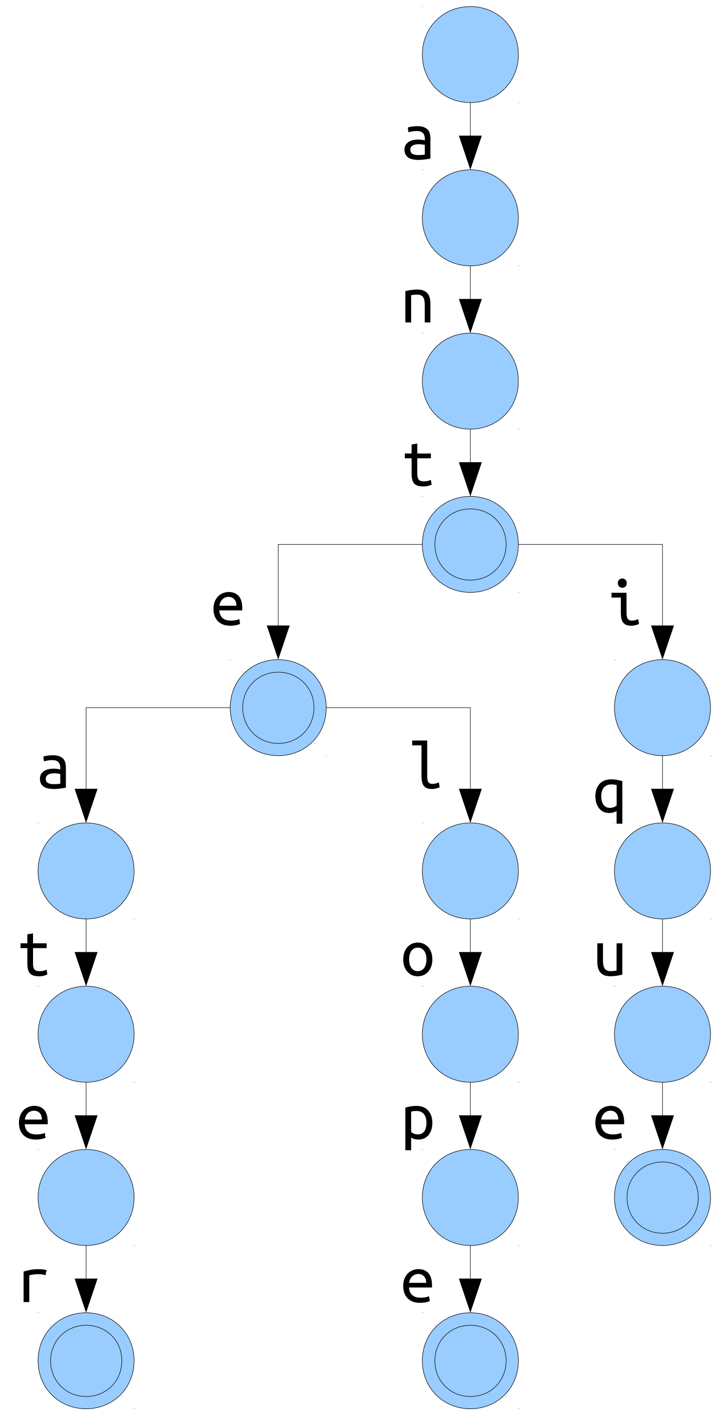
a	n	t	w



We fell off the trie. There are no matches!

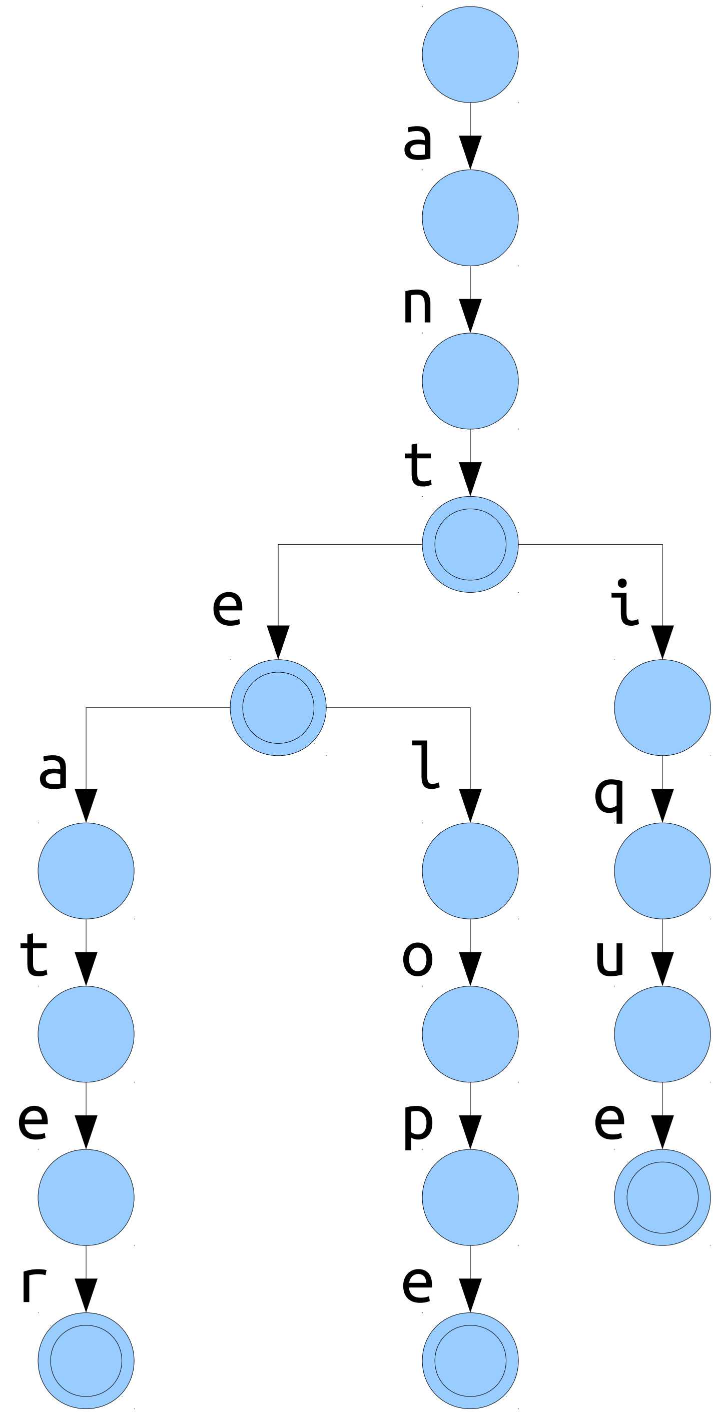
Tries

- ***Question:*** How do we encode the collection of child pointers?
  - Could use an array with one slot per possible character.
  - Could use a BST keyed by character.
  - Could use a hash table.
- For today, we'll ignore any terms that depend on the size of the alphabet.
- ***These terms matter!*** Hypothetically speaking, you might want to think about this a bit. ☺



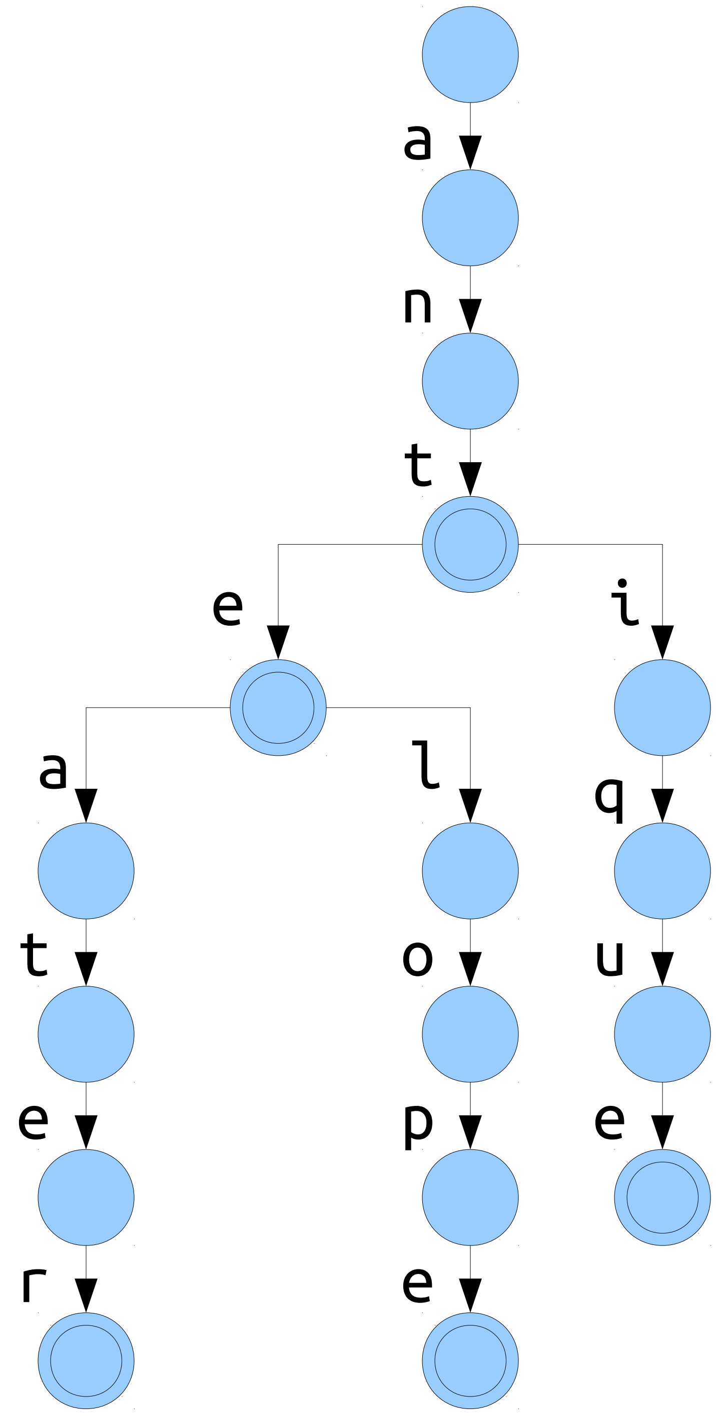
Tries

- ***Recall:*** The total length of our text strings is $\boldsymbol{m}$, and the length of our pattern string is $\boldsymbol{n}$.
- How long does it take to build our trie?
- ***Claim:*** Ignoring the size of the alphabet, the runtime is O($\boldsymbol{m}$).



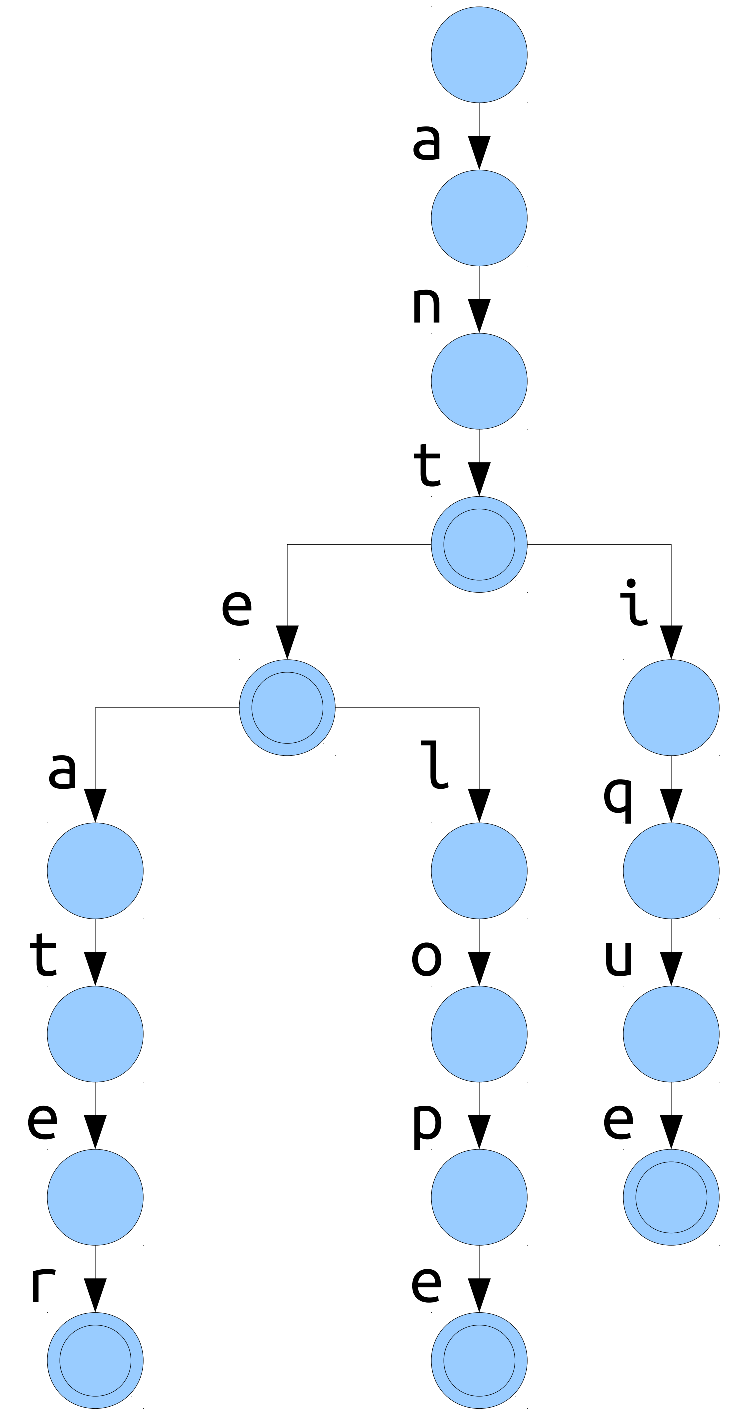
Tries

- ***Recall:*** The total length of our text strings is ***m***, and the length of our pattern string is ***n***.
- How long does it take to check if the pattern is a prefix of any string?
- ***Claim:*** Ignoring the size of the alphabet, the runtime is O(***n***).



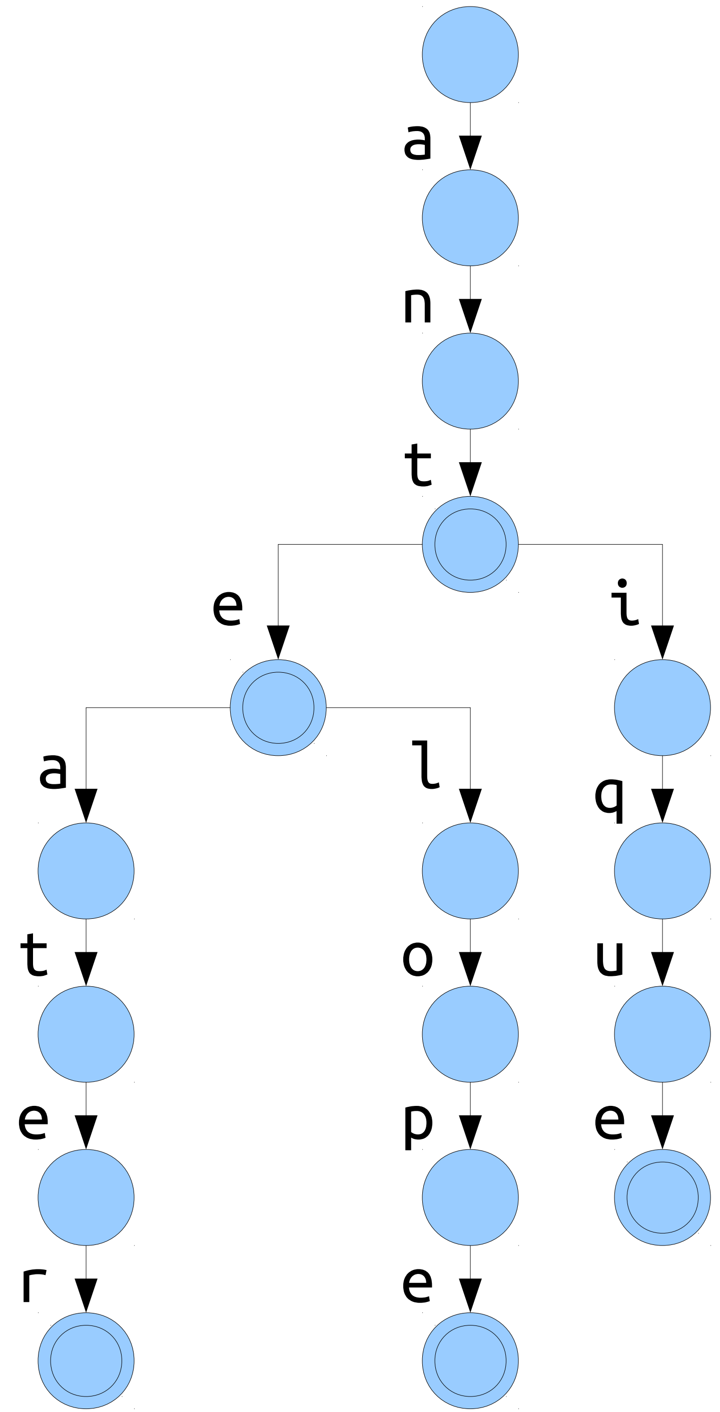
Tries

- ***Recall:*** The total length of our text strings is ***m***, and the length of our pattern string is ***n***.
- How long does it take to find all text strings that start with the pattern?
- That's a trickier question.



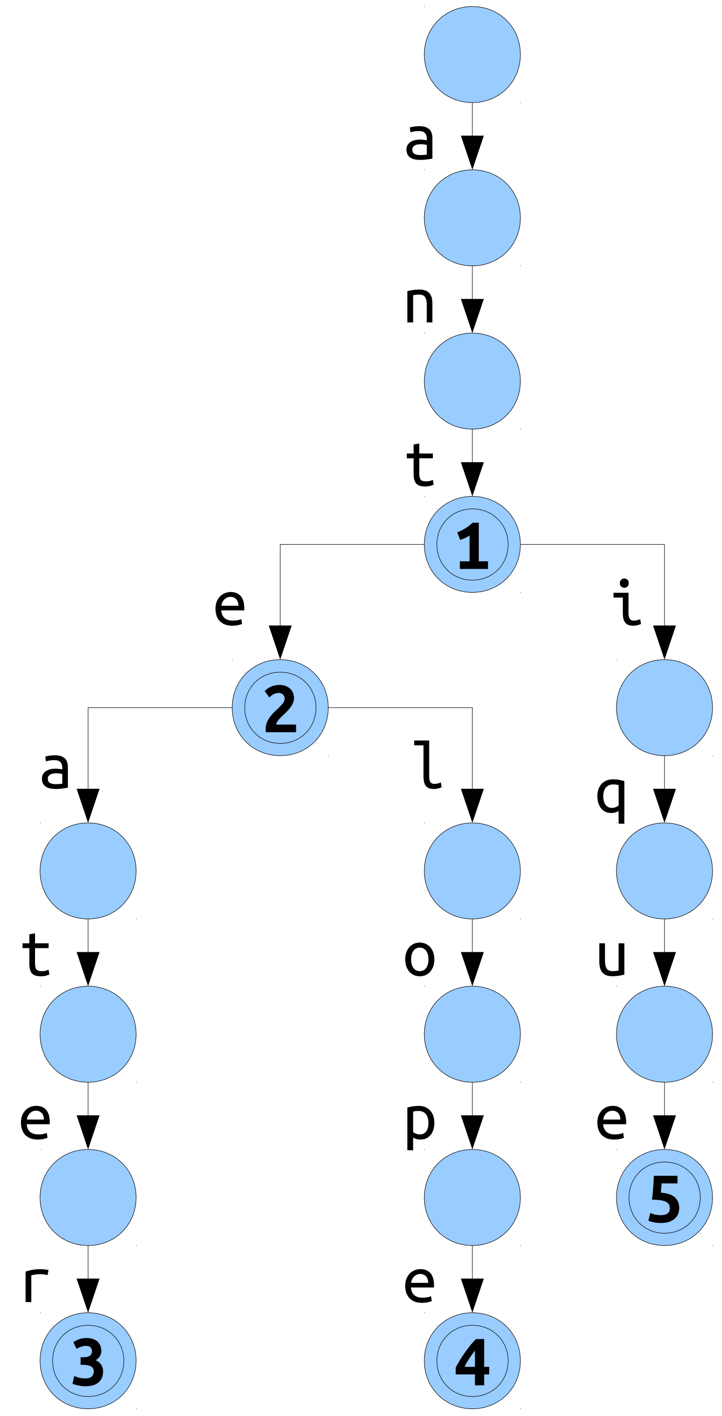
Tries

- ***Question:*** In what format do we want our matches?
- ***Option 1:*** Just print out all the matches.
- We can upper-bound the time complexity at O($\boldsymbol{m}$ + $\boldsymbol{n}$), but it's hard to say much more than that.
  - (We could upper-bound this expression at O($\boldsymbol{m}$) if we'd like, but I like showing both costs here.)

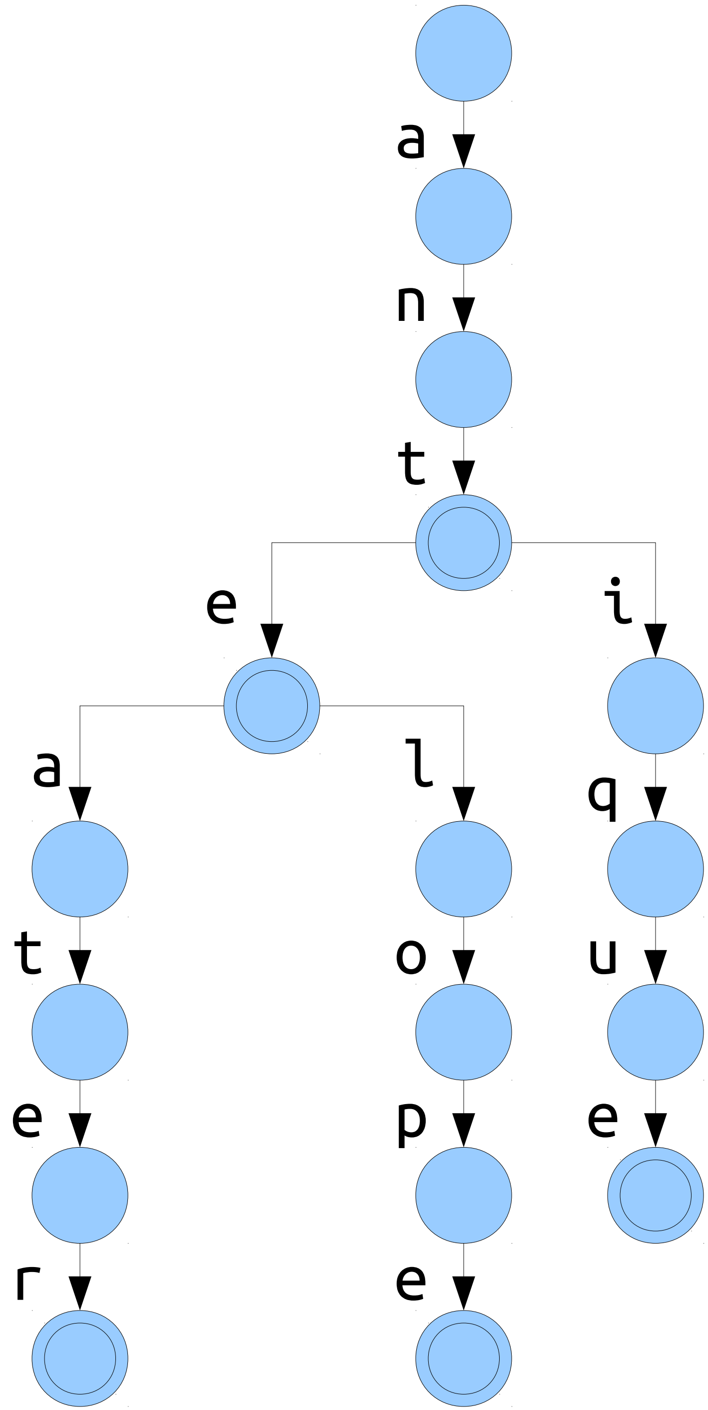


Tries

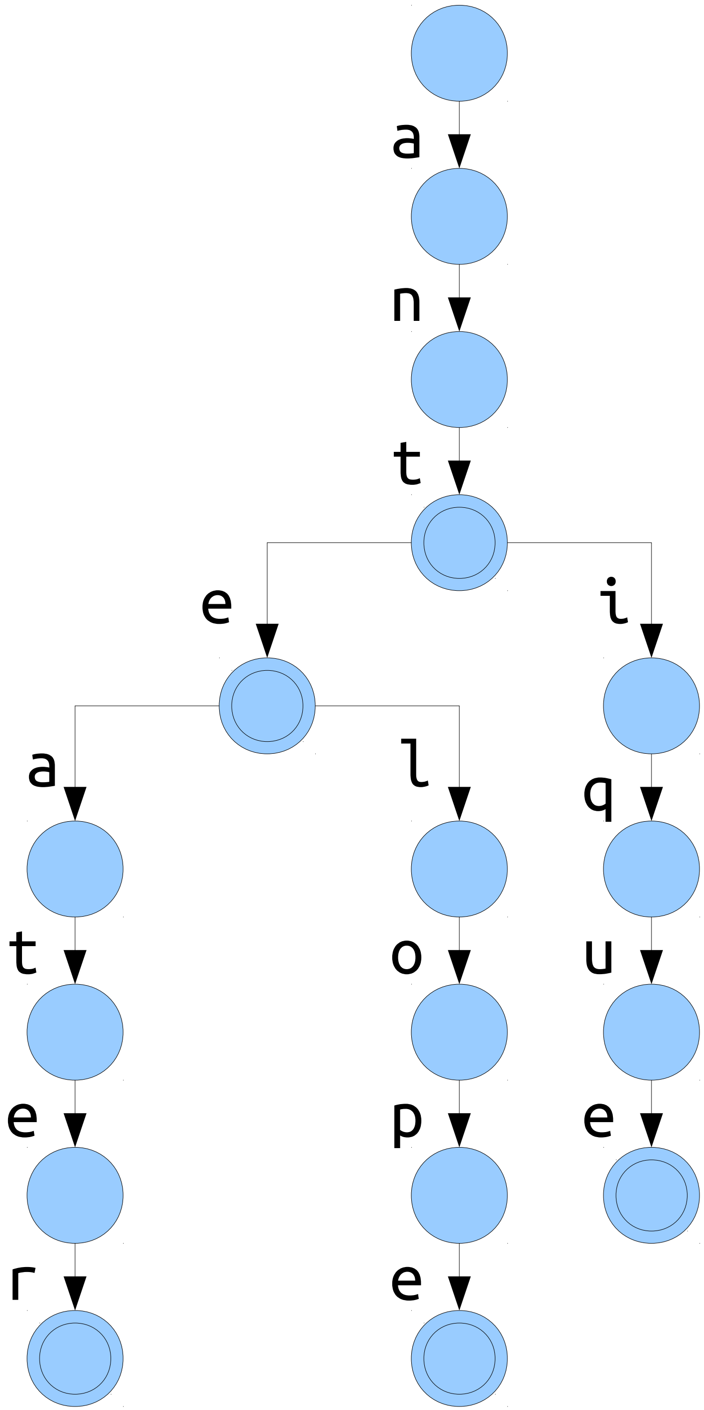
- ***Question:*** In what format do we want our matches?
- ***Option 2:*** Assume each text string has some numeric ID, and we want all matching IDs.
- Ideally, we'd like a time complexity of something like O($\boldsymbol{n}$ + $\boldsymbol{z}$), where $\boldsymbol{z}$ is the number of matches.
- Our current DFS can't achieve this; the lengths of the string matter.
- Can we do better?



