11/5: Tries

Discord: https://discord.gg/68VpV6

Why Tries?

What are the alternatives for storing a lot of data?

Problem Introduction

There are many applications that need to store a lot of words to perform many different actions

Some examples include:

- Dictionary
- Thesaurus
- Words with Friends
- Anything with autocomplete

Main Problem

Suppose we're trying to create a dictionary app, or really any app that interacts with a list of words

What are some of the data structures for storing words that usually come to mind?

- Lists
- Hash Tables
- Tries? (not usually!)

List Implementation

One approach we can look at is storing all of our words in a list

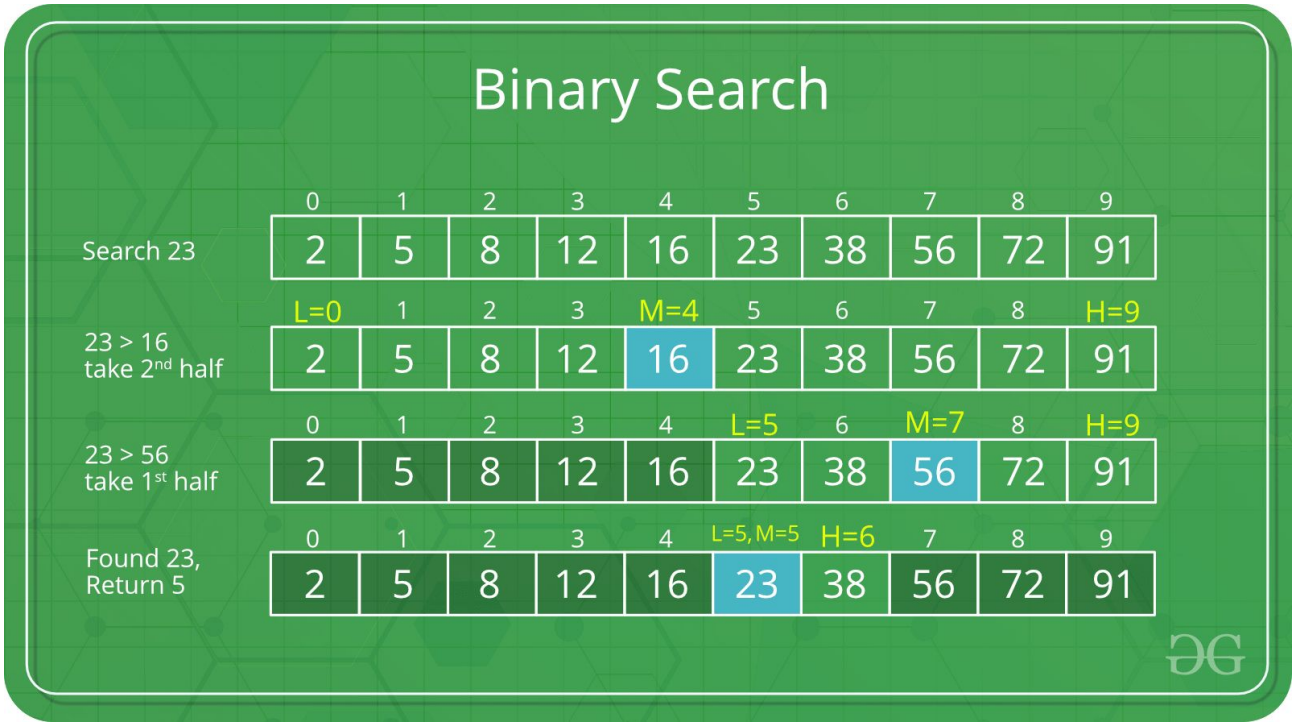
```
["apple", "banana", ..., "watermelon"]
```

Lists are great because its super quick to insert and find elements at a certain index

However, if we wanted to check if a word was in our dictionary, we'd run into some problems

List Implementation: Problem

Even if we assume that we have a sorted list, the best we can do to check if a word exists is binary search, which is $O(\log_2 n)$



Hash Table Implementation

We can also try storing our information in a hash table

```
{ “apple” : 1,
  “banana” : 1,
  ... : ...,
  “Watermelon” : 1}
```

This is great for our lookup time, but if we need to do literally anything else (*i.e. autocomplete*), it’s not the right choice

Is there anything better?

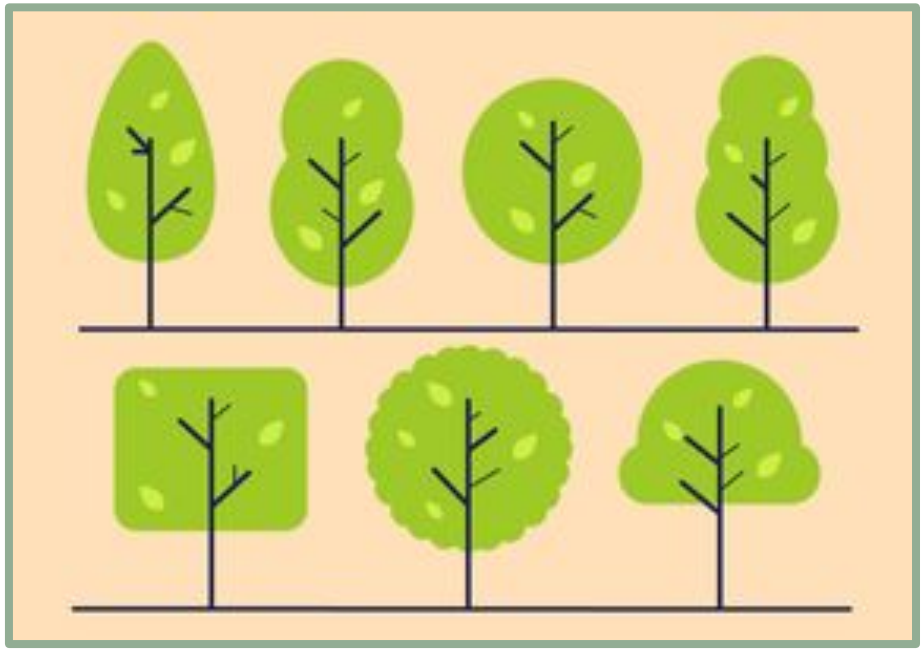
All About Tries

What can a Trie do?