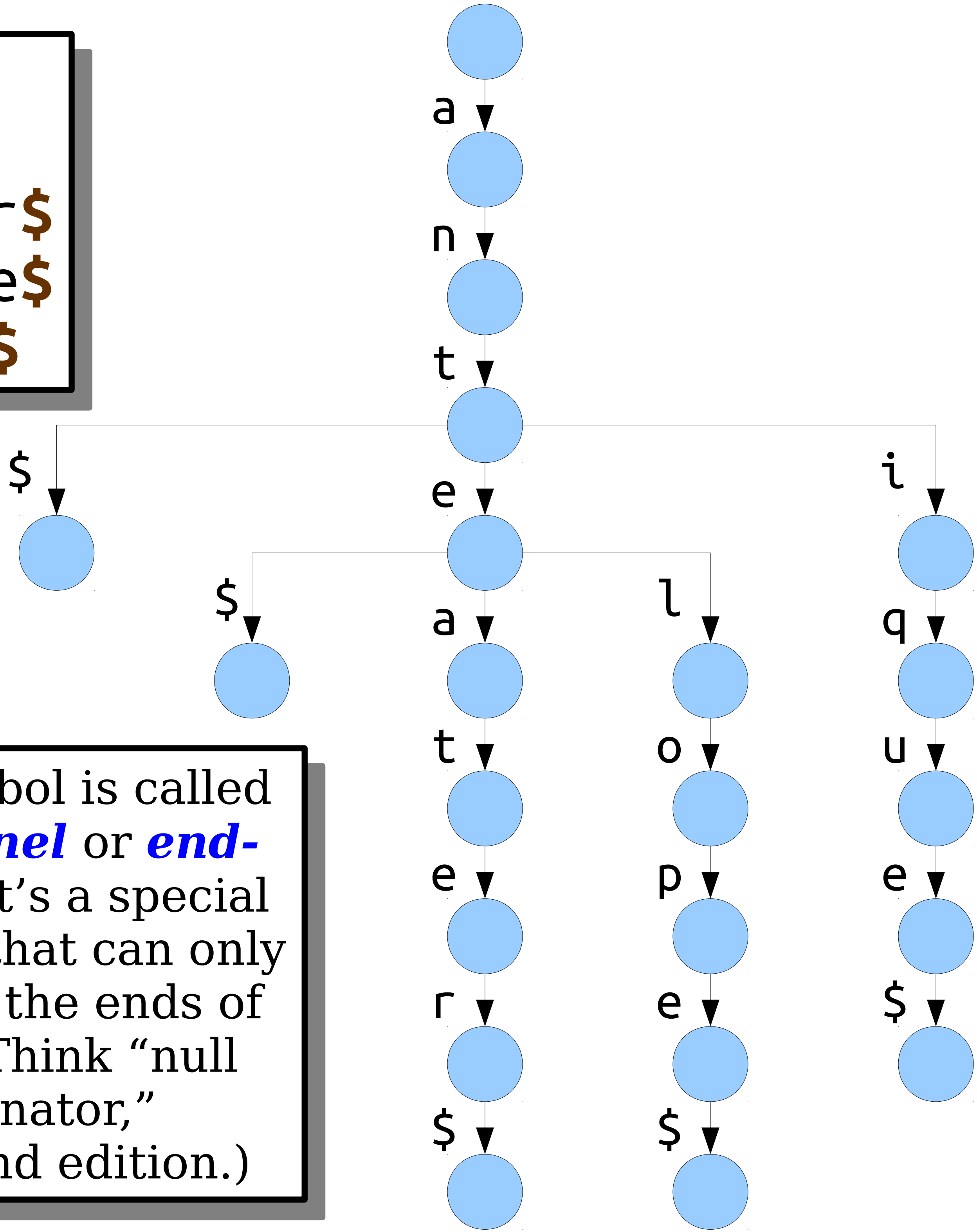
ant
ante
anteater
antelope
antique



ant$
ante$
anteater$
antelope$
antique$

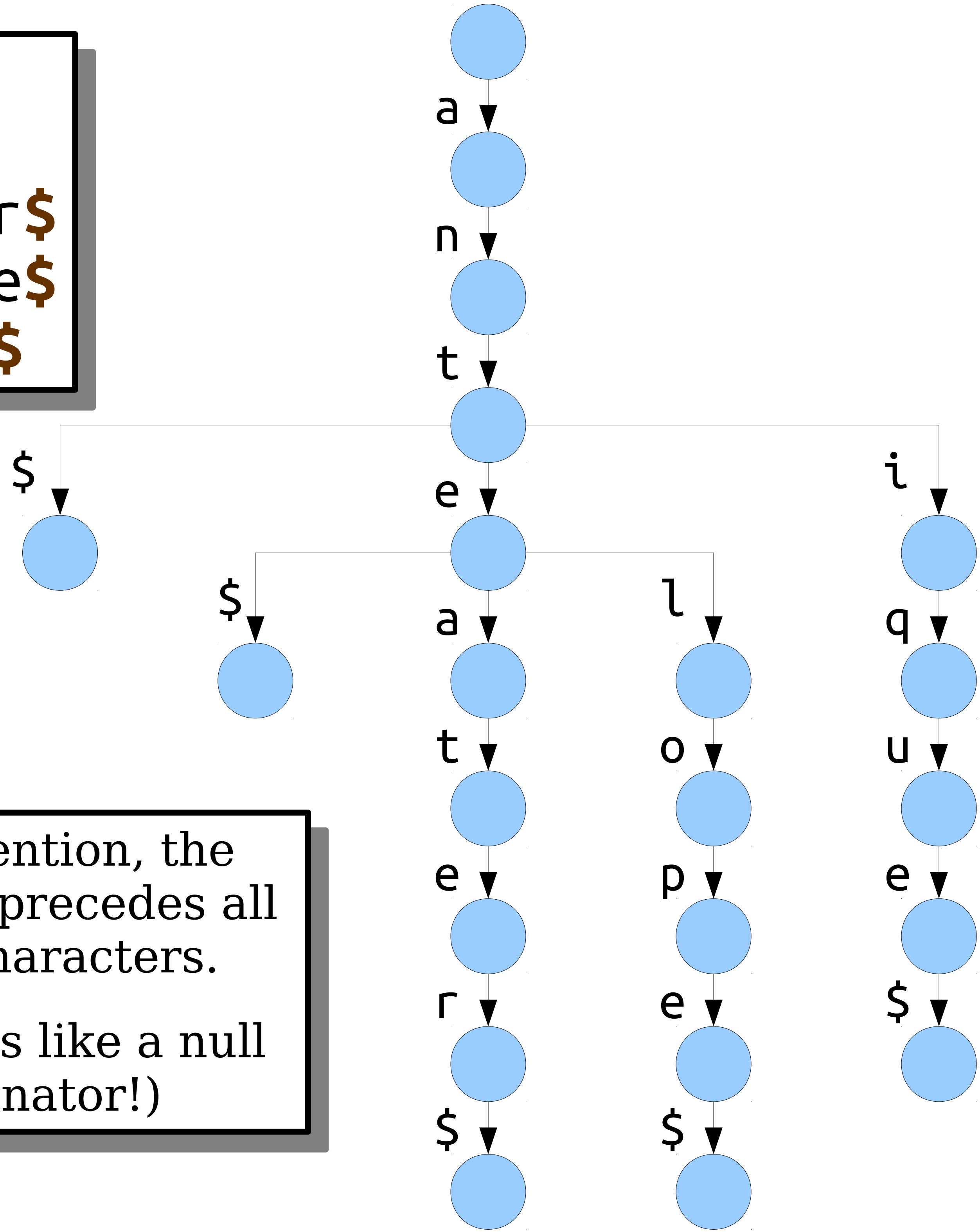


```
ant$
ante$
anteater$
antelope$
antique$
```



The **$** symbol is called the ***sentinel*** or ***end-marker***. It's a special character that can only appear at the ends of words. (Think "null terminator," Theoryland edition.)

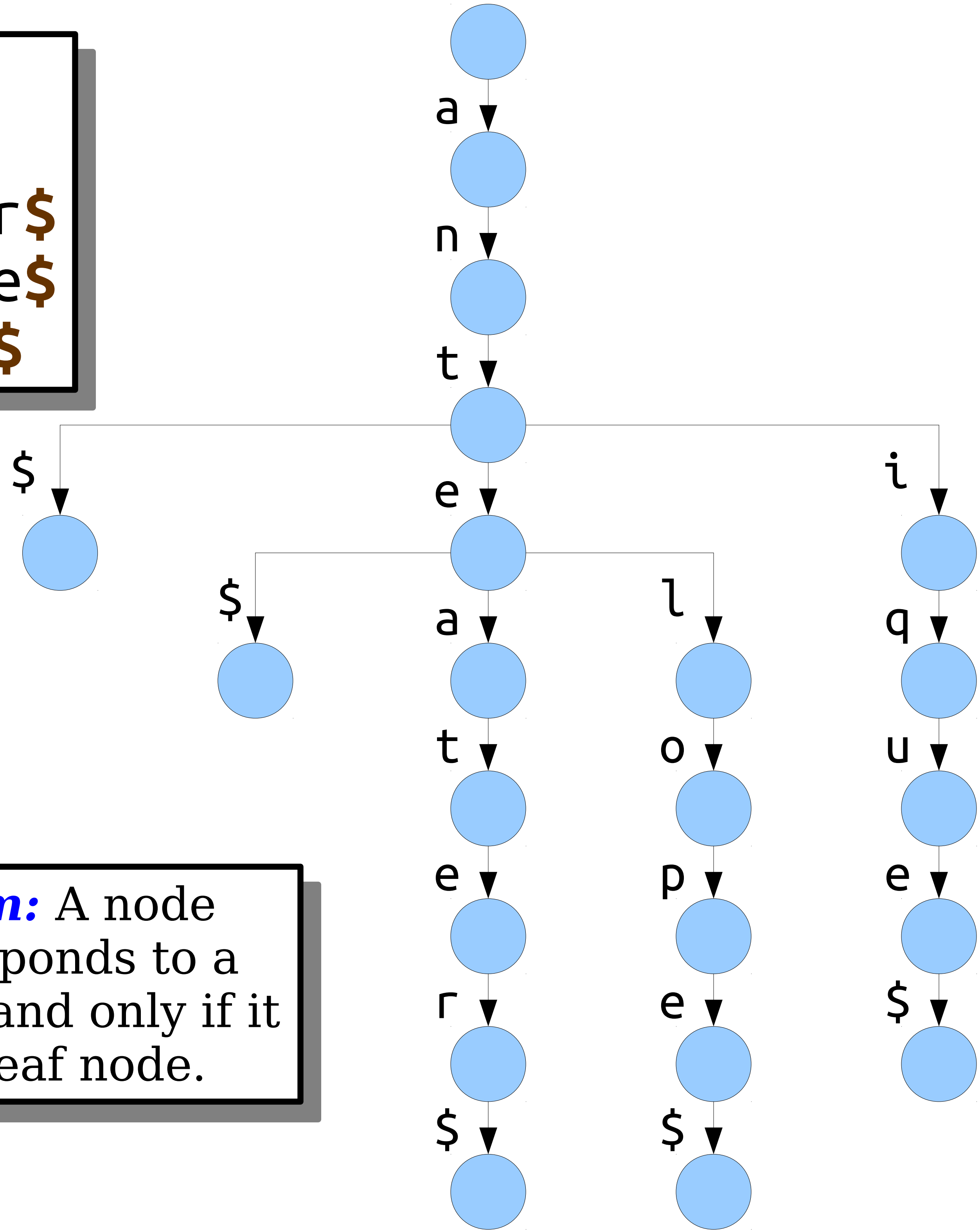
ant$
ante$
anteater$
antelope$
antique$



By convention, the sentinel $ precedes all other characters.

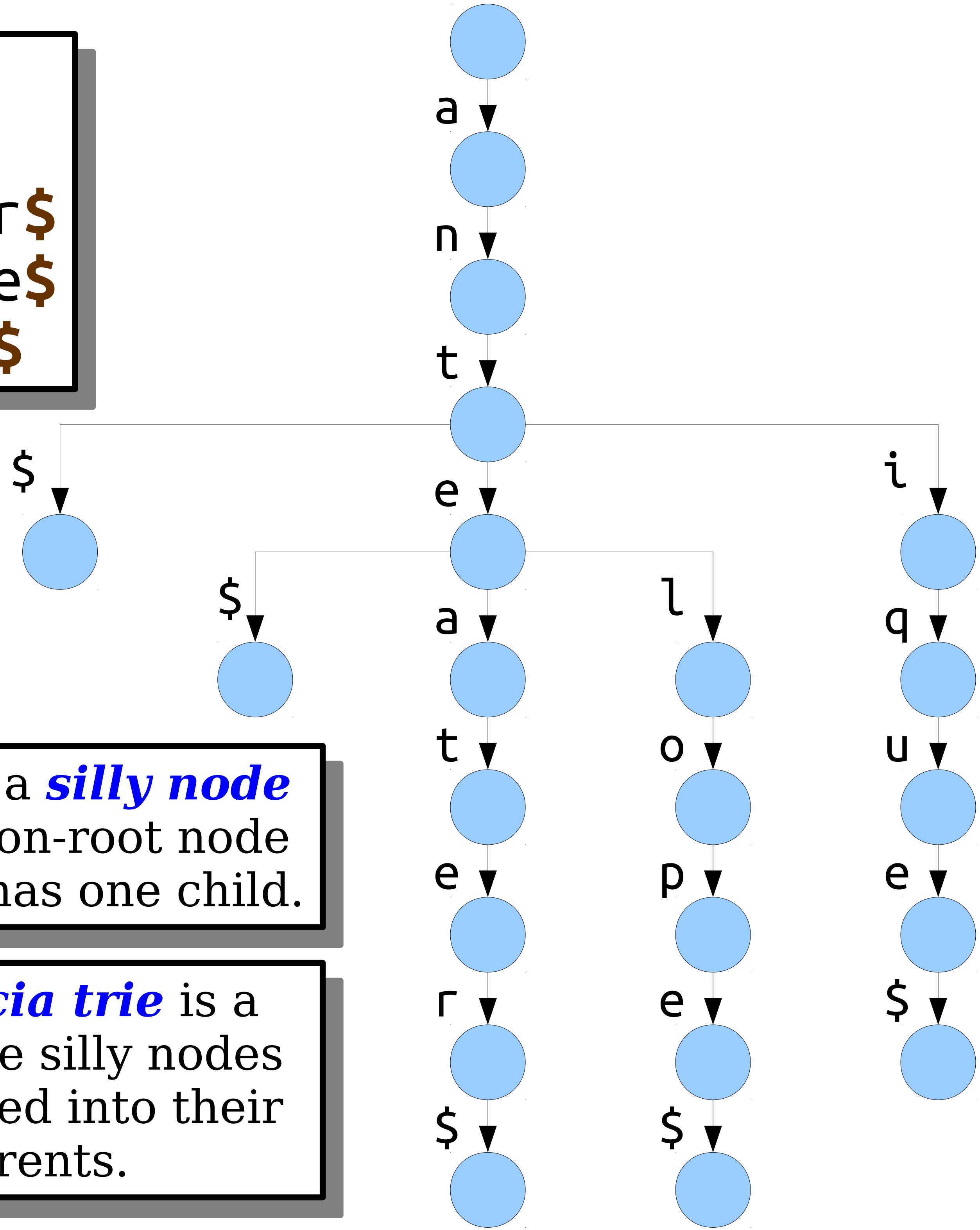
(It really is like a null terminator!)

ant$
ante$
anteater$
antelope$
antique$



Claim: A node corresponds to a word if and only if it is a leaf node.

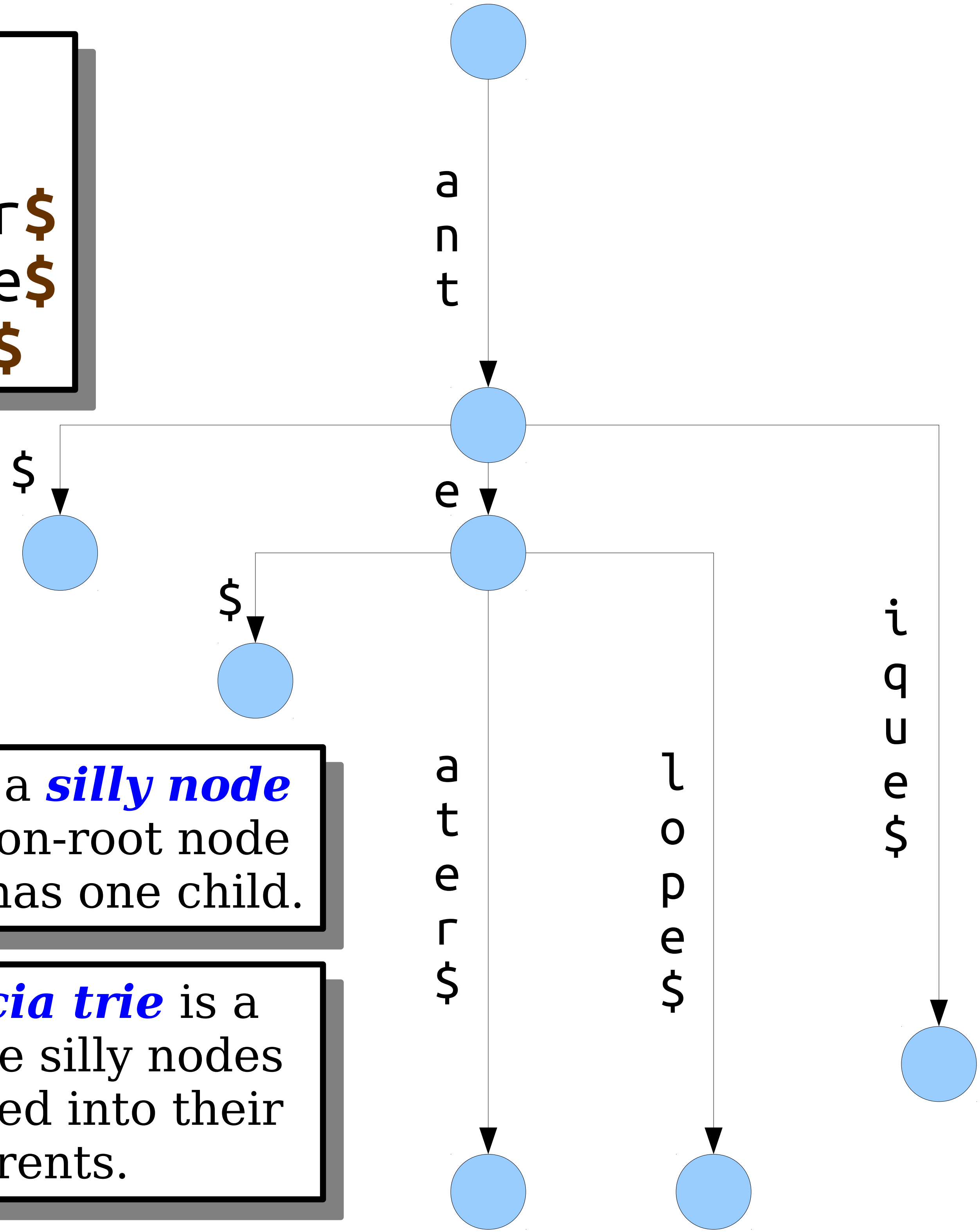
ant$
ante$
anteater$
antelope$
antique$



A node is a ***silly node*** if it is a non-root node that only has one child.

A ***Patricia trie*** is a trie where silly nodes are merged into their parents.

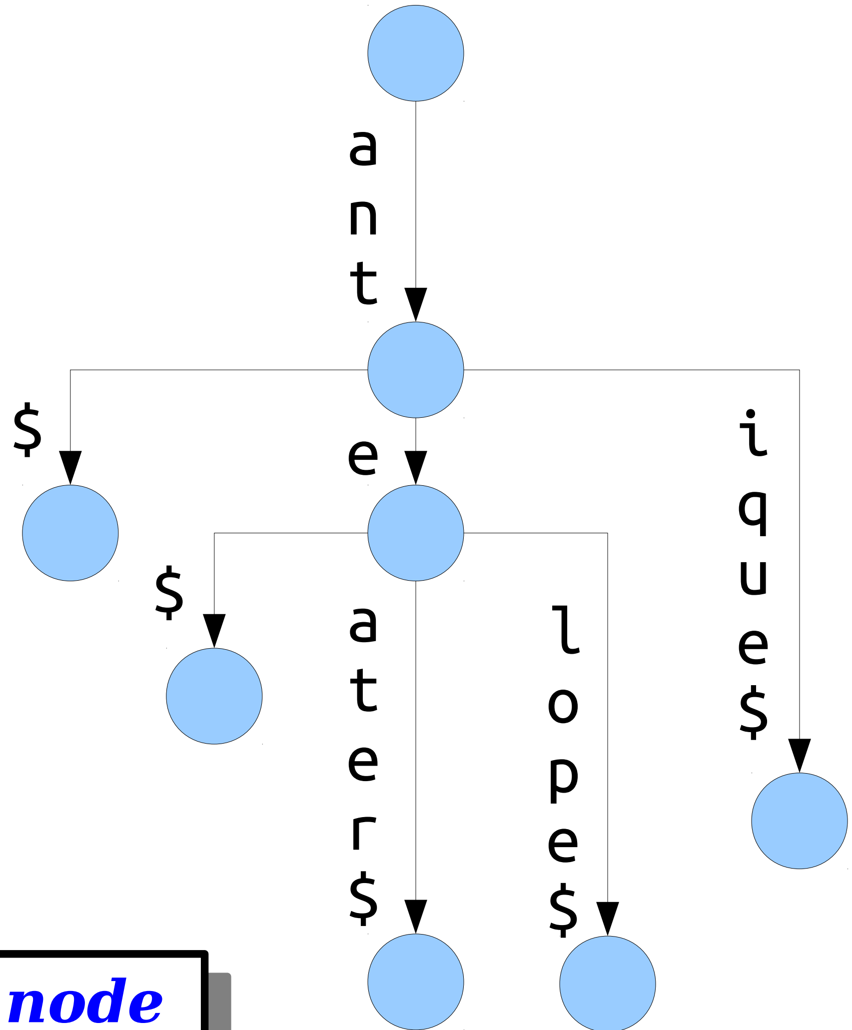
ant$
ante$
anteater$
antelope$
antique$



A node is a ***silly node*** if it is a non-root node that only has one child.

A ***Patricia trie*** is a trie where silly nodes are merged into their parents.

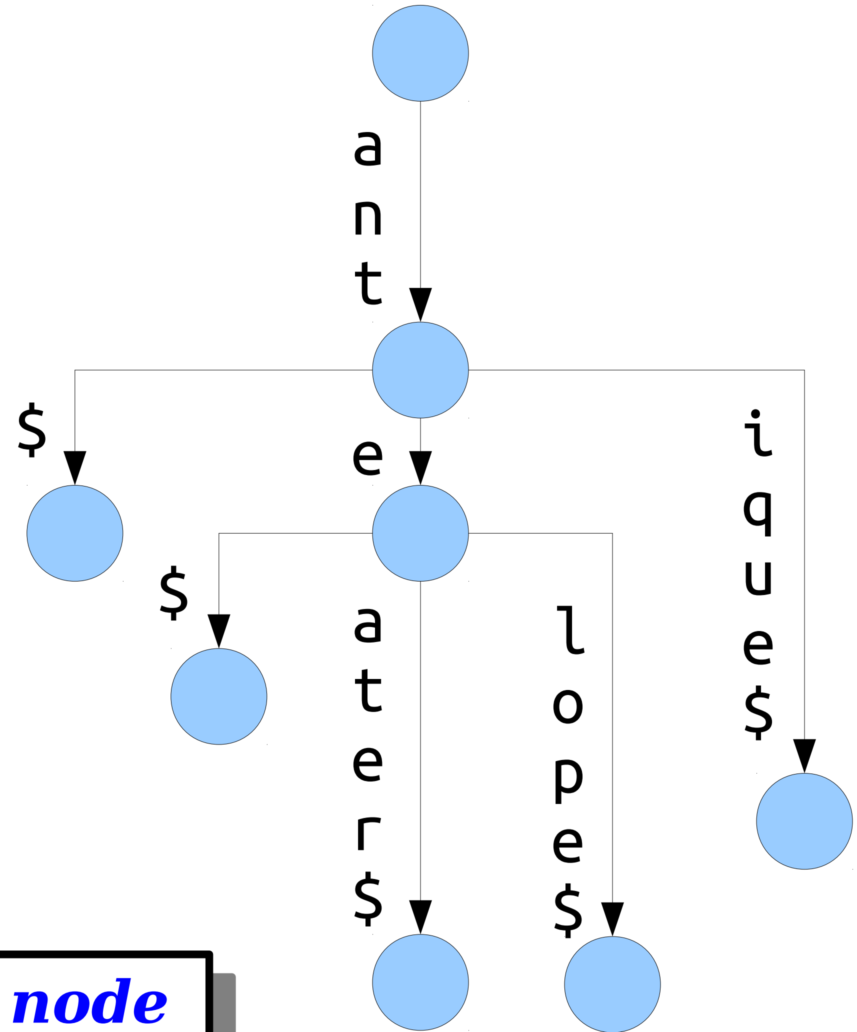
ant$
ante$
anteater$
antelope$
antique$



A node is a ***silly node*** if it is a non-root node that only has one child.

A ***Patricia trie*** is a trie where silly nodes are merged into their parents.

ant$
ante$
anteater$
antelope$
antique$



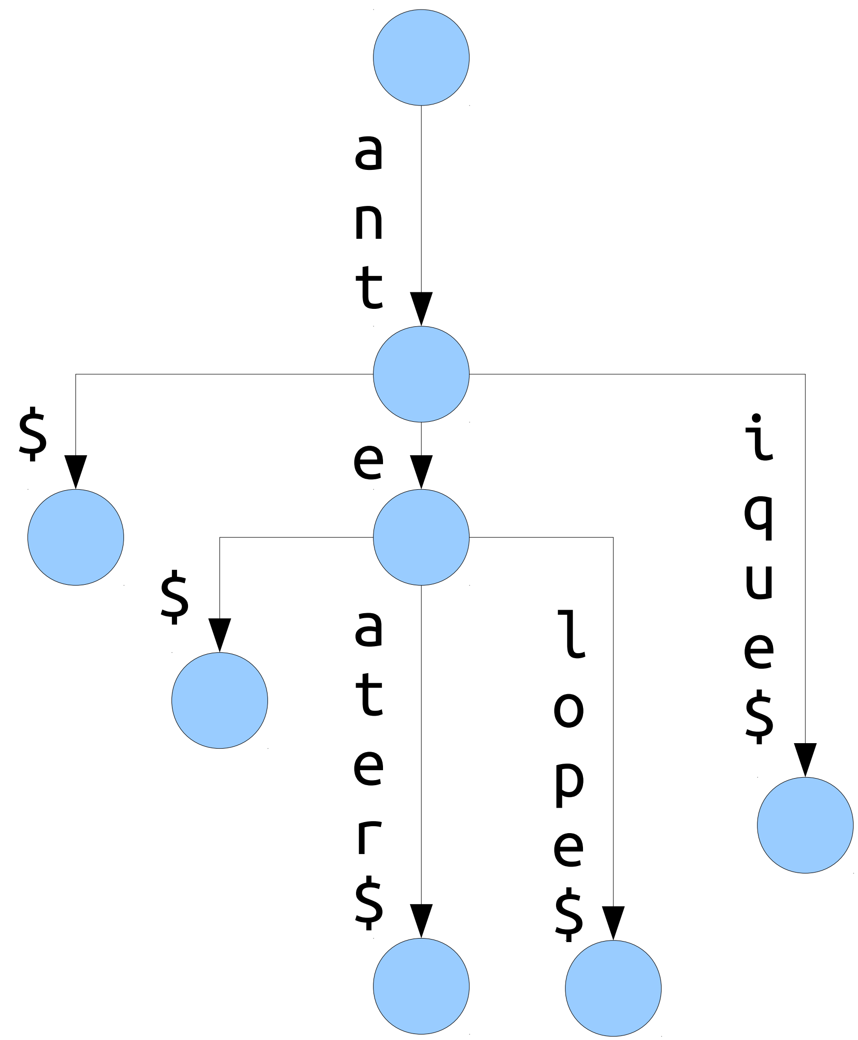
A node is a ***silly node*** if it is a non-root node that only has one child.

A ***Patricia trie*** is a trie where silly nodes are merged into their parents.

Observation: Every internal node in a Patricia trie (except possibly the root) has two or more children.

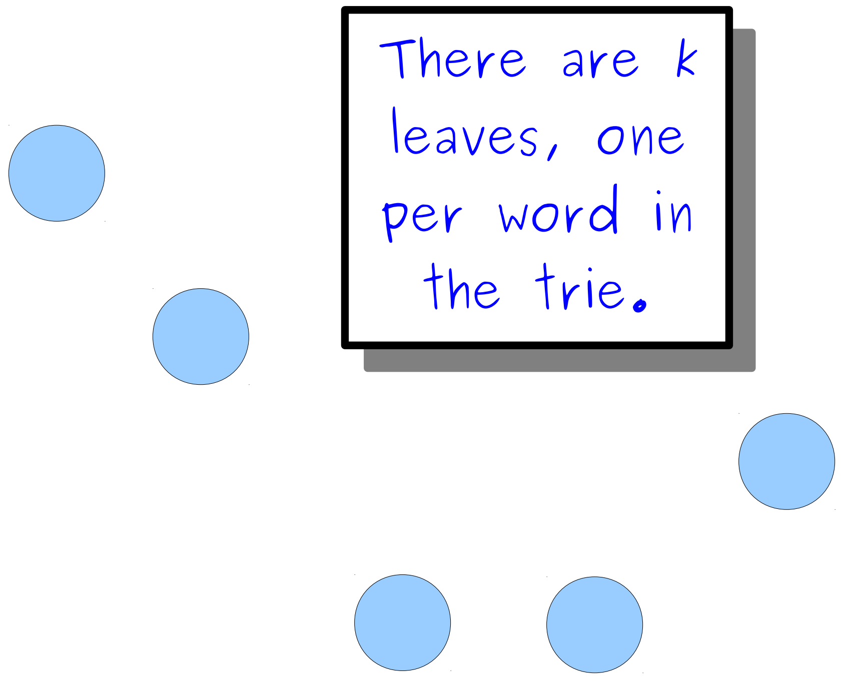
Patricia Tries

- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.



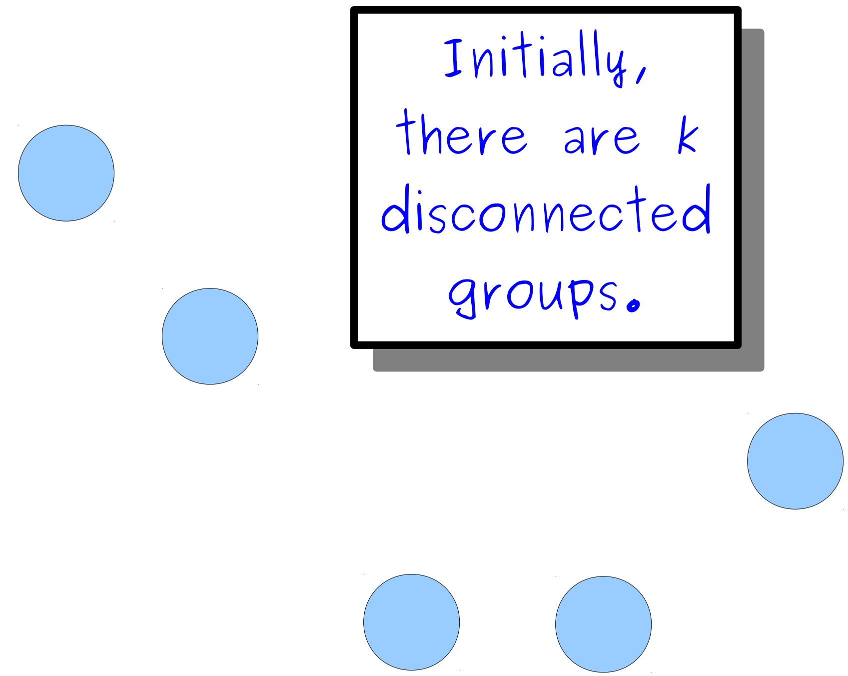
Patricia Tries

- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.



Patricia Tries

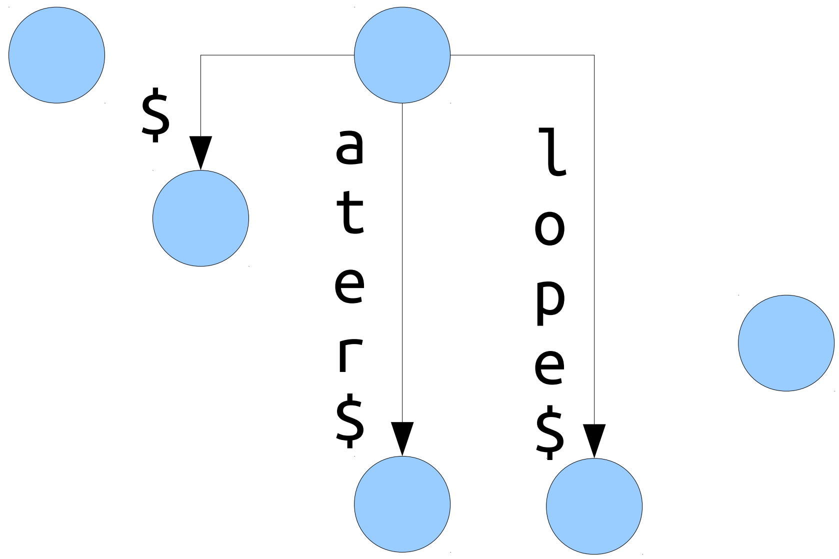
- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.



Patricia Tries

- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.

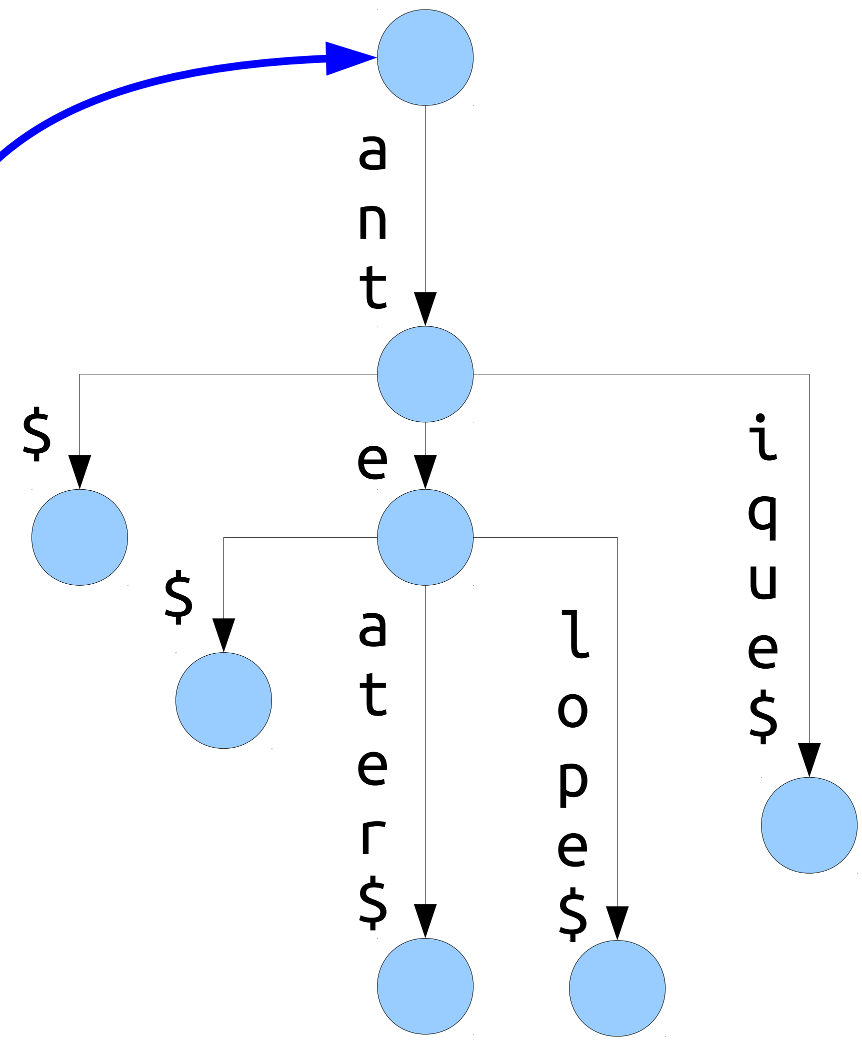
Adding an internal node decreases the number of clusters by at least one.



Patricia Tries

- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.

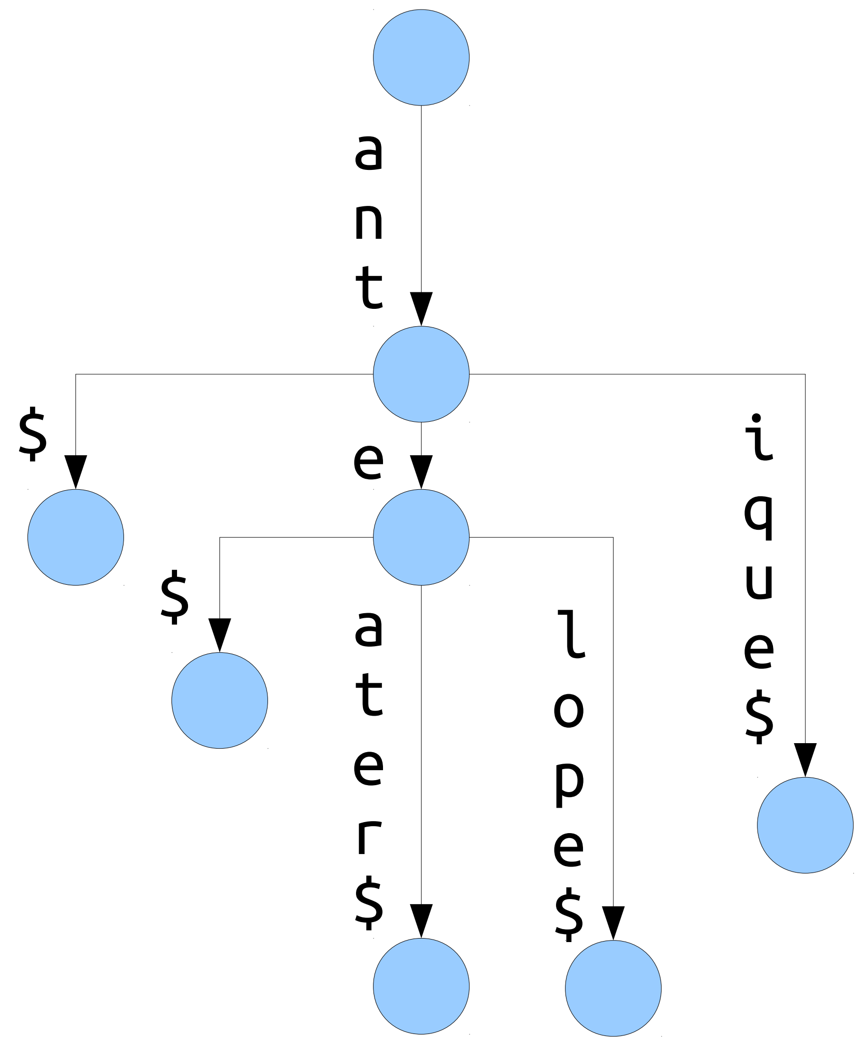
The root might not decrease the number of clusters, but there's only one root.



Patricia Tries

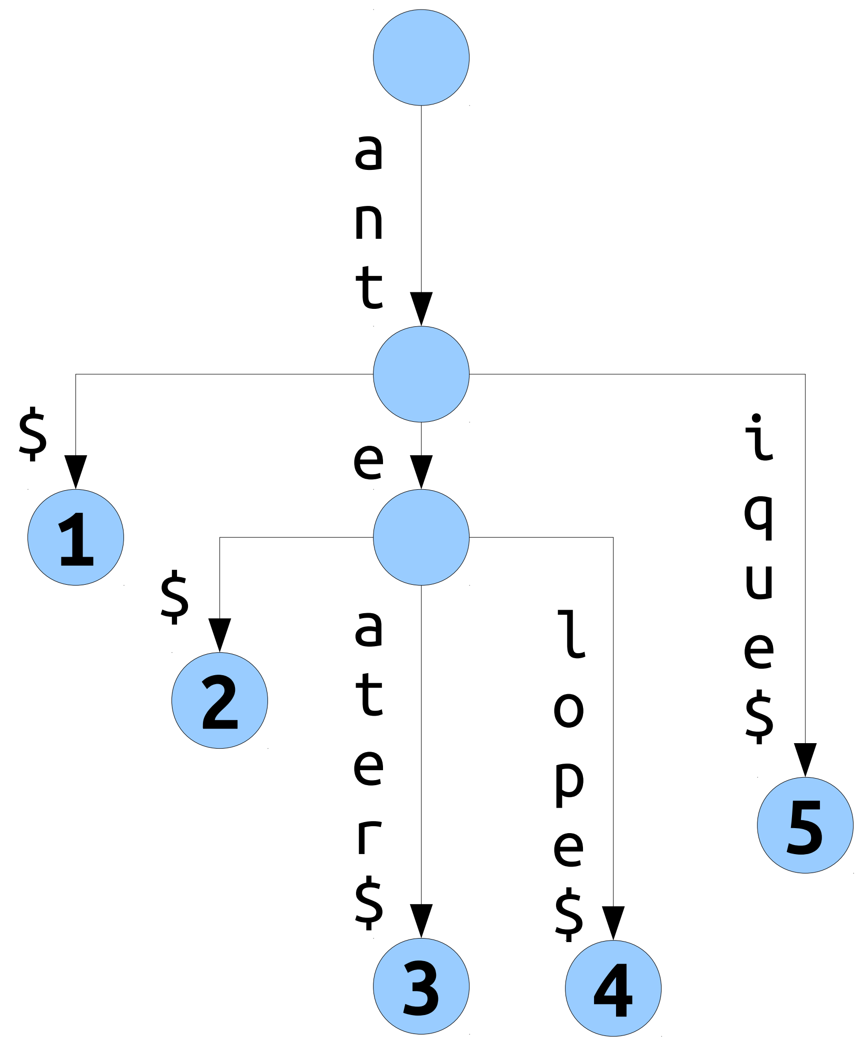
- ***Theorem:*** The number of nodes in a Patricia trie with k words is always $O(k)$, regardless of what those words are.
- ***Proof Sketch:*** There are k leaf nodes, one per word. Remove all the internal nodes, leaving k clusters.

  Now add the internal nodes back one at a time. Each addition decreases the number of clusters by at least one, since each internal node has at least two children. This means there are at most k internal nodes, plus the root, for a total of $O(k)$ nodes. ■



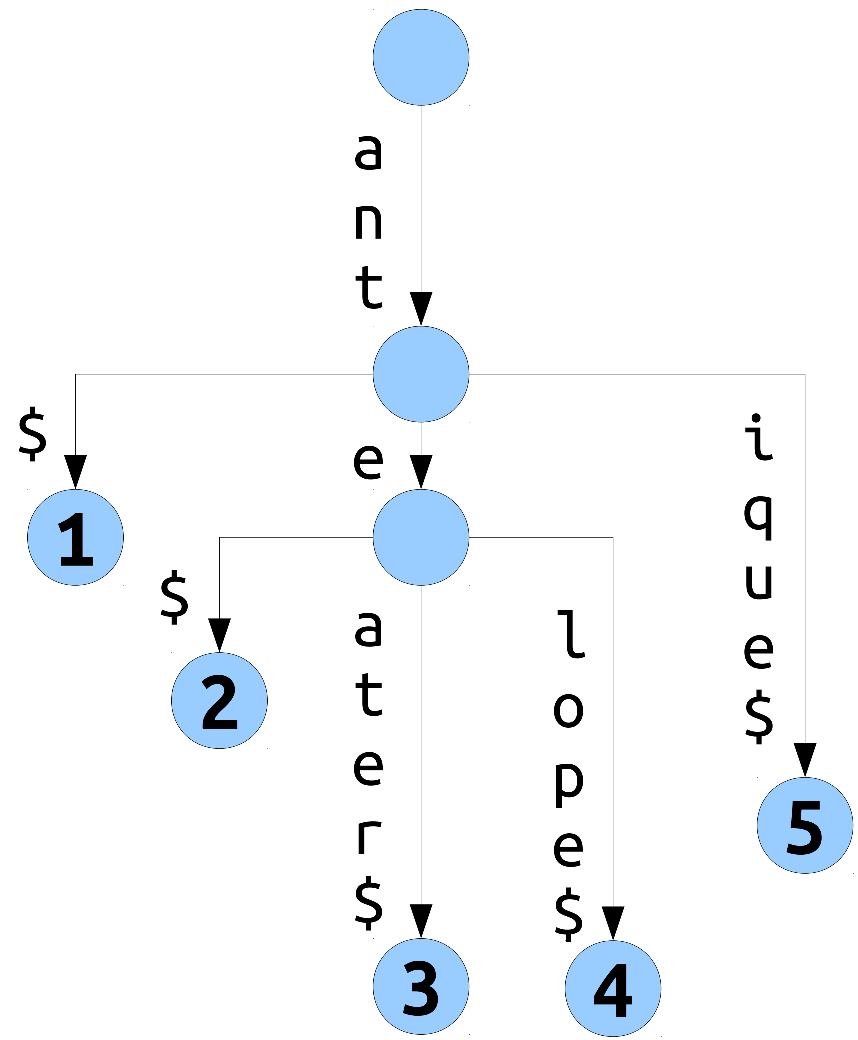
Patricia Tries

- ***Claim:*** If each leaf in a Patricia trie is annotated with the index of the word it comes from, all strings starting with a given prefix can be found in time O($\boldsymbol{n}$ + $\boldsymbol{z}$), where $\boldsymbol{n}$ is the length of that prefix and $\boldsymbol{z}$ is the number of matches.
- ***Question:*** How is this possible?



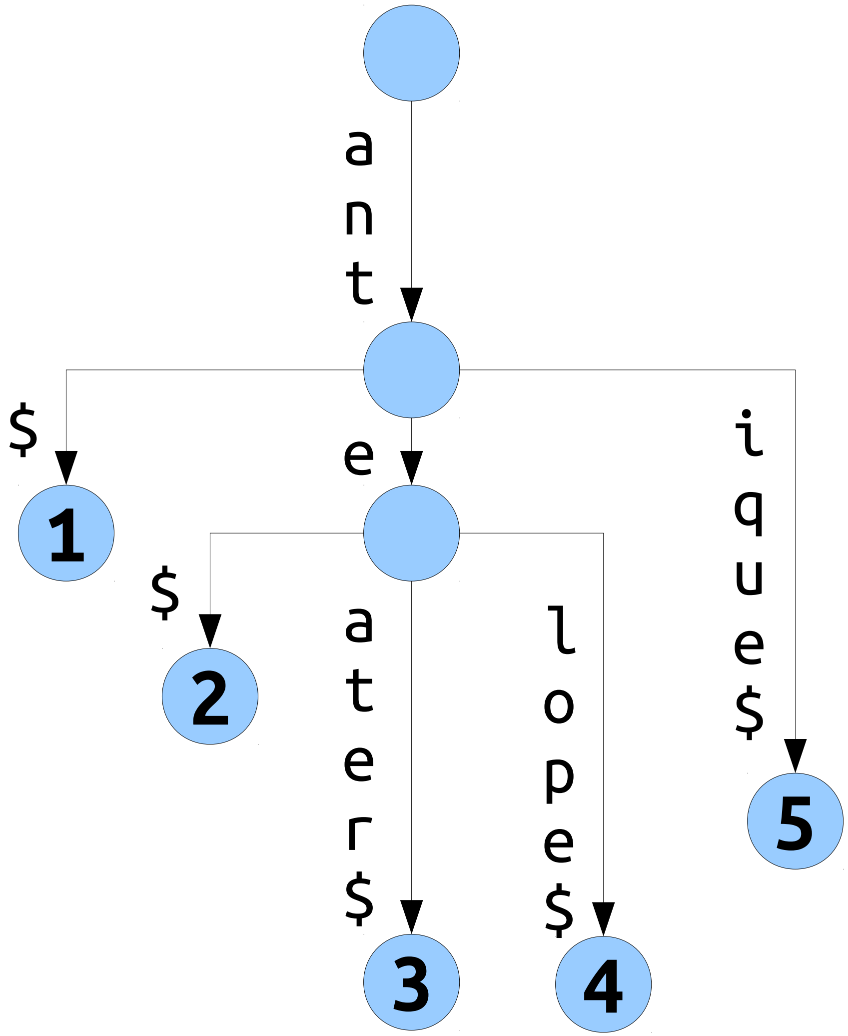
Patricia Tries

- ***Algorithm:*** Search for the prefix in time O(***n***). If you fell off the trie, stop.
- Otherwise, ensure that you're at a node by skipping to the end of the current edge if you're in the middle of an edge. Then run a DFS from the node you're visiting and output all leaves you find.
- If the DFS visits ***z*** leaf nodes, by our earlier argument, the subtrie containing those leaves has O(***z***) nodes. We don't need to read the letters on the edges; we just care about the labels on the leaves.
- Total runtime: O(***n*** + ***z***).



The Story So Far

- Adopting our notation from RMQ, a Patricia trie gives an ⟨O($\boldsymbol{m}$), O($\boldsymbol{n}$ + $\boldsymbol{z}$)⟩ solution to prefix matching.
- Those runtimes hide the effect of the alphabet size; take some time to evaluate those tradeoffs!



Time-Out for Announcements!

Stanford Women in Computer Science

WiCS Board Applications 2019-2020 are LIVE

Passionate about empowering a community of women in CS?

Want to shape the future of Stanford's computer science community?

How about meeting some of the most talented CS students at Stanford?

Apply for WiCS Board today!

Apply **HERE**.

Read about our teams **HERE**.

The Deadline is Sunday, April 14th at 11:59 PM

Contact Kasey (kaseyluo) or Pau (panzaldo) with any questions!

Problem Set One

- Problem Set Zero was due today at 2:30PM.
  - Did you remember to submit both the written and coding components?
- Problem Set One goes out today. It's due next Tuesday at 2:30PM.
  - Play around with range minimum queries and the concepts and techniques associated with them!
  - Have questions? Ask on Piazza or stop by office hours!

Back to CS166!

Two Motivating Problems

The *United States Statutes at Large* contains all legislation ever passed in the United States.

Make it searchable.

Cancers often have repeated copies the same gene.

Given a cancer genome (length ~3,000,000,000),
find the longest repeated DNA sequence.

Patricia tries are great tools for finding ***prefixes***.

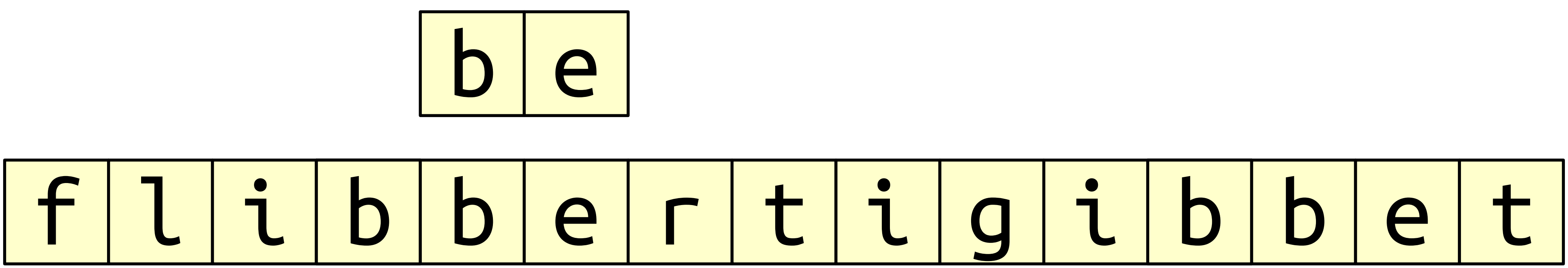
These problems involve looking for ***substrings***.

Can we use what we've developed so far?

A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

w is a substring of x
if and only if
w is a prefix of a suffix of x



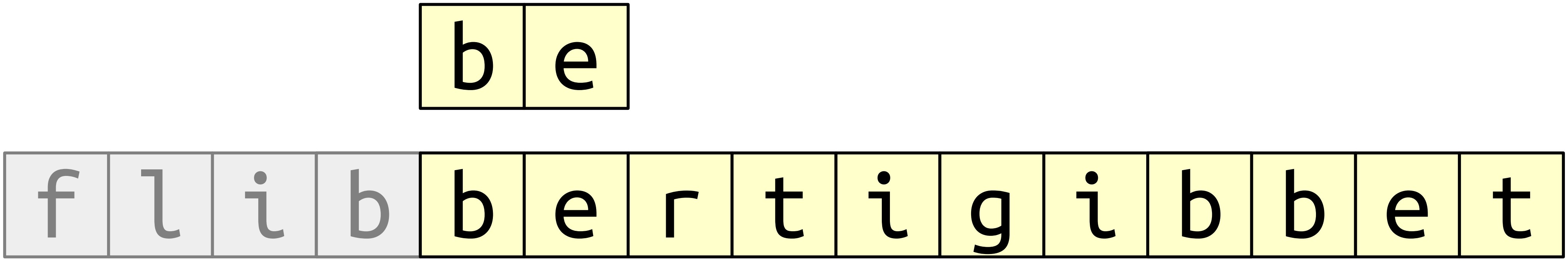
A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

_w_ is a substring of _x_

if and only if

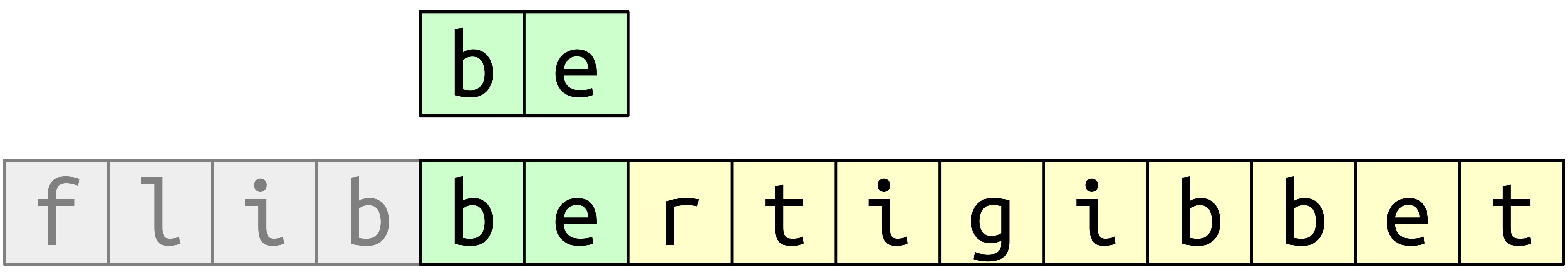
_w_ is a prefix of a suffix of _x_



A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

_w_ is a substring of _x_
if and only if
_w_ is a prefix of a suffix of _x_



A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

_w_ is a substring of _x_
if and only if
_w_ is a prefix of a suffix of _x_

												b	e	
f	l	i	b	b	e	r	t	i	g	i	b	b	e	t

A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

_w_ is a substring of _x_
if and only if
_w_ is a prefix of a suffix of _x_

A Fundamental Theorem

- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

_w_ is a substring of _x_

if and only if

_w_ is a prefix of a suffix of _x_

A Fundamental Theorem

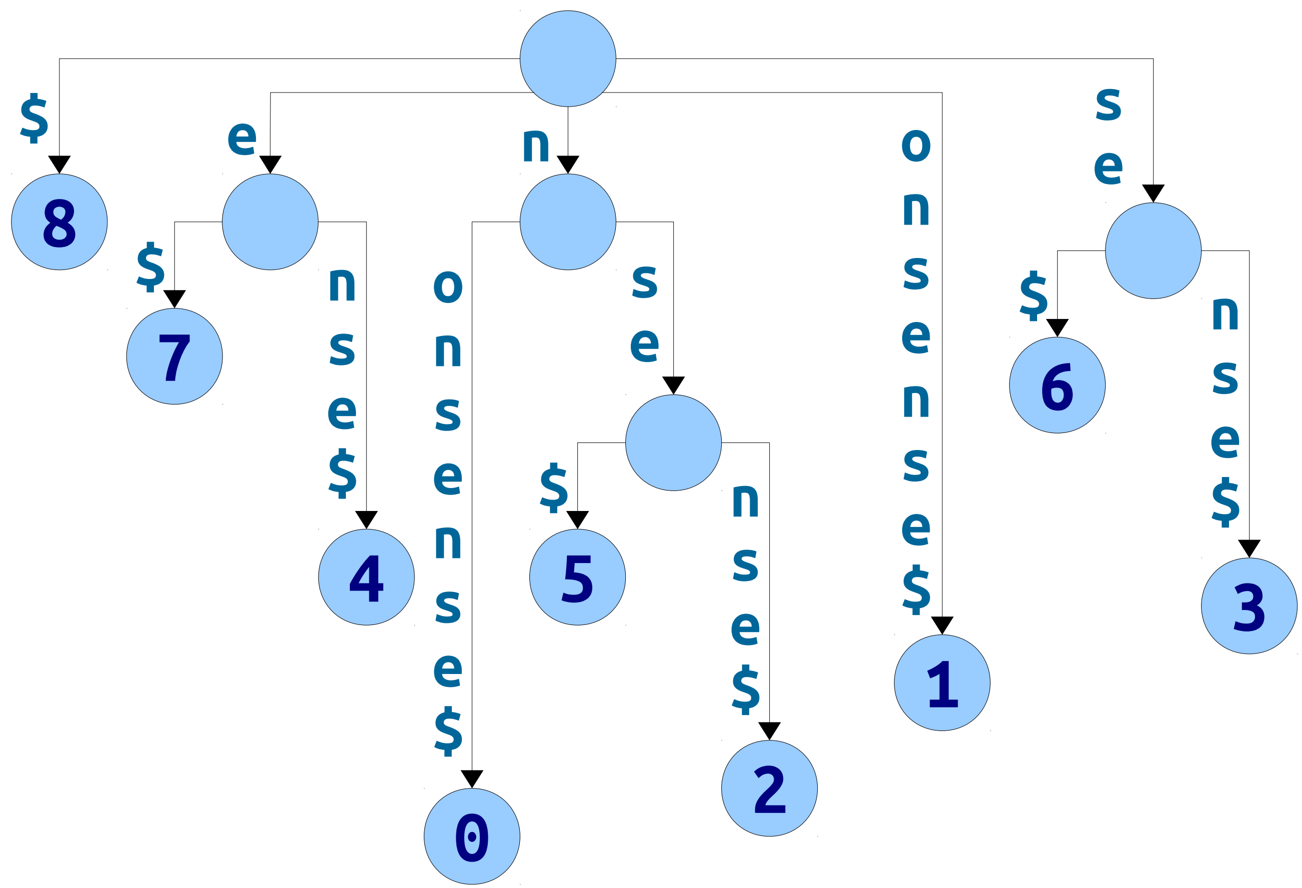
- The ***fundamental theorem of stringology*** says that, given two strings *w* and *x*, that

  w* is a substring of *x
  if and only if
  w* is a prefix of a suffix of *x

- ***Recall:*** Patricia tries make it really easy to check if something is a prefix of any number of strings.
- ***Idea:*** Store all the suffixes of a string in a Patricia trie!

Suffix Trees

- A ***suffix tree*** for a string *T* is an Patricia trie of all suffixes of *T*$.
- Each leaf is labeled with the starting index of that suffix.
- ***Coming Up:***
  - How does this help us solve our earlier problems?
  - How much space does this take up?
  - How quickly can we build this?



nonsense$
012345678

Suffix Trees

A ***suffix tree*** for a string *T* is an Patricia trie of all suffixes of *T*$.

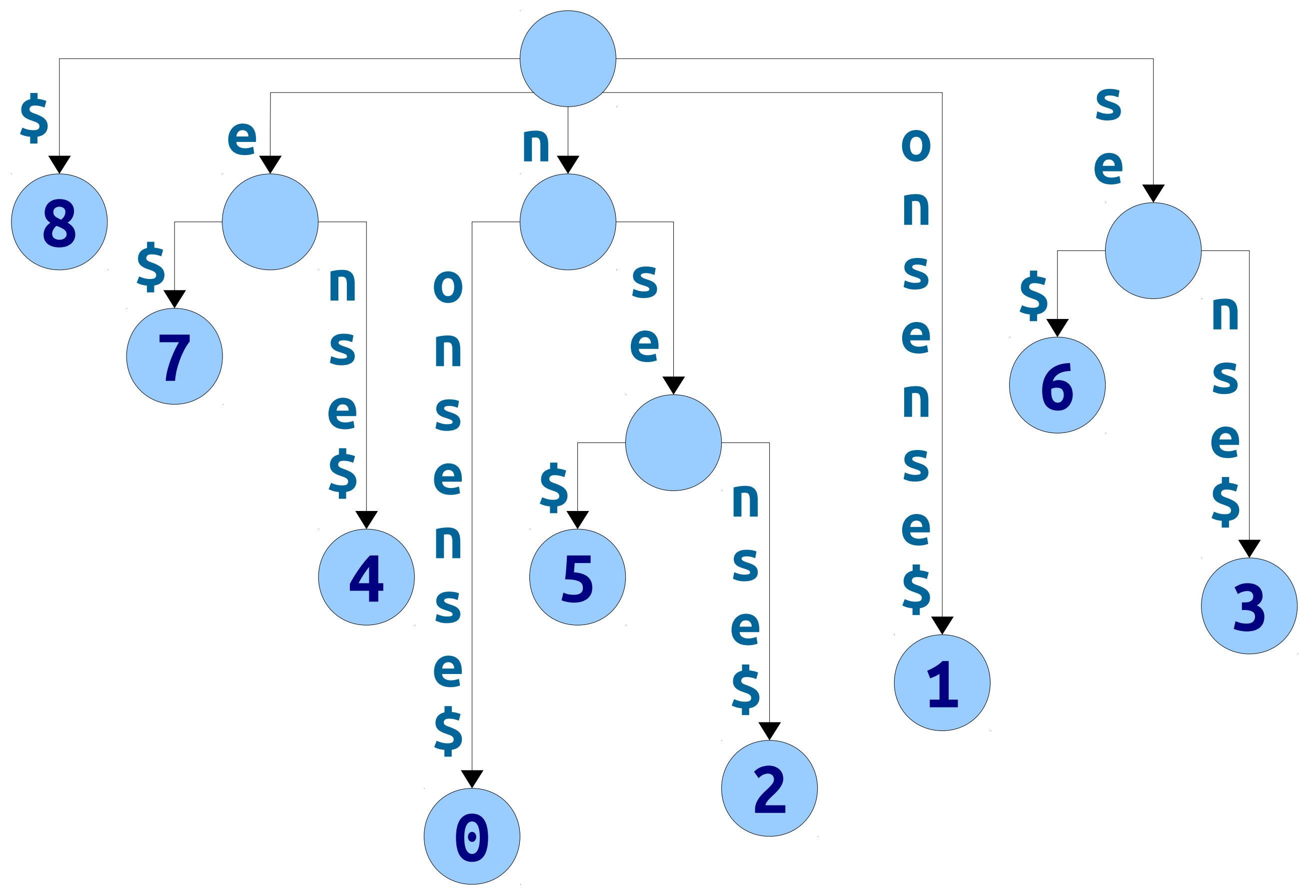
Each leaf is labeled with the starting index of that suffix.

Coming Up:

- How does this help us solve our earlier problems?

  How much space does this take up?

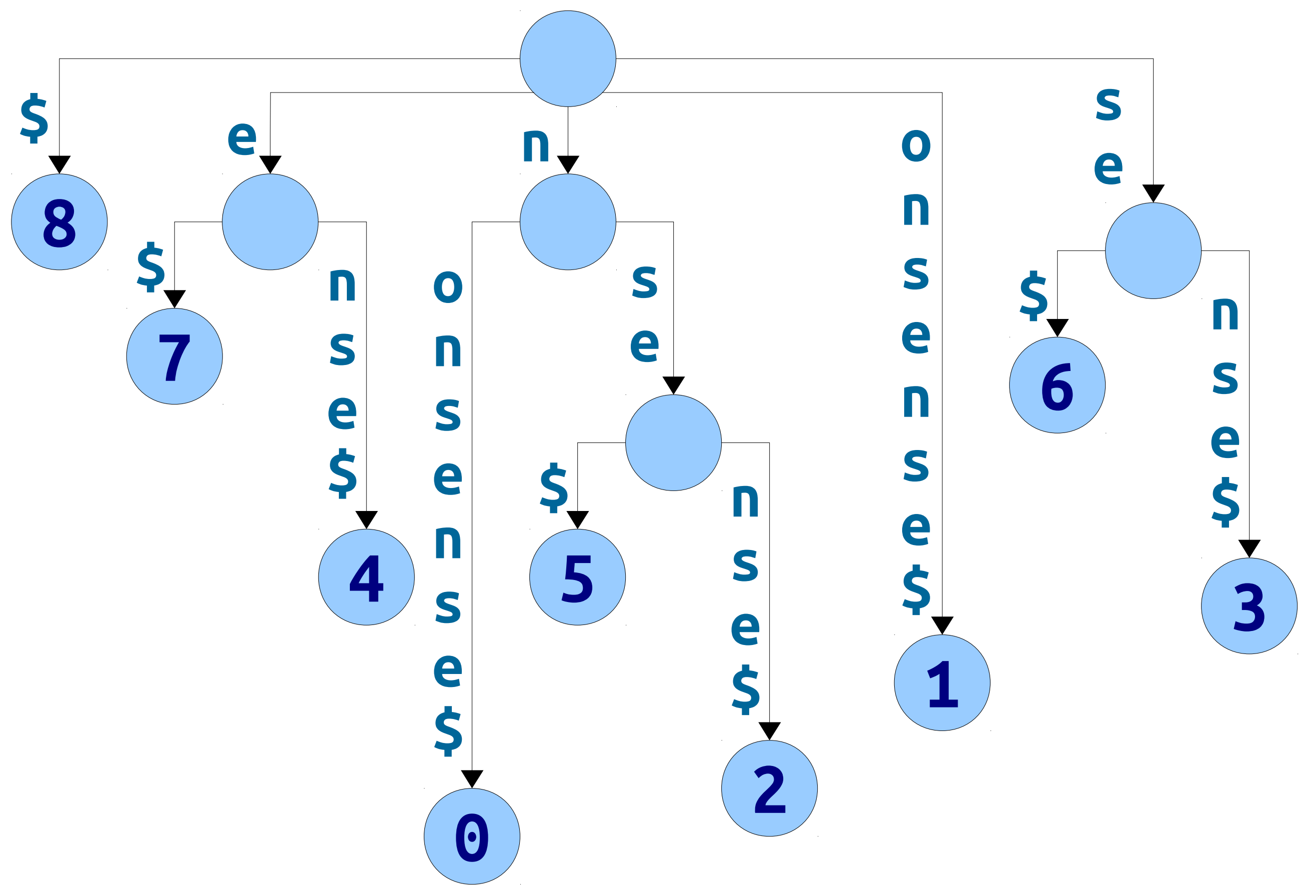
  How quickly can we build this?



nonsense$
012345678

Substring Search

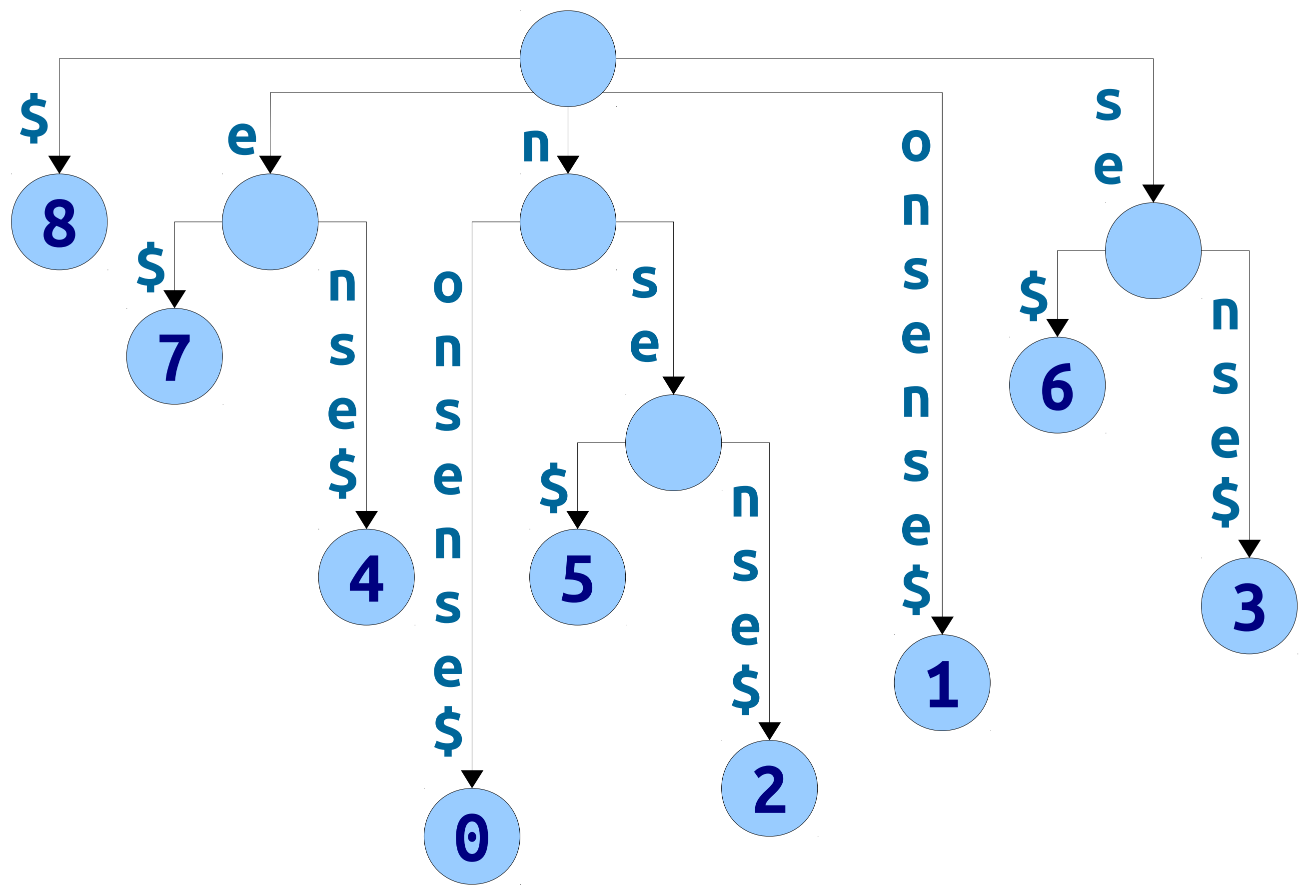
- ***Claim:*** Once we have a suffix tree for a string T, we can find all matches of a pattern P of length $\boldsymbol{n}$ in time $O(\boldsymbol{n} + \boldsymbol{z})$, where $\boldsymbol{z}$ is the number of matches.
- ***Question:*** How is this possible?



nonsense$
012345678

Substring Search

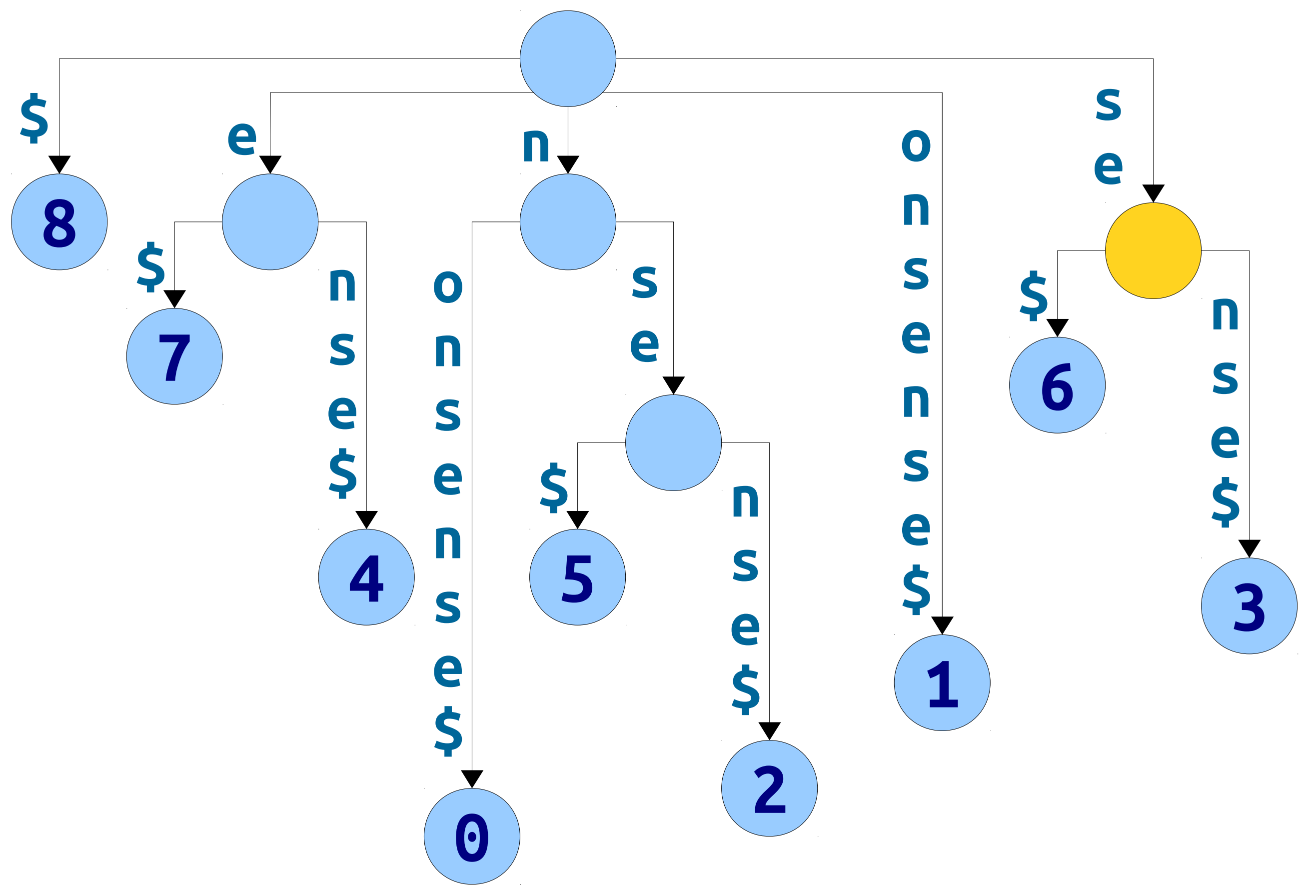
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O($\boldsymbol{n}$).
- Finding all matches takes time O($\boldsymbol{z}$).



nonsense$
012345678

Substring Search

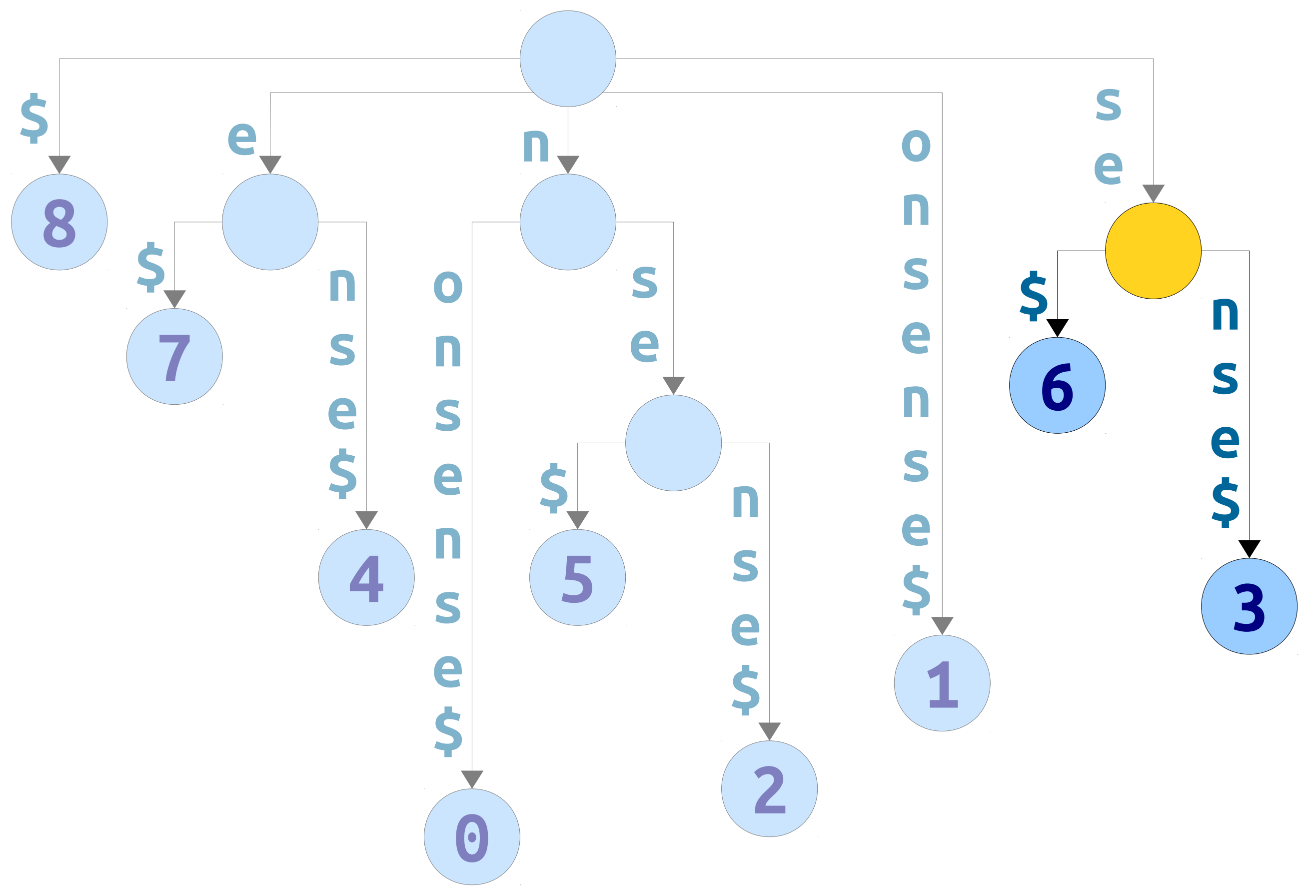
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



nonsense$
012345678

Substring Search

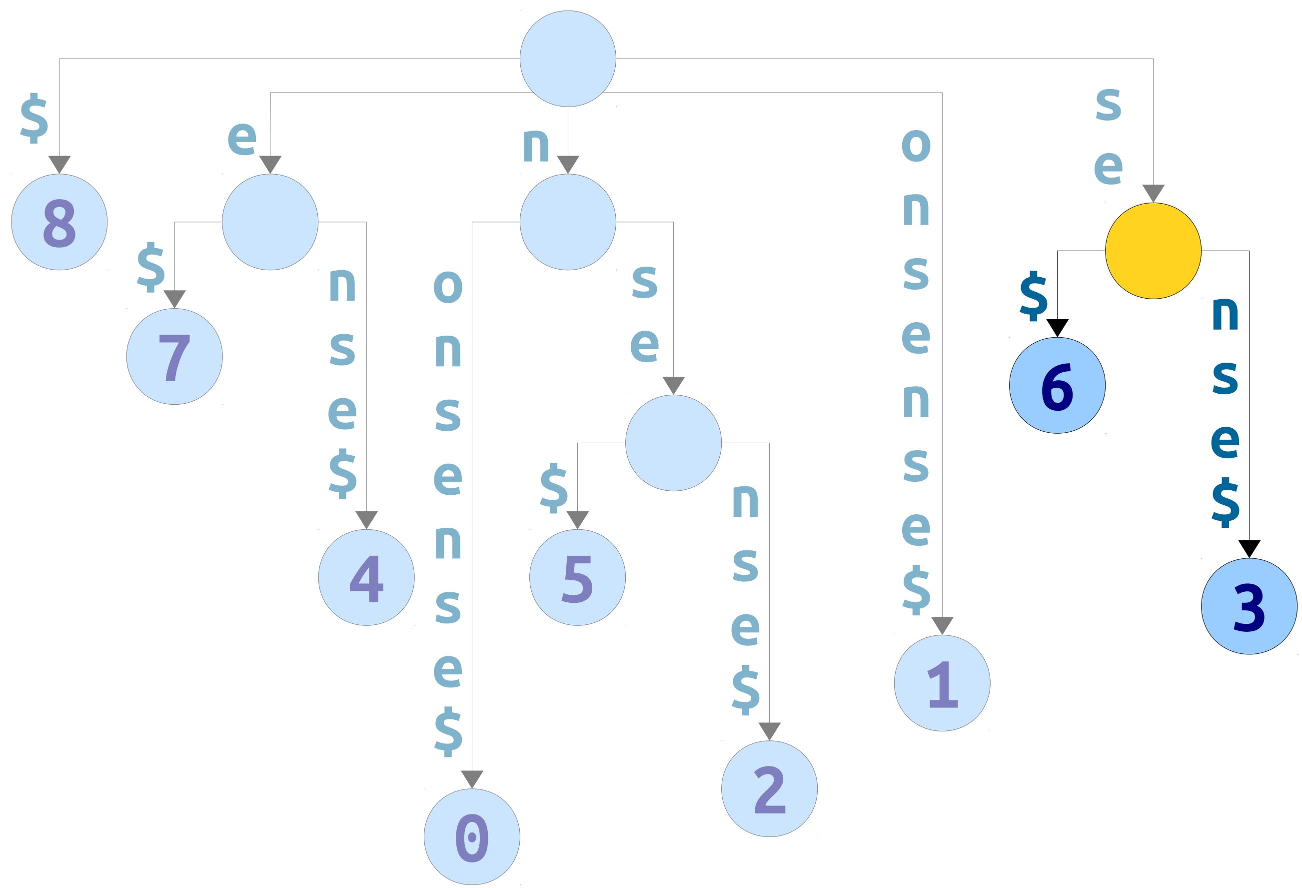
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O($\boldsymbol{n}$).
- Finding all matches takes time O($\boldsymbol{z}$).



nonsense$
012345678

Substring Search

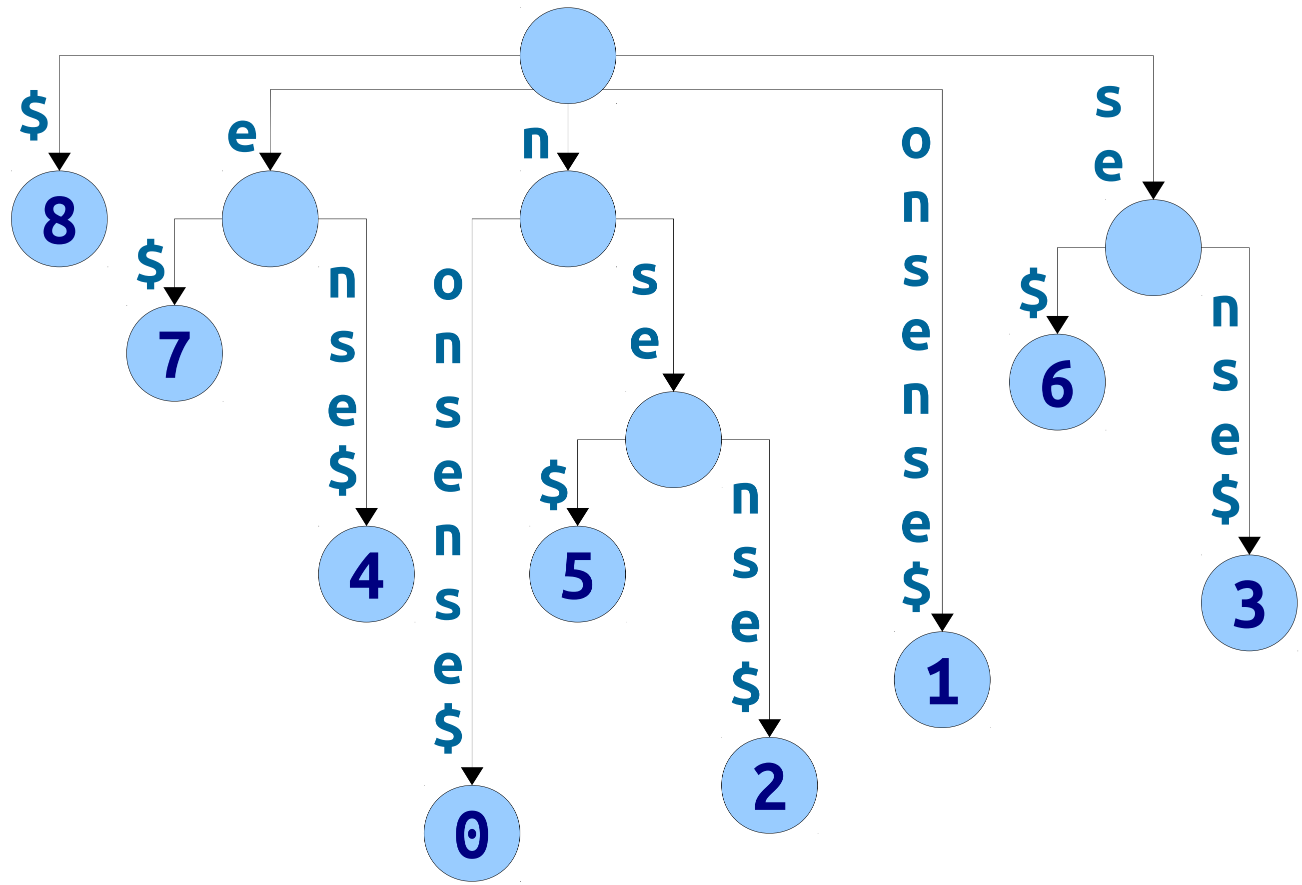
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



non<u>se</u>n<u>se</u>$
012345678

Substring Search

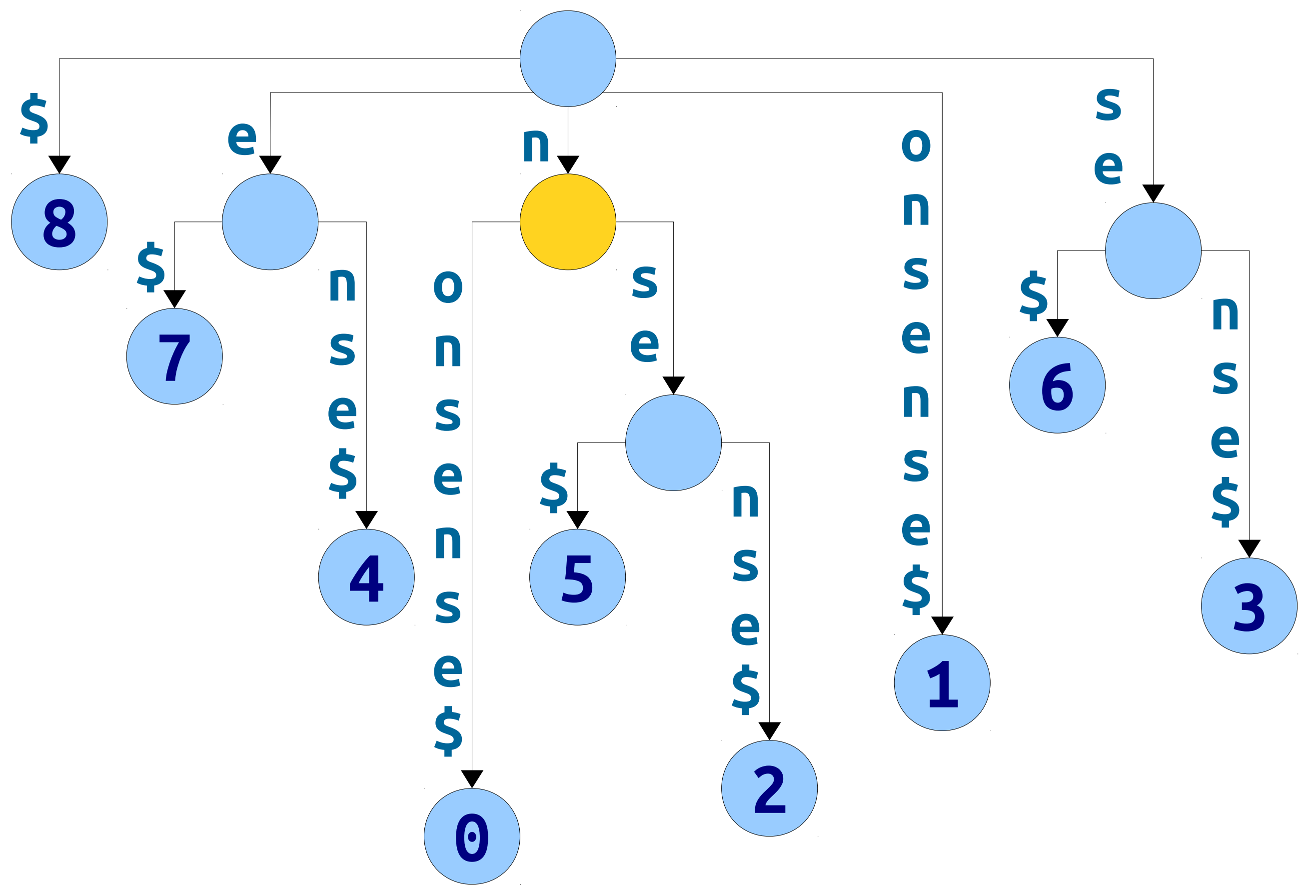
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



nonsense$
012345678

Substring Search

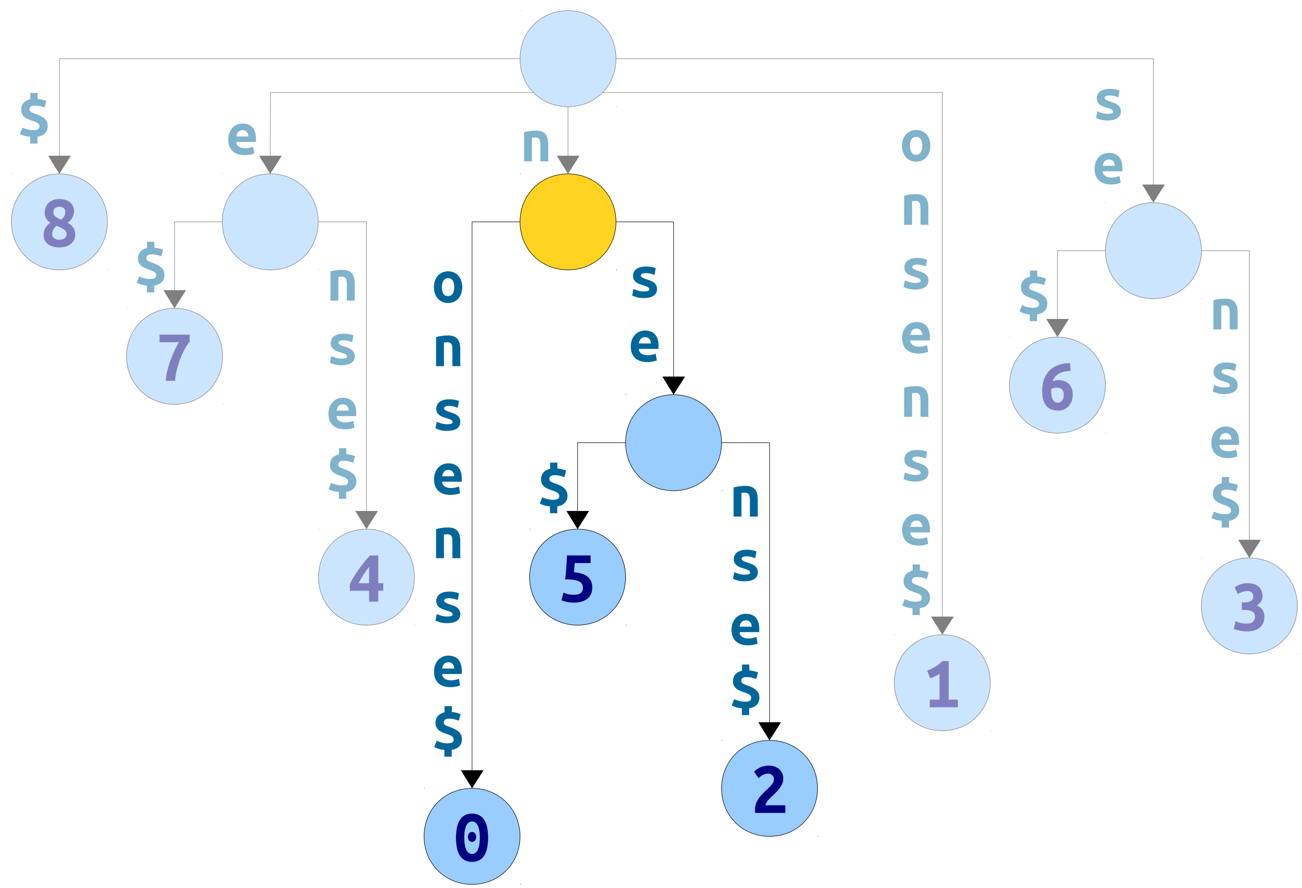
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



nonsense$
012345678

Substring Search

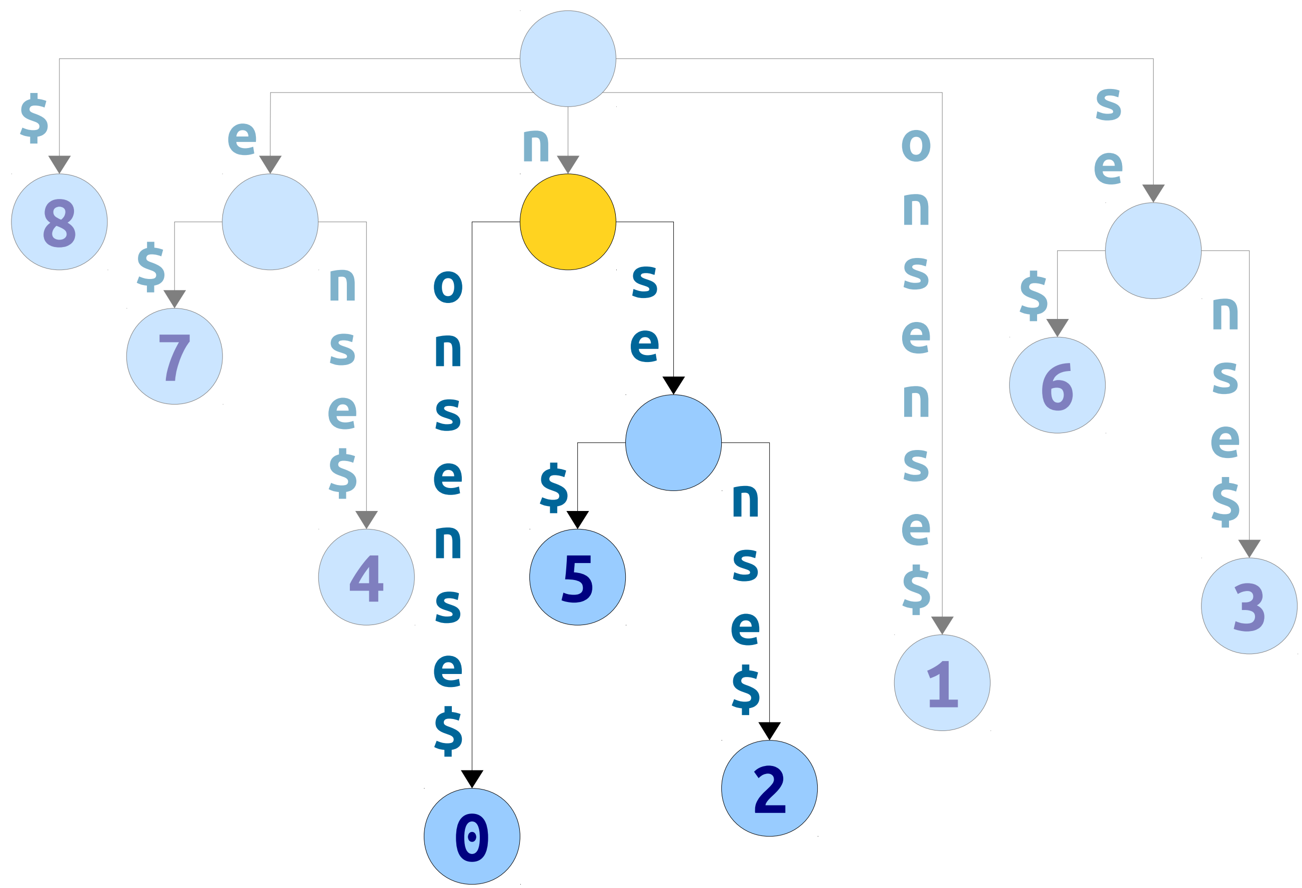
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O($\boldsymbol{n}$).
- Finding all matches takes time O($\boldsymbol{z}$).



nonsense$
012345678

Substring Search

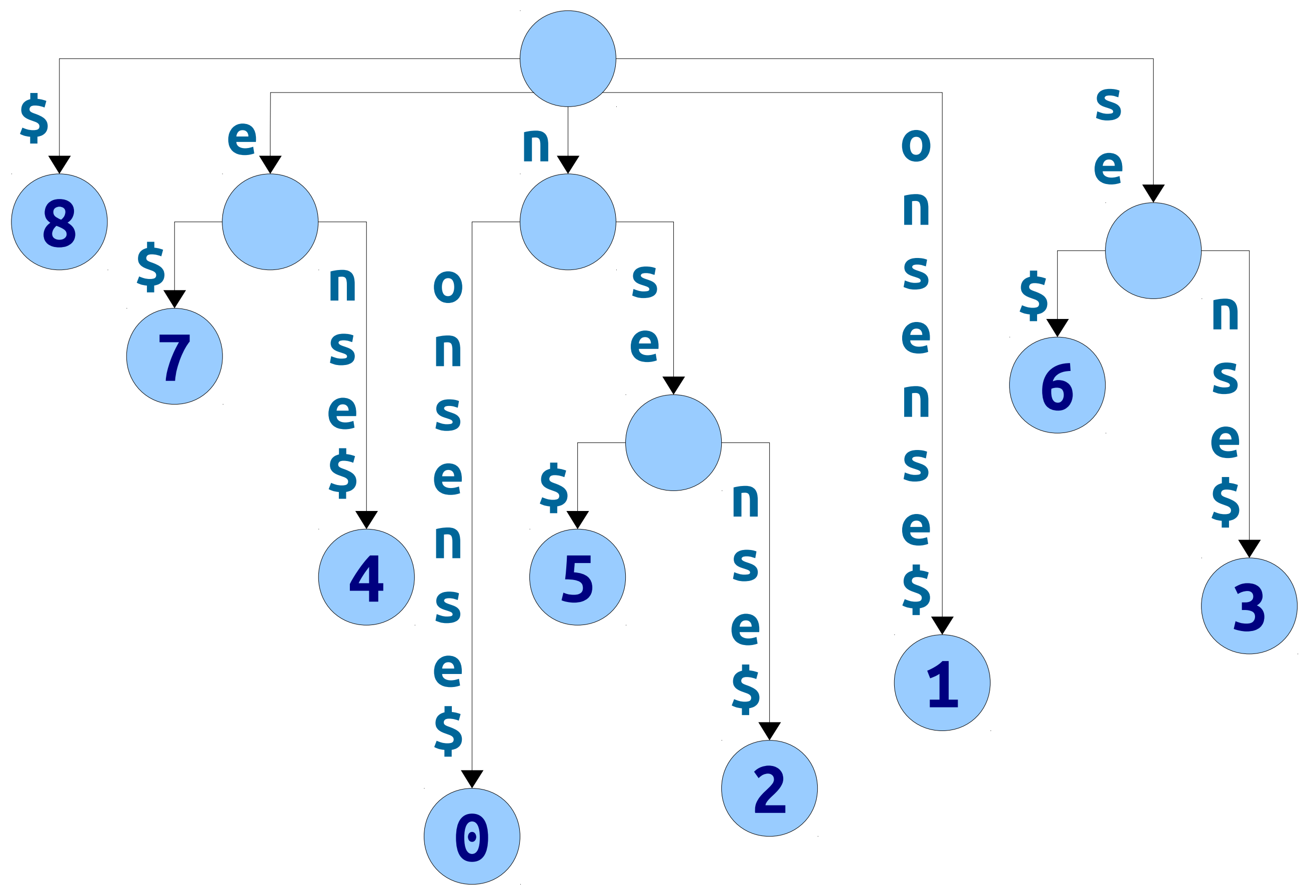
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(n).
- Finding all matches takes time O(z).



nonsense$
012345678

Substring Search

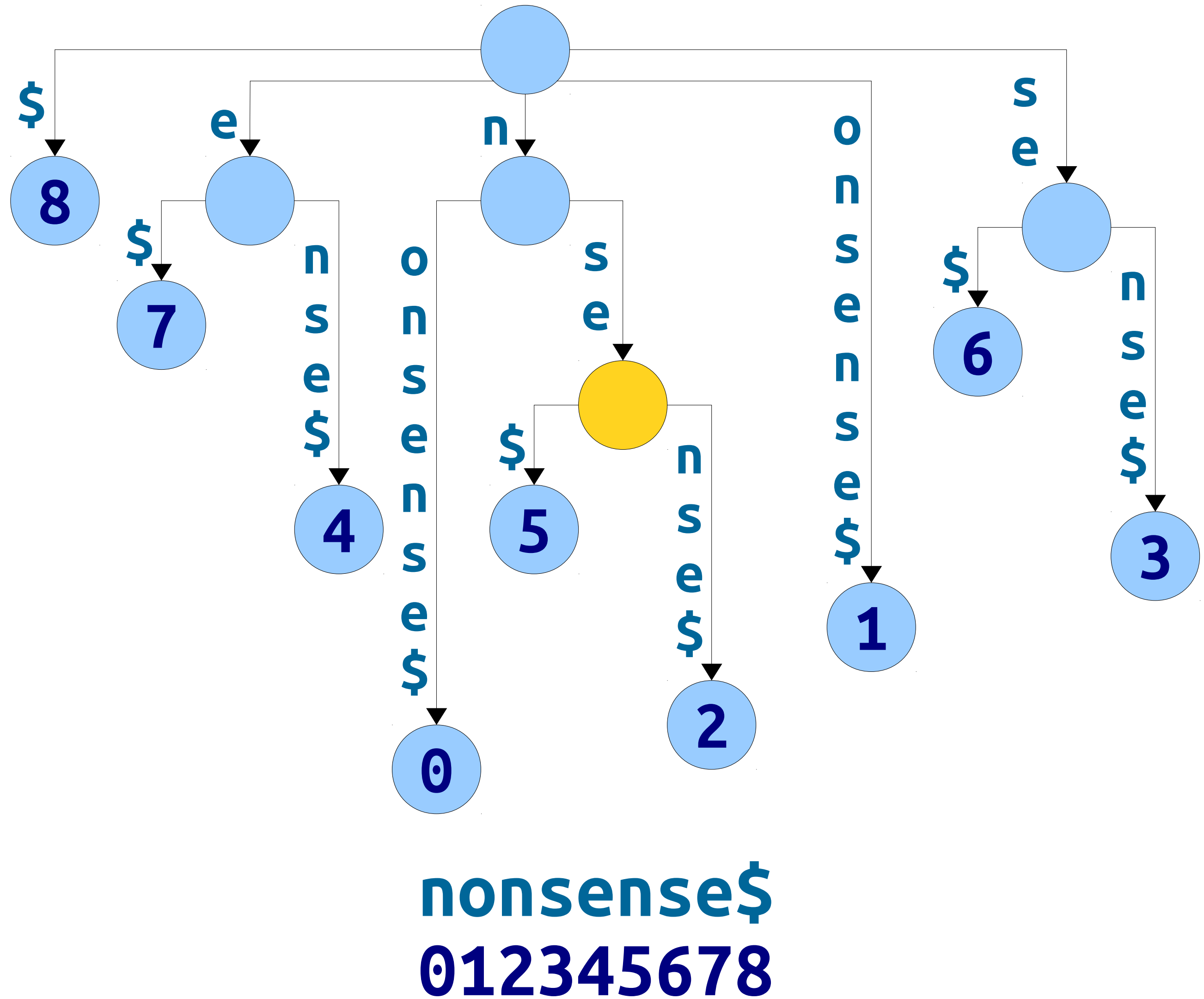
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O($\boldsymbol{n}$).
- Finding all matches takes time O($\boldsymbol{z}$).



nonsense$
012345678

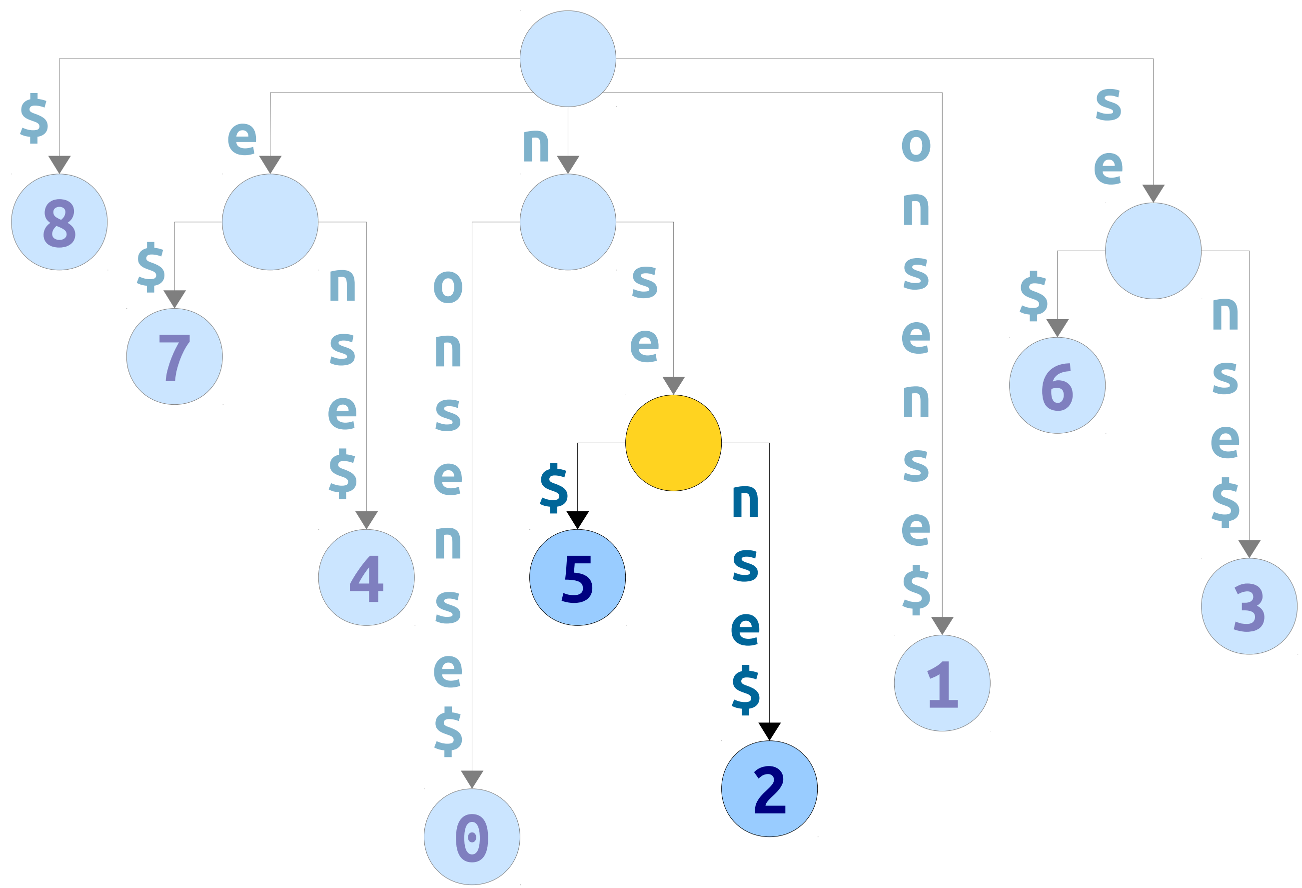
Substring Search

- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O($\boldsymbol{n}$).
- Finding all matches takes time O($\boldsymbol{z}$).



Substring Search

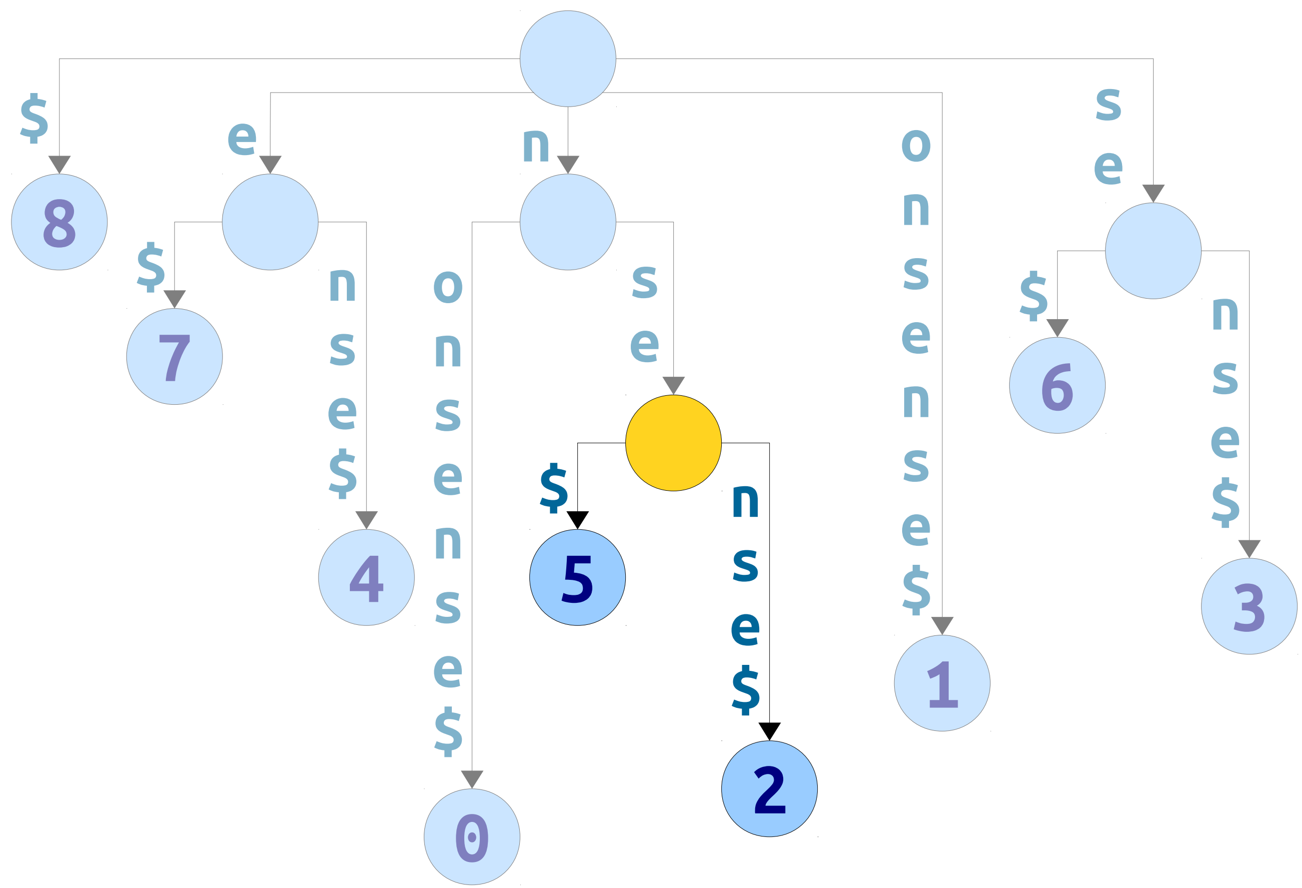
- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



nonsense$
012345678

Substring Search

- ***Algorithm:*** Use the standard Patricia trie search!
- Look up the pattern in the suffix tree, then use a DFS to find all matches.
- Looking up the pattern takes time O(***n***).
- Finding all matches takes time O(***z***).



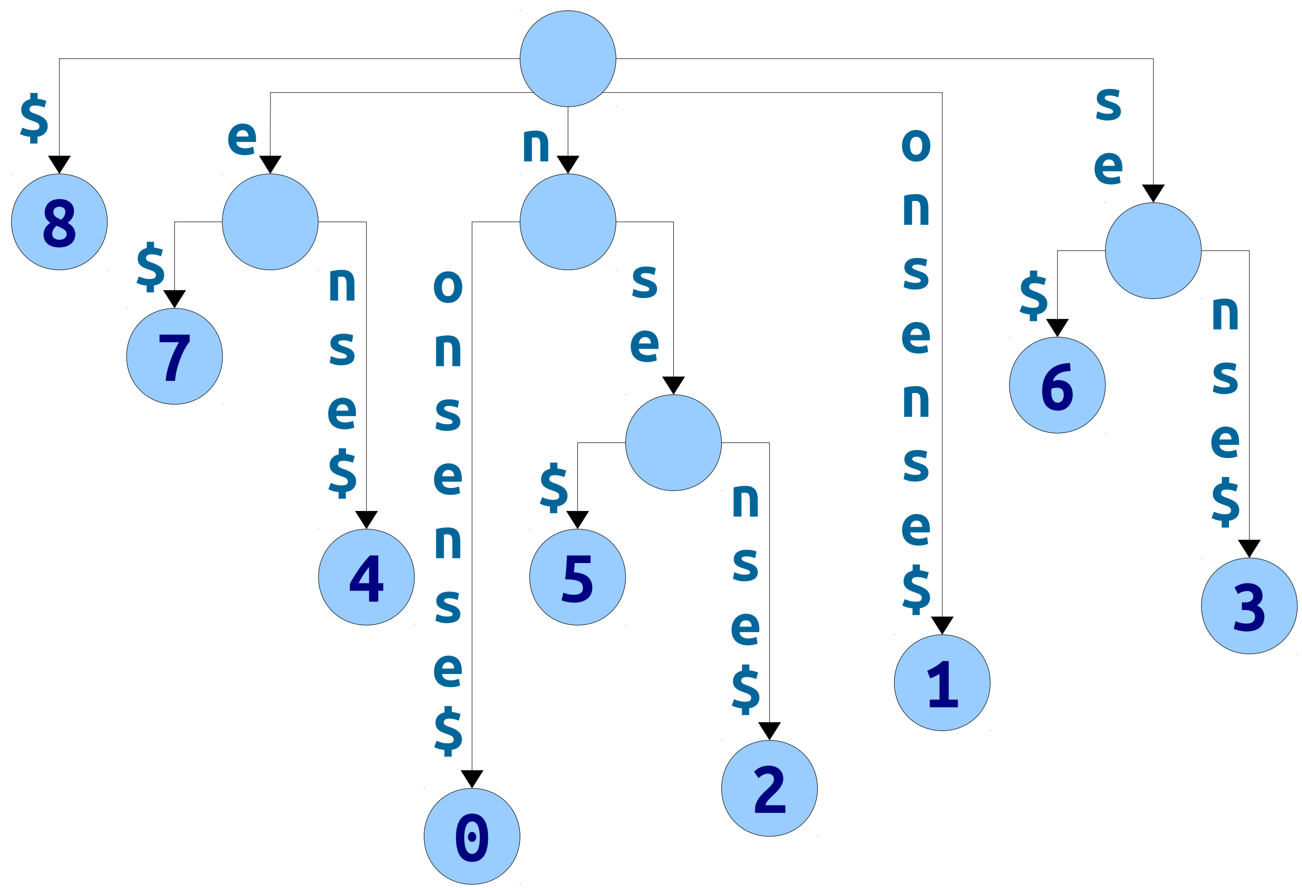
no<u>ns</u>e<u>ns</u>e$
012345678

Cancers often have repeated copies the same gene.

Given a cancer genome (length ~3,000,000,000), find the longest repeated DNA sequence.

The Anatomy of a Suffix Tree

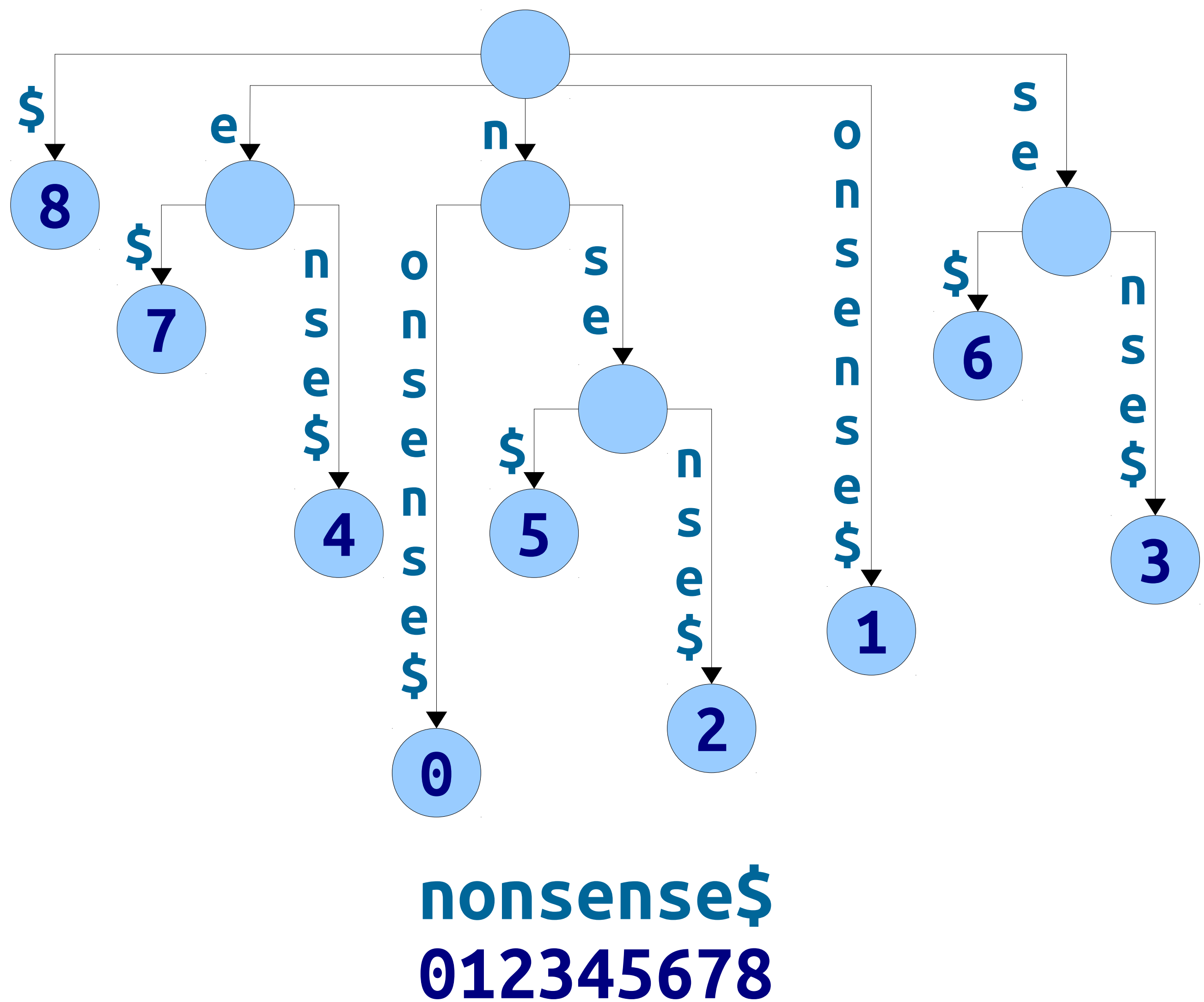
- Think back to Cartesian trees. We can describe them in two ways.
  - ***Mechnically:*** Hoist the minimum element up to the root, then recursively process the two subarrays.
  - ***Operationally:*** It's a min-heap whose inorder traversal gives the original array.
- We now have a ***mechanical*** definition of a suffix tree. Can we get can ***operational*** one?



nonsense$
012345678

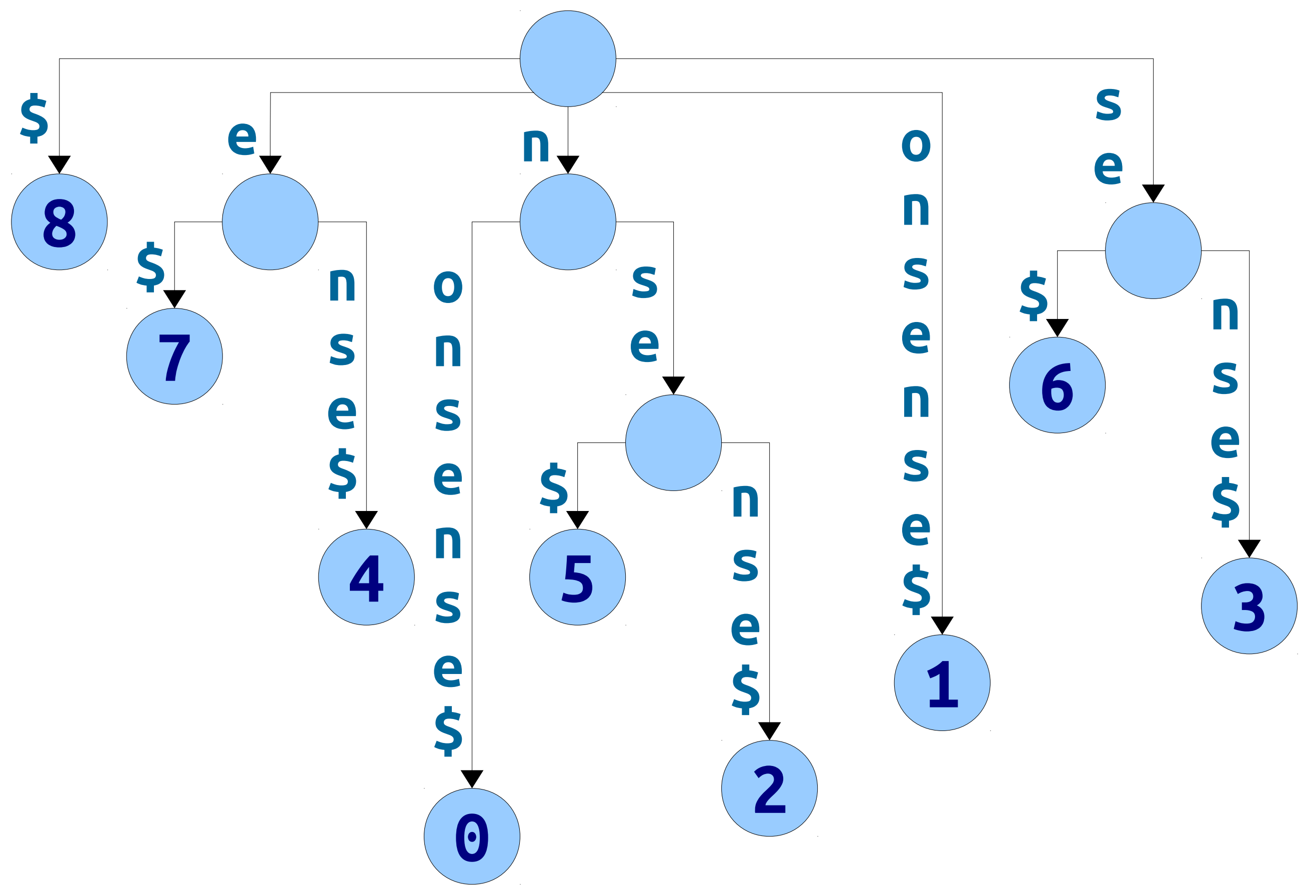
The Anatomy of a Suffix Tree

- The leaves of a suffix tree correspond to the suffixes of the text string *T*$.
- ***Question:*** What do the *internal* nodes of the suffix tree correspond to?



The Anatomy of a Suffix Tree

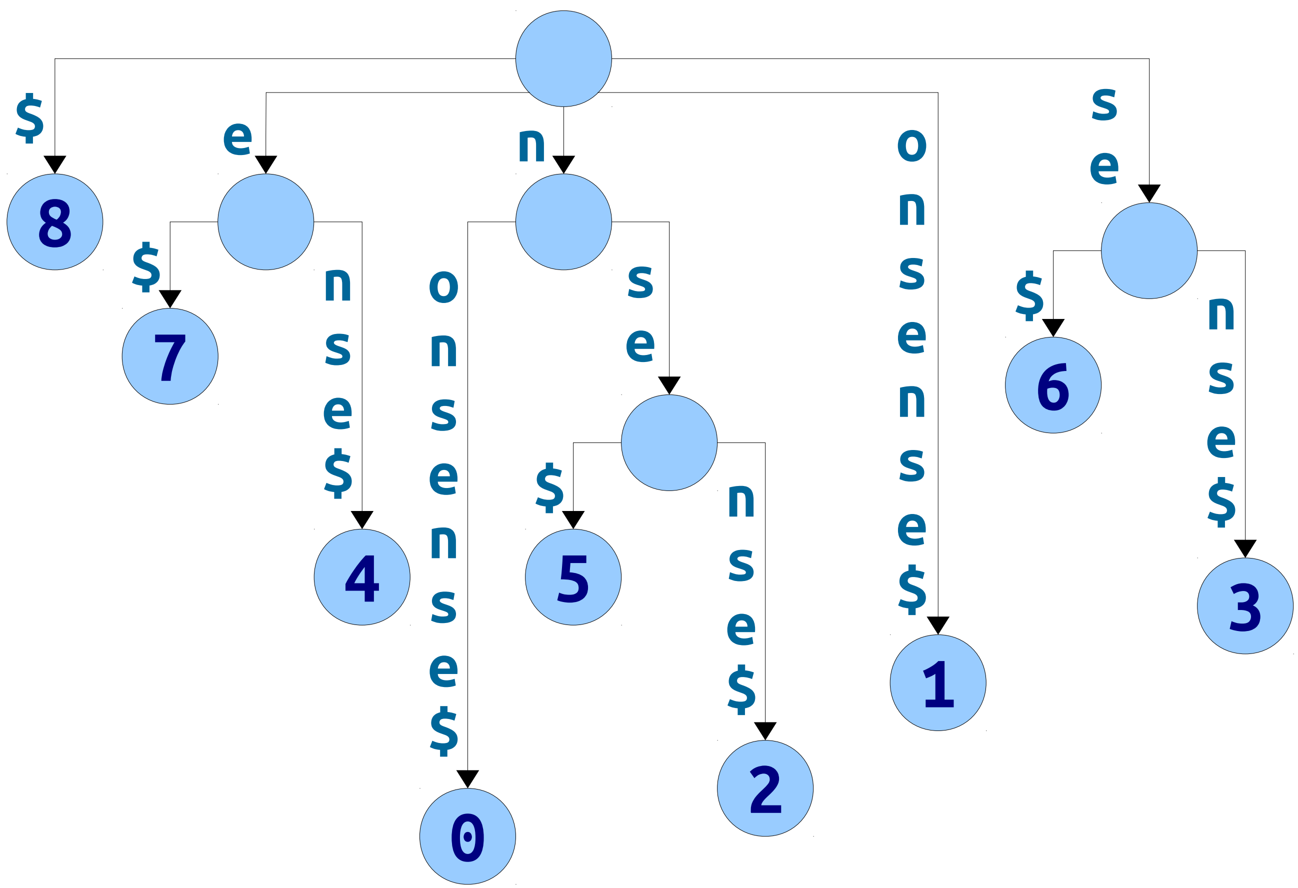
- In this suffix tree, there are internal nodes for the substrings **e**, **n**, **nse**, and **se**.
- All these substrings appear at least twice in the original string!
- More generally: if there is an internal node for a substring α, then α appears at least twice in the original text.



nonsense$
012345678

The Anatomy of a Suffix Tree

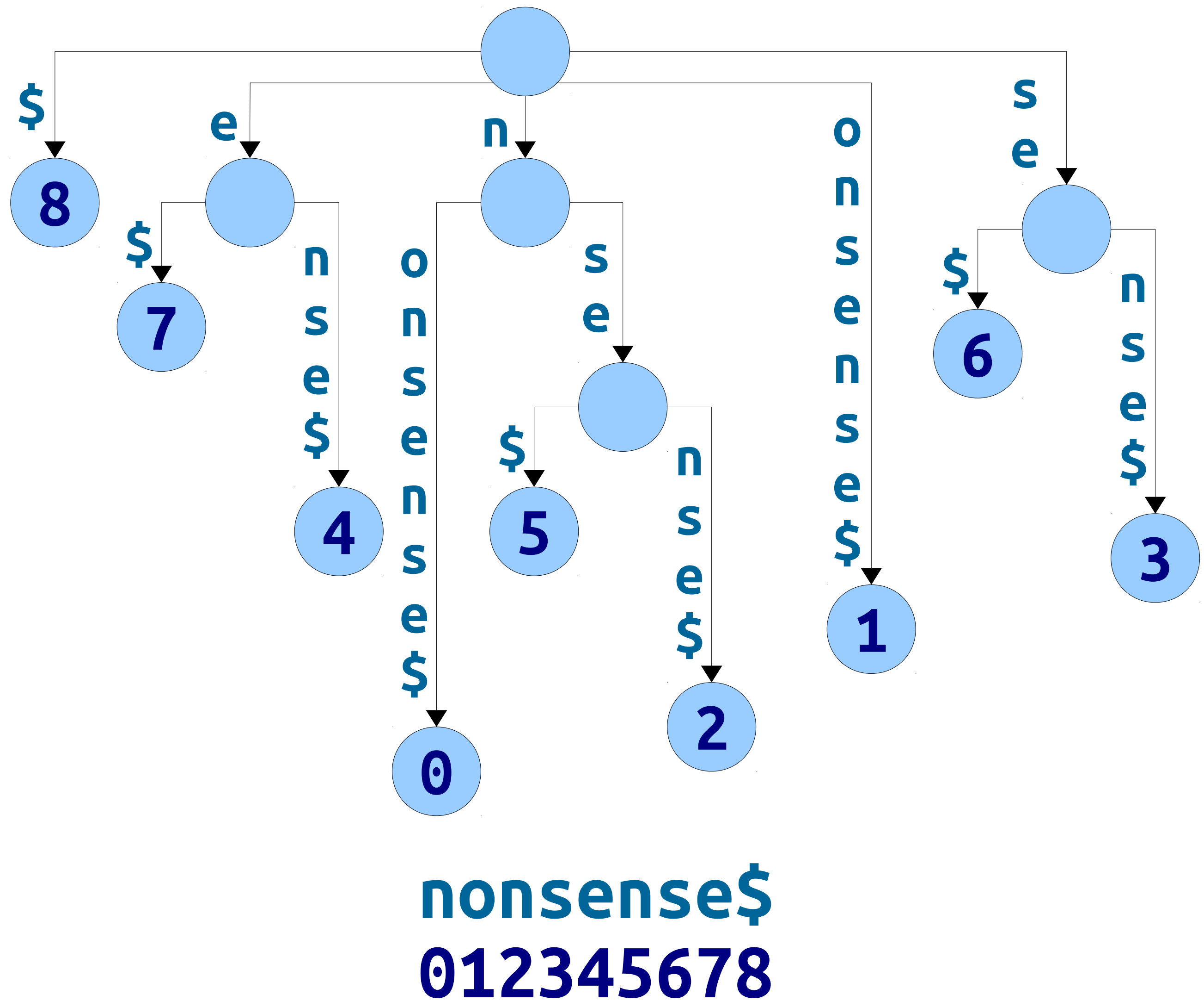
- ***Question:*** why *is* there an internal node for the substring **n**, but *isn't* there an internal node for the substring **ns**?
- *Every* occurrence of **ns** can be extended by appending the same character (**e**)
- *Not all* occurrences of **n** can be extended by appending the same character.



nonsense$
012345678

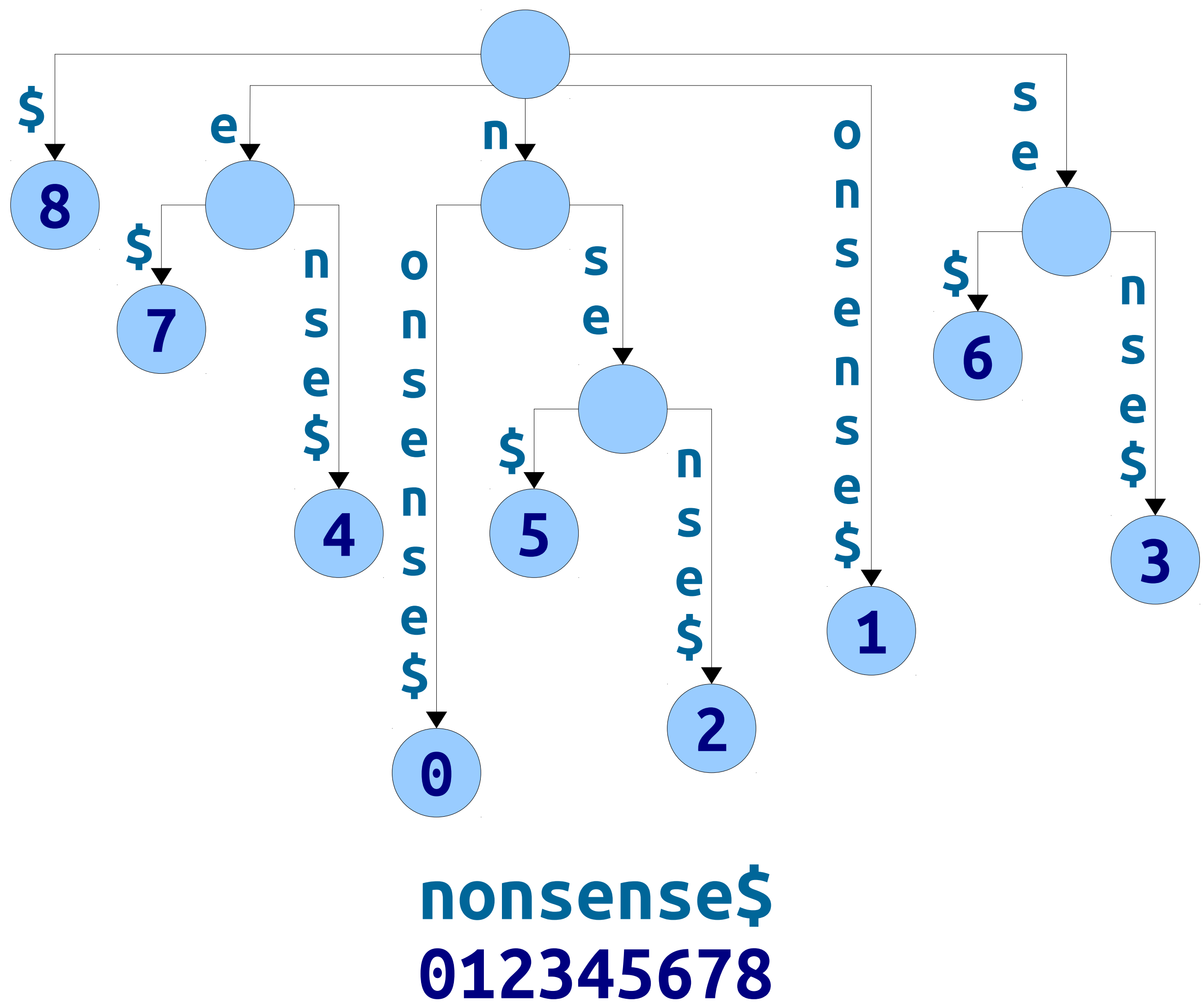
The Anatomy of a Suffix Tree

- A ***branching word*** in T is a string ω where there are characters $a \neq b$ such that ωa and ωb are substrings of $T\$$.
- ***Theorem:*** The suffix tree for a string T has an internal node for a string ω if and only if ω is a branching word in T.



The Anatomy of a Suffix Tree

- Summarizing those previous points:
  - Every *leaf* corresponds to a *suffix*.
  - Every internal node corresponds to a *branching word*.
- Keep these intuitions in mind; they're great for understanding how suffix trees work!



Longest Repeated Substrings

- ***Theorem:*** The longest repeated substring of a string T must be a branching word in T.
- ***Proof idea:*** If ω isn't branching, it can't be the longest repeated substring.

f	l	i	b	b	e	r	t	i	g	i	b	b	e	t

Longest Repeated Substrings

- ***Theorem:*** The longest repeated substring of a string T must be a branching word in T.
- ***Proof idea:*** If ω isn't branching, it can't be the longest repeated substring.

The substring `berti` isn't repeated.

It therefore can't be the longest repeated substring.

f	l	i	b	**b**	**e**	**r**	**t**	**i**	g	i	b	b	e	t

Longest Repeated Substrings

- ***Theorem:*** The longest repeated substring of a string T must be a branching word in T.
- ***Proof idea:*** If ω isn't branching, it can't be the longest repeated substring.

f	l	i	b	b	e	r	t	i	g	i	b	b	e	t

Longest Repeated Substrings

- ***Theorem:*** The longest repeated substring of a string *T* must be a branching word in *T*.
- ***Proof idea:*** If ω isn't branching, it can't be the longest repeated substring.

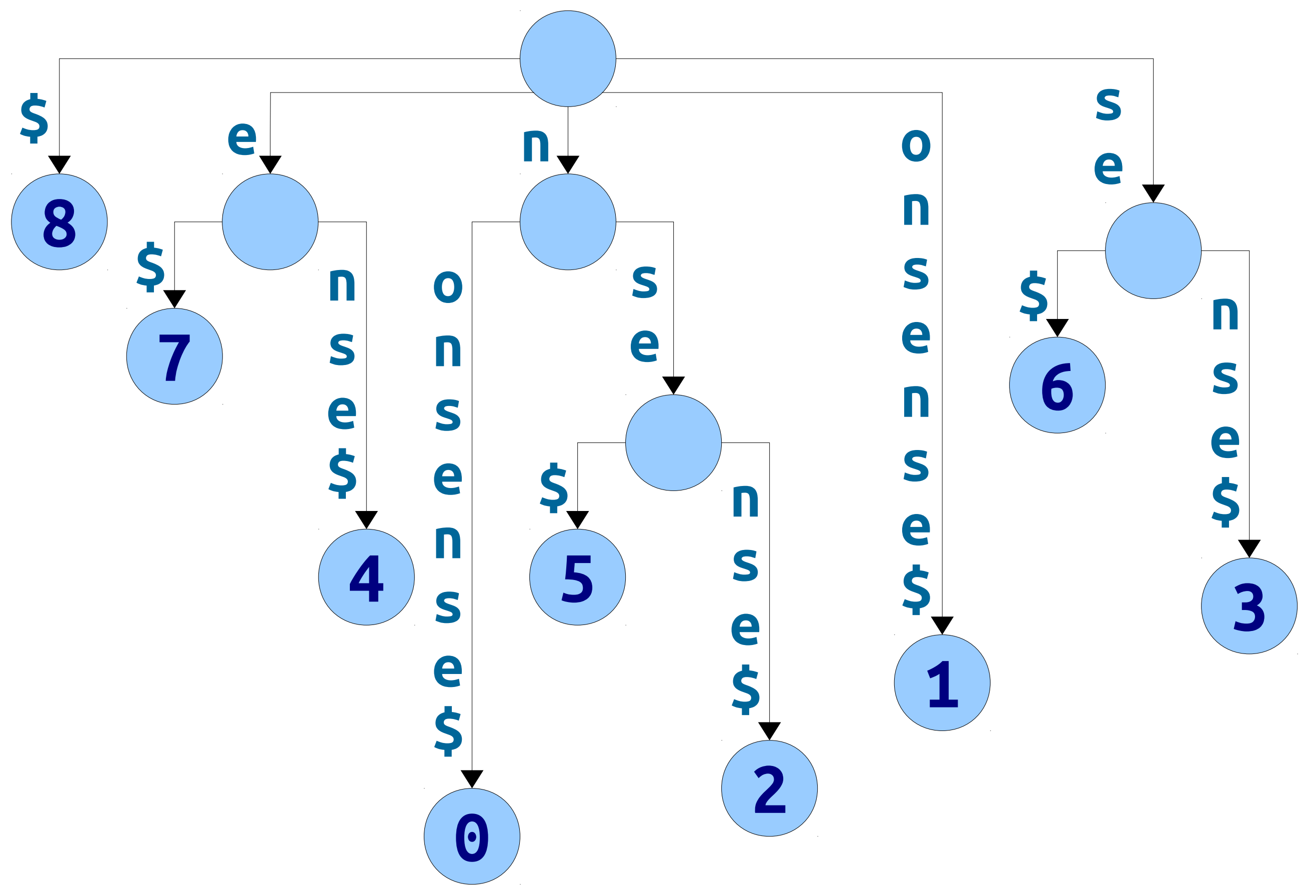
Every instance of `bb` can be extended to `bbe`.

It therefore can't be the longest repeating substring.

f	l	i	b	b	e	r	t	i	g	i	b	b	e	t

Longest Repeated Substrings

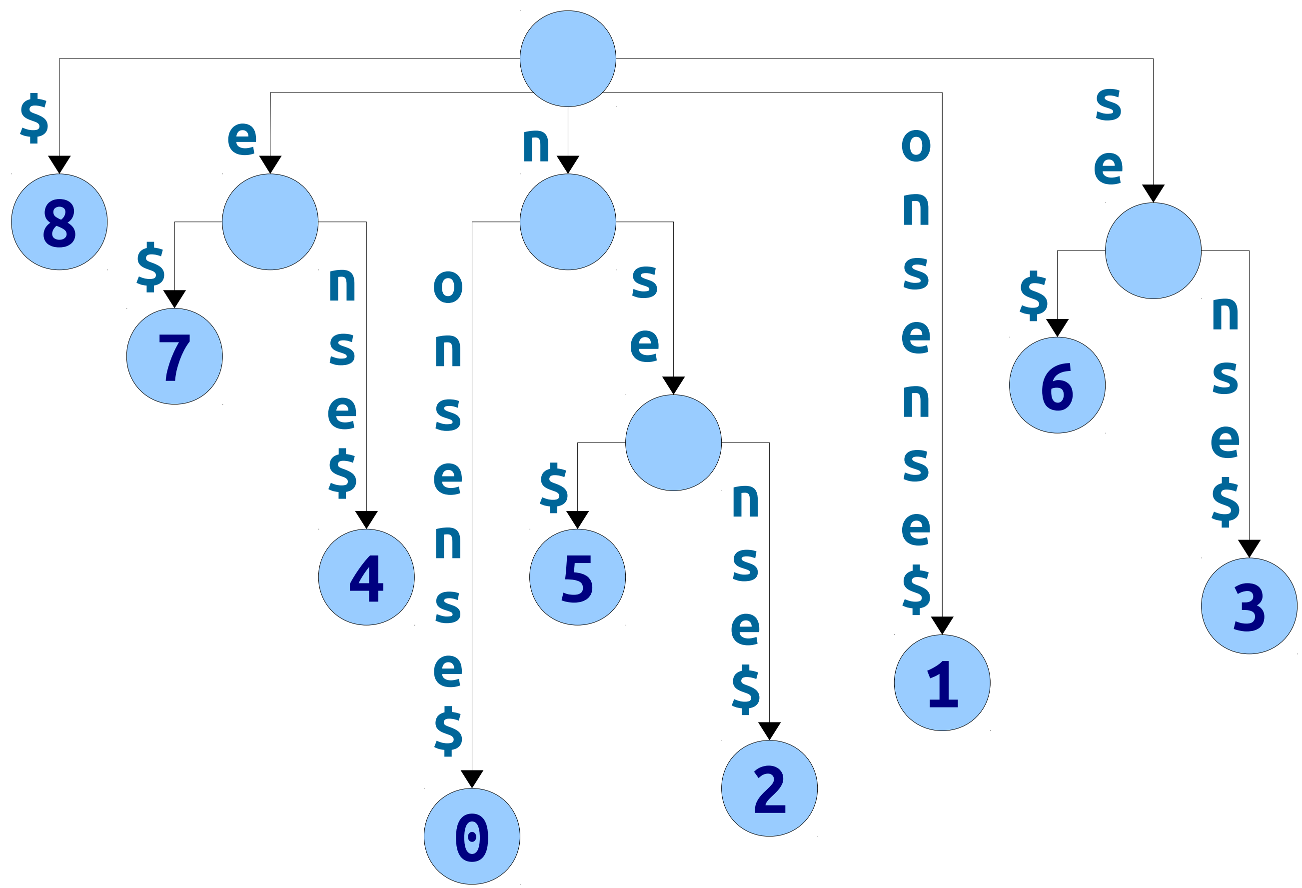
- ***Theorem:*** The longest repeated substring of T is a branching word in T.
- To find the longest repeated substring of a string T, we just need to find the internal node with the longest label!



nonsense$
012345678

Longest Repeated Substrings

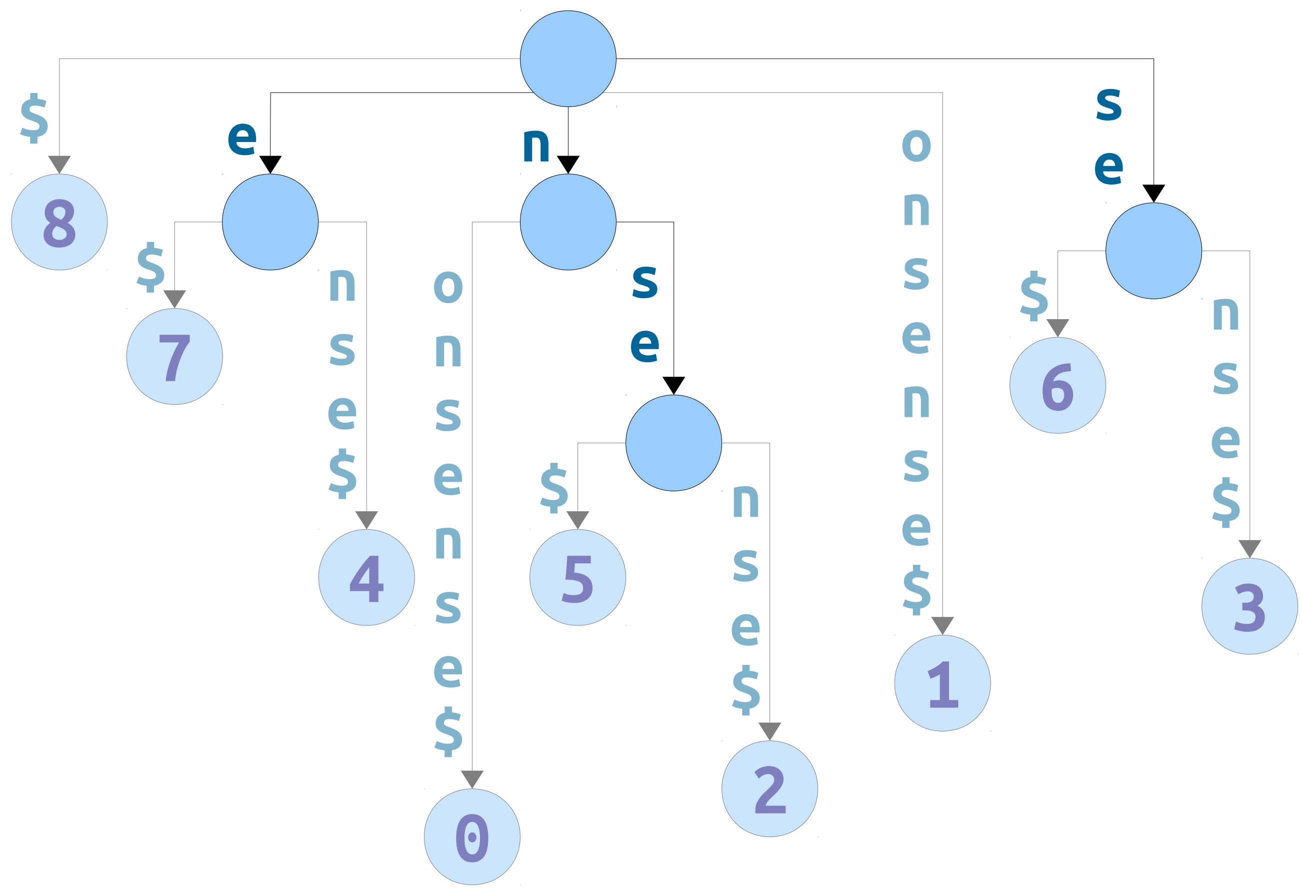
- Given a suffix tree for a string *T* of length ***m***, there is an O(***m***)-time algorithm for finding the longest repeated substring of ***m***.
- ***Basic idea:*** Run a DFS over the tree and find the internal node with the longest string on its path from the root.
- There are some subtle details required to get this to run in time O(***m***). Think this over! See what you find.



nonsense$
012345678

Longest Repeated Substrings

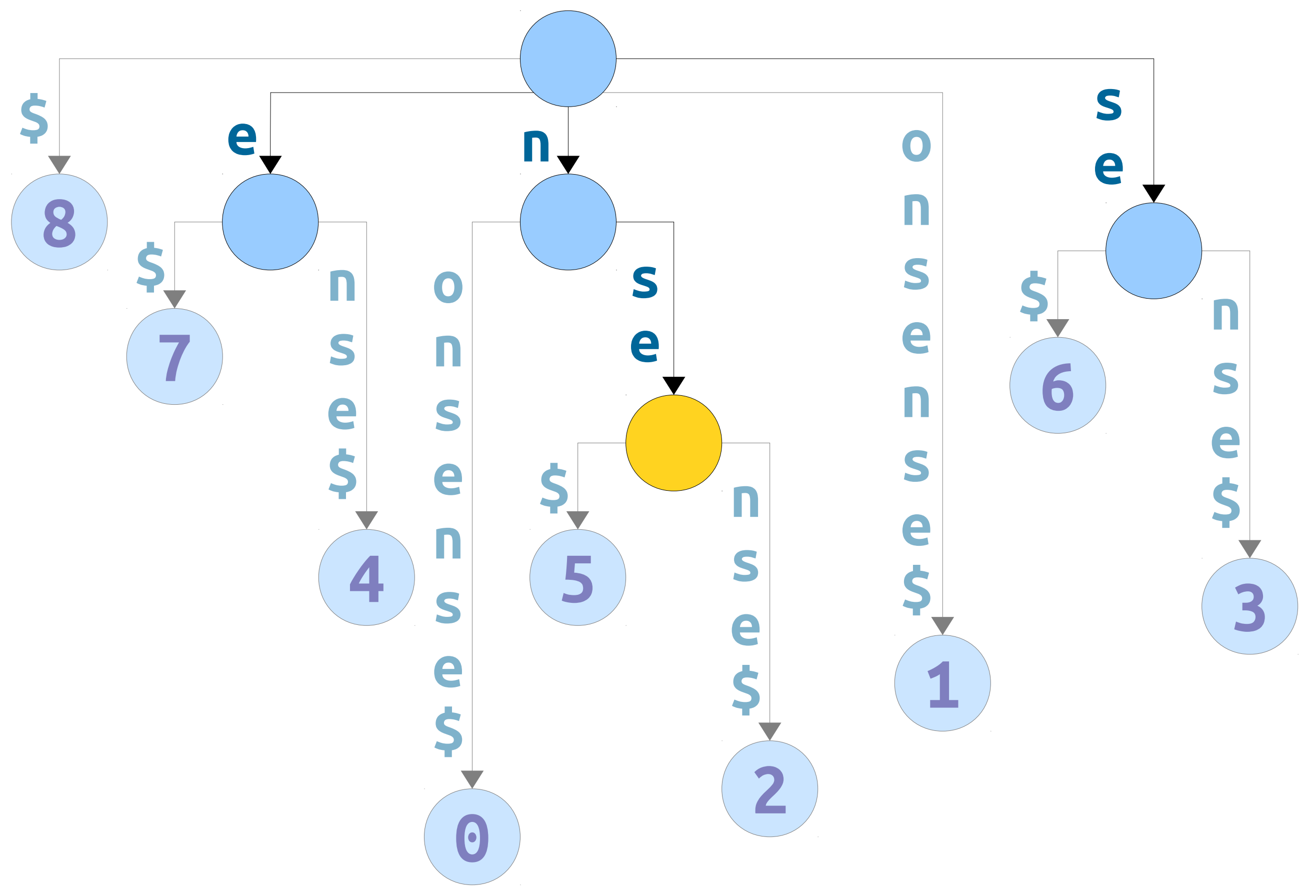
- Given a suffix tree for a string T of length $\boldsymbol{m}$, there is an $O(\boldsymbol{m})$-time algorithm for finding the longest repeated substring of $\boldsymbol{m}$.
- ***Basic idea:*** Run a DFS over the tree and find the internal node with the longest string on its path from the root.
- There are some subtle details required to get this to run in time $O(\boldsymbol{m})$. Think this over! See what you find.



nonsense$
012345678

Longest Repeated Substrings

- Given a suffix tree for a string T of length m, there is an O(m)-time algorithm for finding the longest repeated substring of m.
- ***Basic idea:*** Run a DFS over the tree and find the internal node with the longest string on its path from the root.
- There are some subtle details required to get this to run in time O(m). Think this over! See what you find.



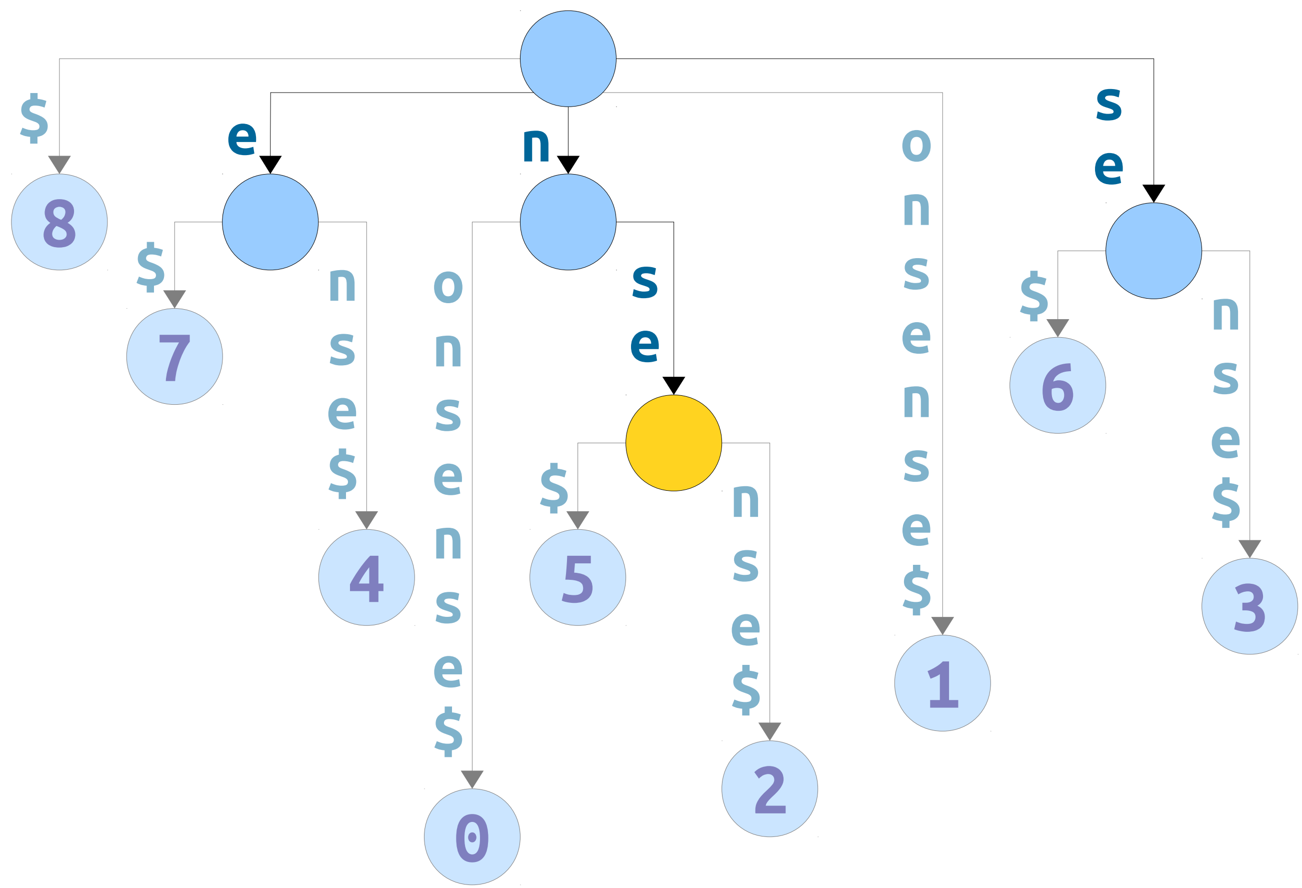
nonsense$
012345678

Longest Repeated Substrings

- Given a suffix tree for a string T of length $\boldsymbol{m}$, there is an O($\boldsymbol{m}$)-time algorithm for finding the longest repeated substring of m.

  Basic idea: Run a DFS over the tree and find the internal node with the longest string on its path from the root.

  There are some subtle details required to get this to run in time O($\boldsymbol{m}$). Think this over! See what you find.

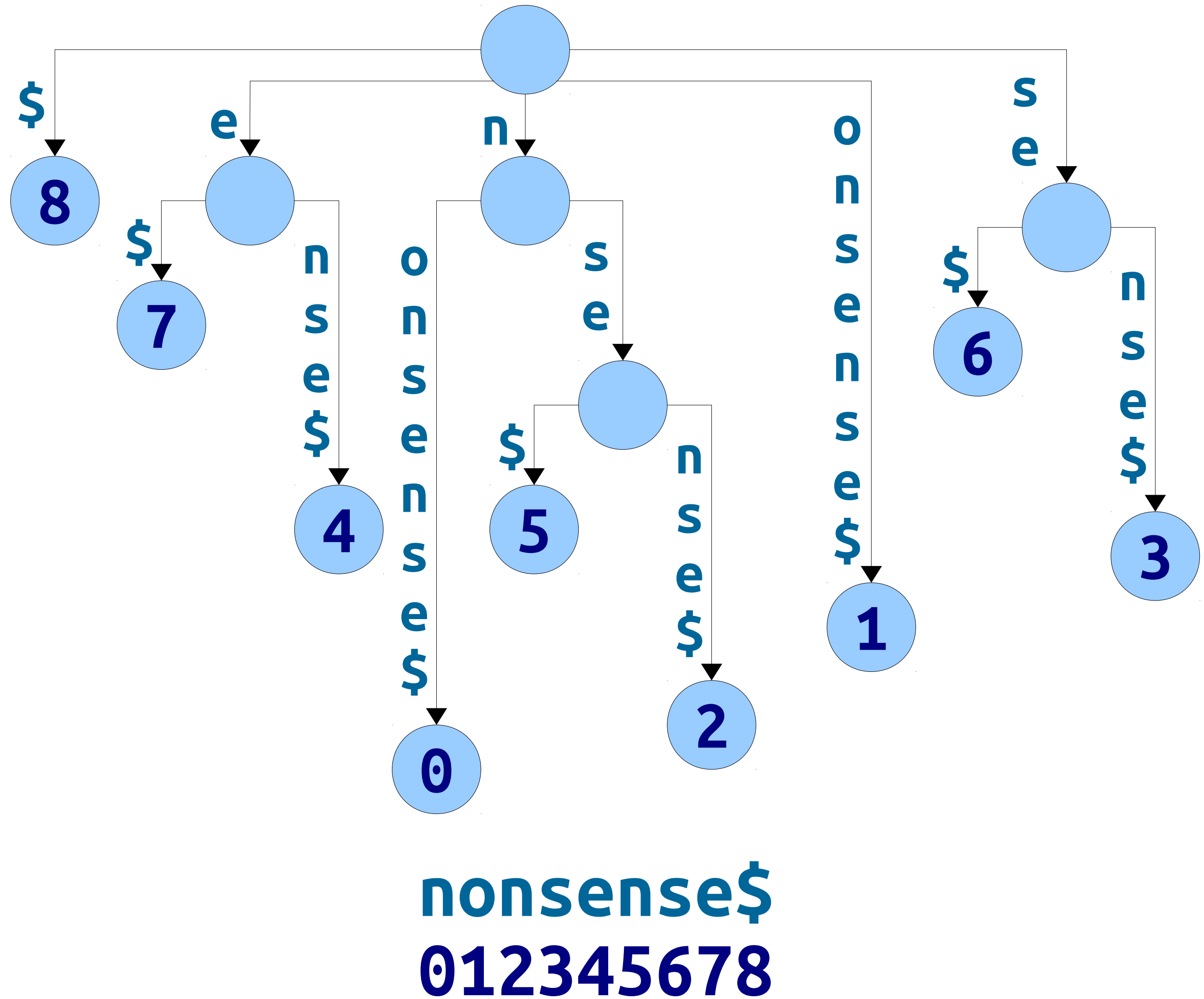


nonsense$
012345678

Representing Suffix Trees

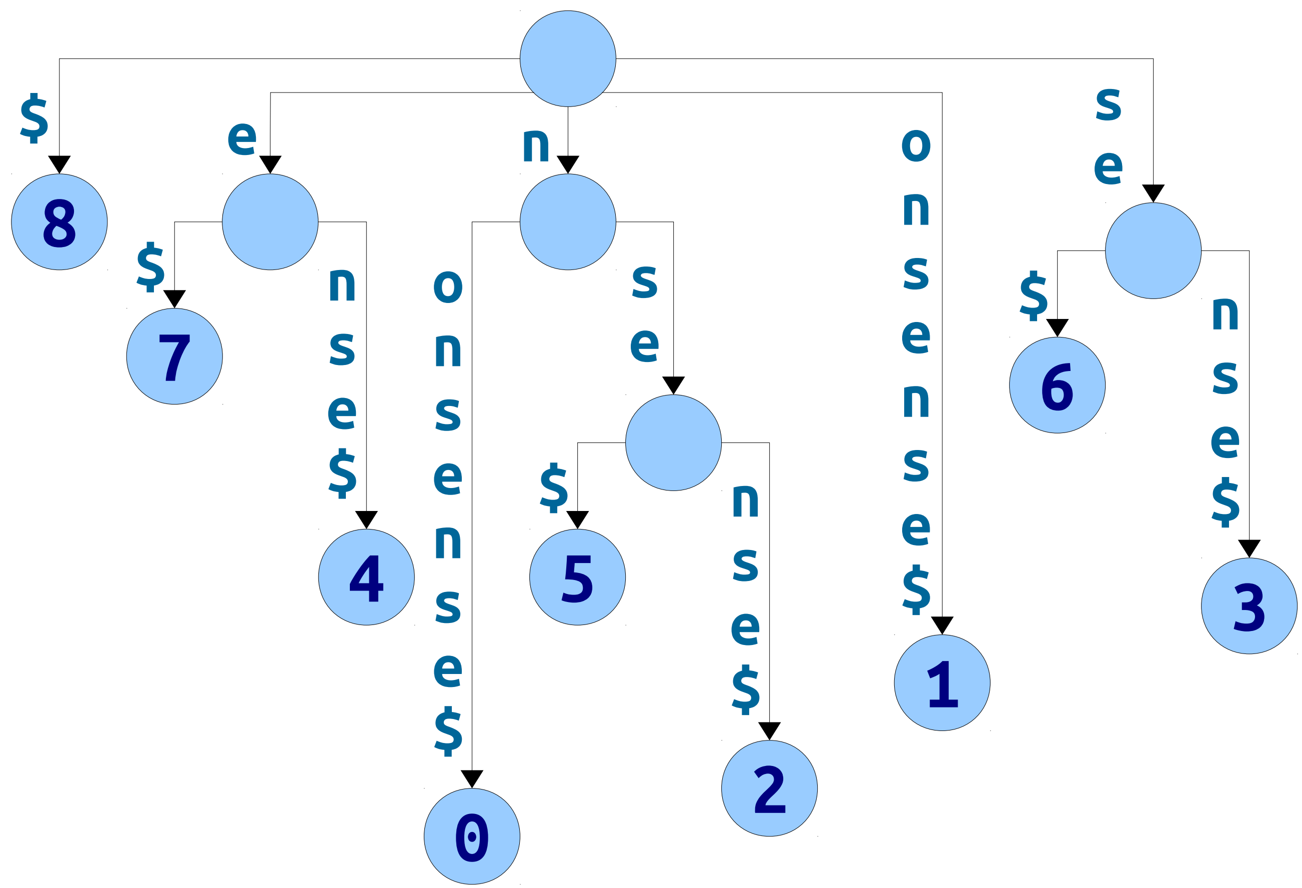
Representing a Suffix Tree

- We know that a suffix tree has O(***m***) nodes, where ***m*** is the number of characters in the input string.
- This means that there are O(***m***) edges. *(Why?)*
- ***Question:*** Why can't we immediately claim that the space usage of the suffix tree is O(***m***)?



Representing a Suffix Tree

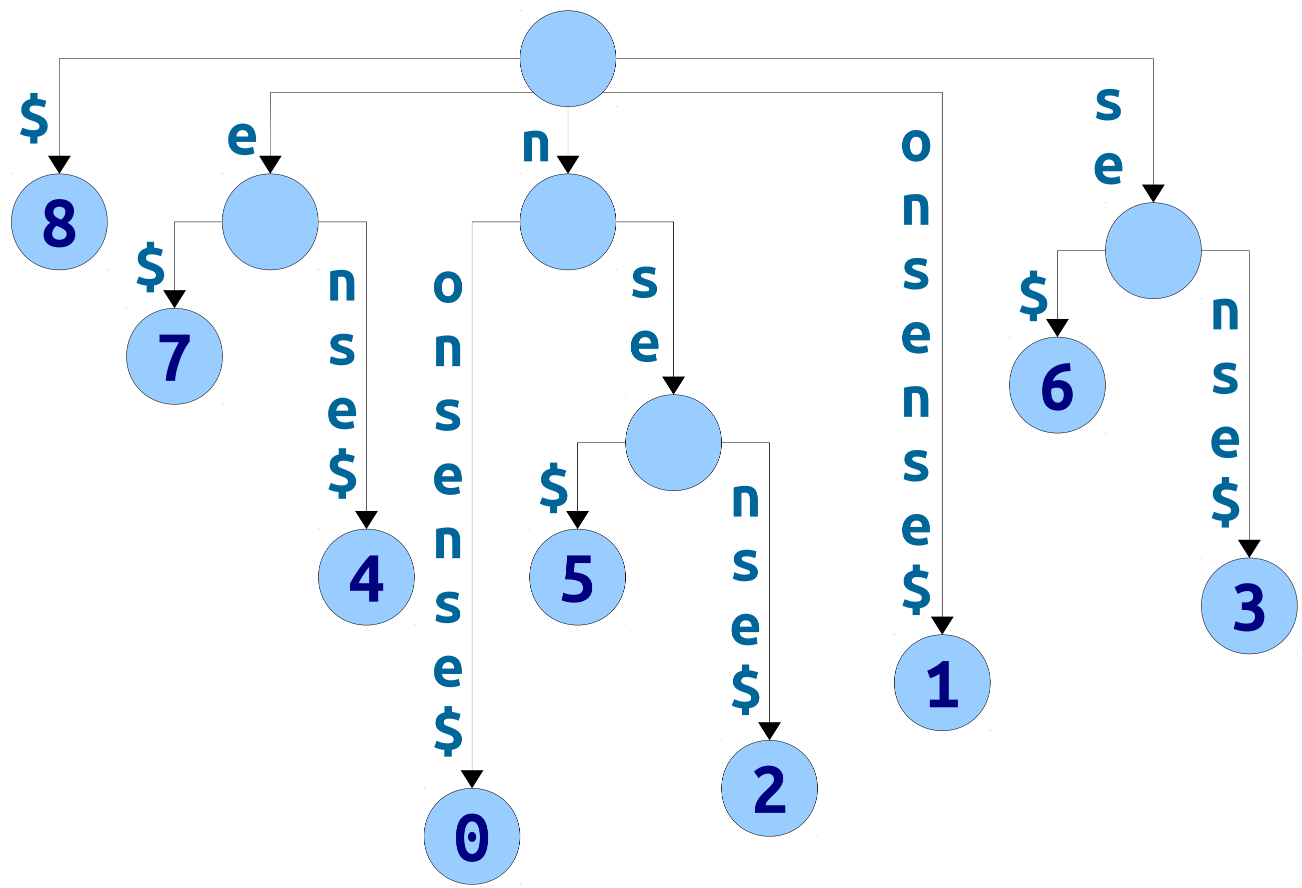
- ***Claim:*** Writing out all suffixes of a string of length m requires $\Theta(m^2)$ characters.
- ***Proof idea:*** Those suffixes have length $1 + 2 + \ldots + (m+1)$, factoring in the special **$** character.
- ***Problem:*** It is indeed possible to build a suffix tree with $\Theta(m^2)$ total letters on the edges.



nonsense$
012345678

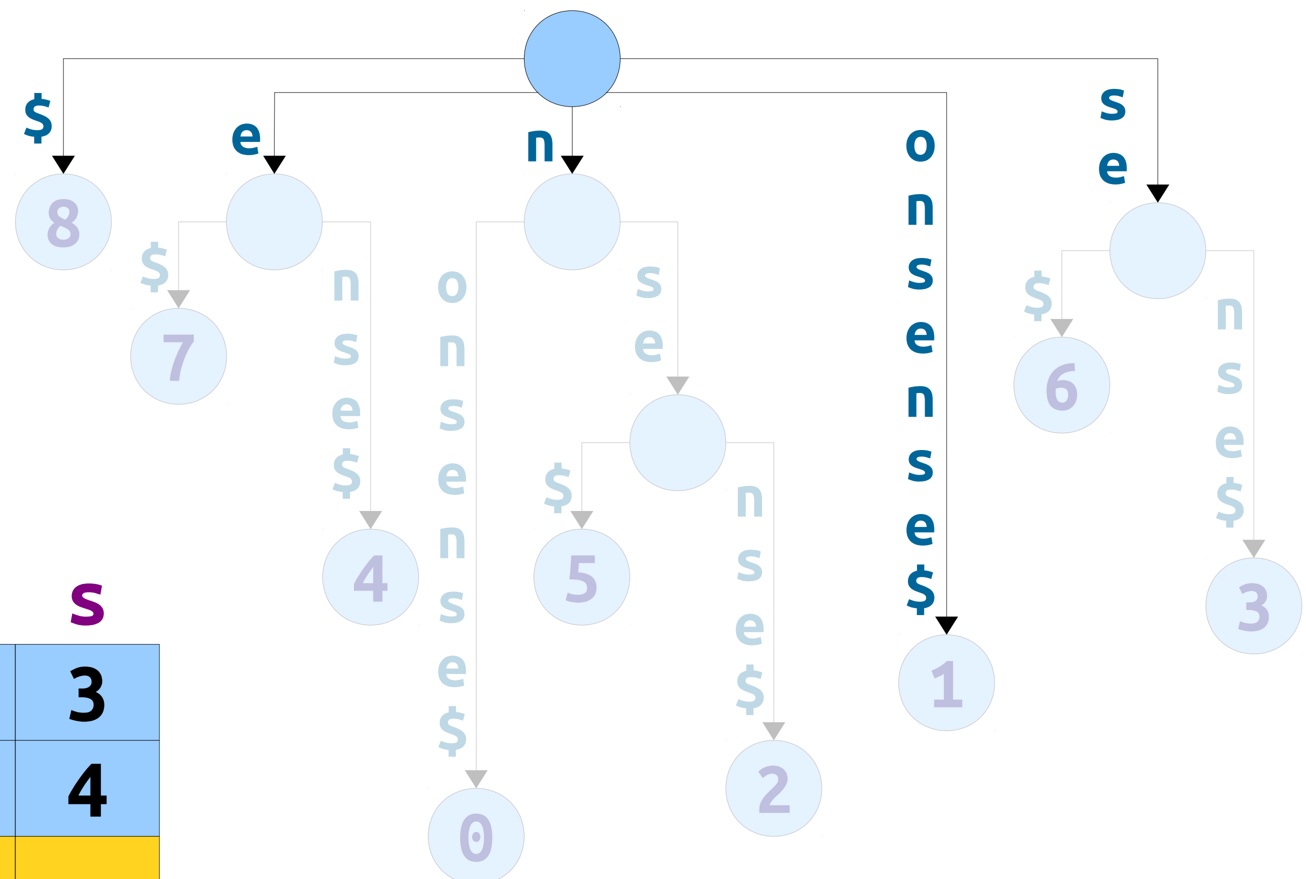
Representing a Suffix Tree

- By being clever with our representation, we can guarantee that a suffix tree uses only Θ(***m***) space, regardless of the input string.
- ***Observation:*** Each edge is labeled with a substring of the original input string.
- ***Idea:*** Don't actually write out the labels on the edges. Just write down the start and end index!



nonsense$
012345678

Representing a Suffix Tree

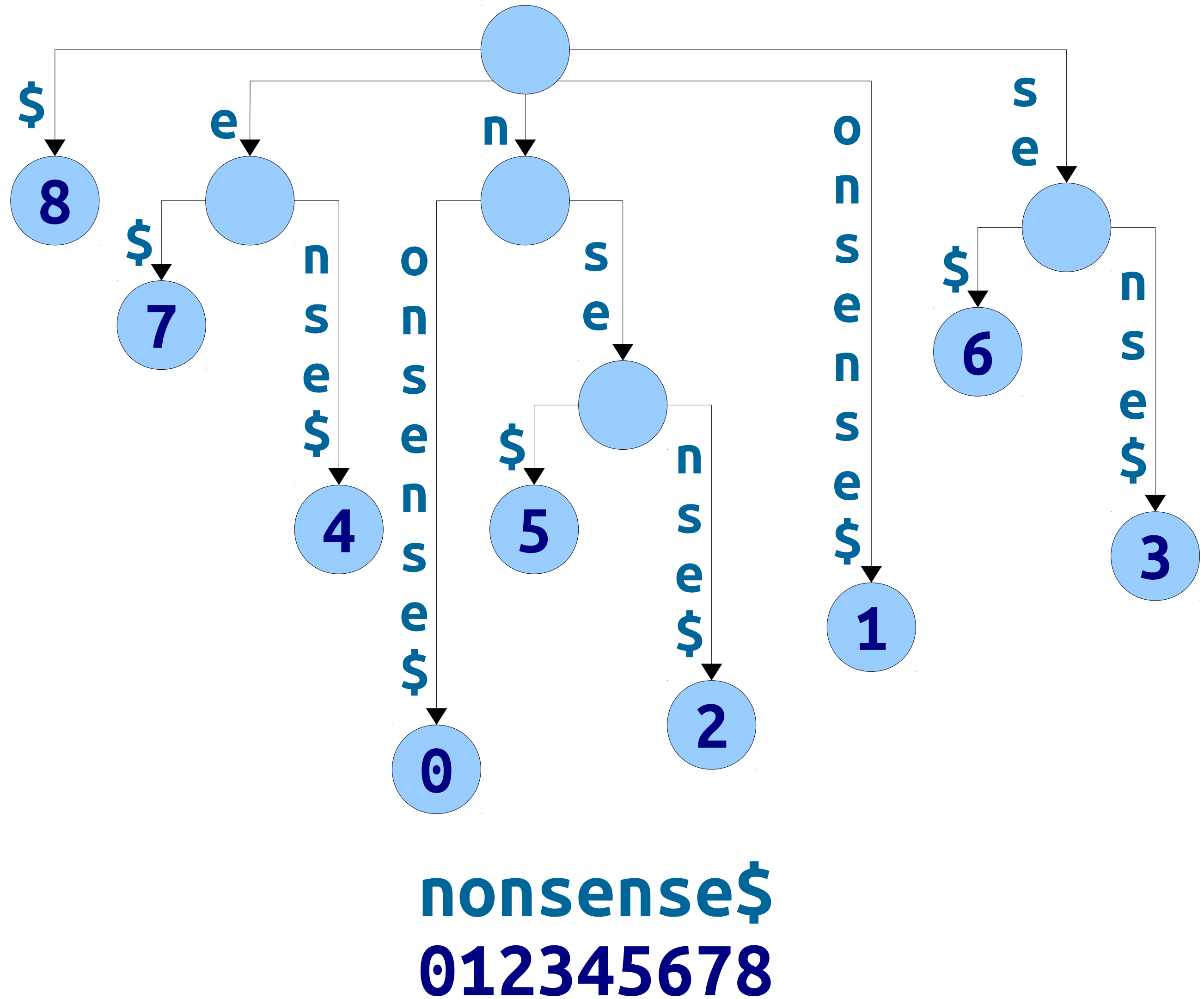


	$	e	n	o	s
start	8	4	0	1	3
end	8	4	0	8	4
child					

nonsense$
012345678

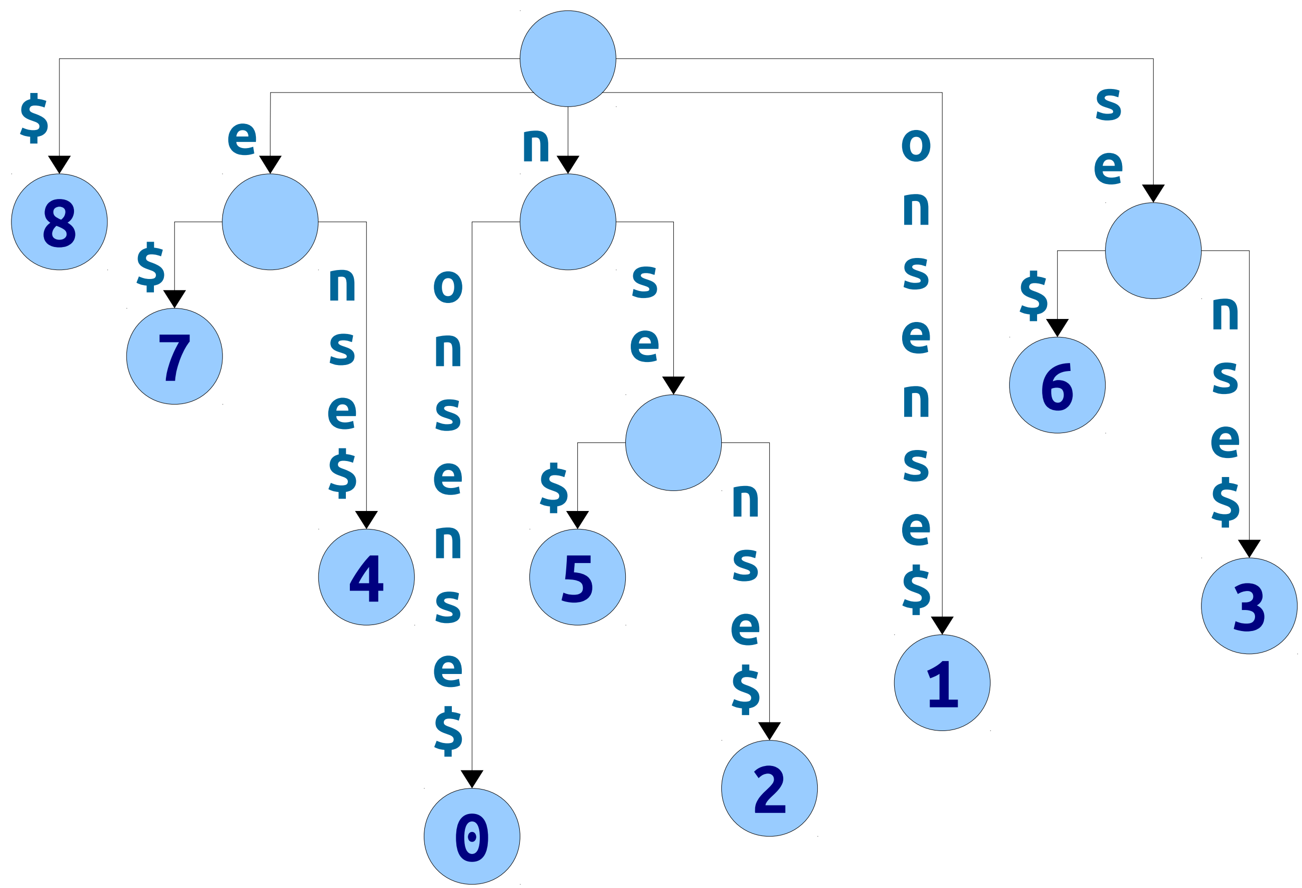
Representing a Suffix Tree

- Space usage required for a suffix tree:
  - O(m) space for all the nodes.
  - O(m) space for a copy of the original string.
  - O(m) space for the edges.
- Total space: O(m).



Constructing a Suffix Tree

- The naive algorithm for building a suffix tree (add one suffix at a time) takes time $\Theta(m^2)$.
- ***Claim:*** With a much more clever approach, this can be done in time $O(m)$.
- ***This is not obvious***. We'll spend a full lecture on this idea later on.



nonsense$
012345678

The Story So Far

- Suffix trees give
  - an $\langle O(\boldsymbol{m}), O(\boldsymbol{n} + \boldsymbol{z})\rangle$ solution to substring searching, and
  - an $O(\boldsymbol{m})$-time solution to longest repeated substring.
- That alone is pretty cool.
- But we can take this even further!

More to Explore

- We’ve barely scratched the surface of suffix trees. They can be used for tons of other problems.
- A sampling:
  - ***Approximate string matching***: Given a fixed text string T and a pattern P, see the closest match to P in T.
  - ***Fast matrix multiplication***: The matrix multiplications needed in computing word embeddings can, amazingly, be optimized using suffix trees.
- This is a rich space to explore for a final project, if that’s something you’d like to do!

Next Time

- ***Suffix Arrays***
  - A space-efficient alternative to suffix trees.
- ***LCP Arrays***
  - Implicitly capturing suffix tree structure.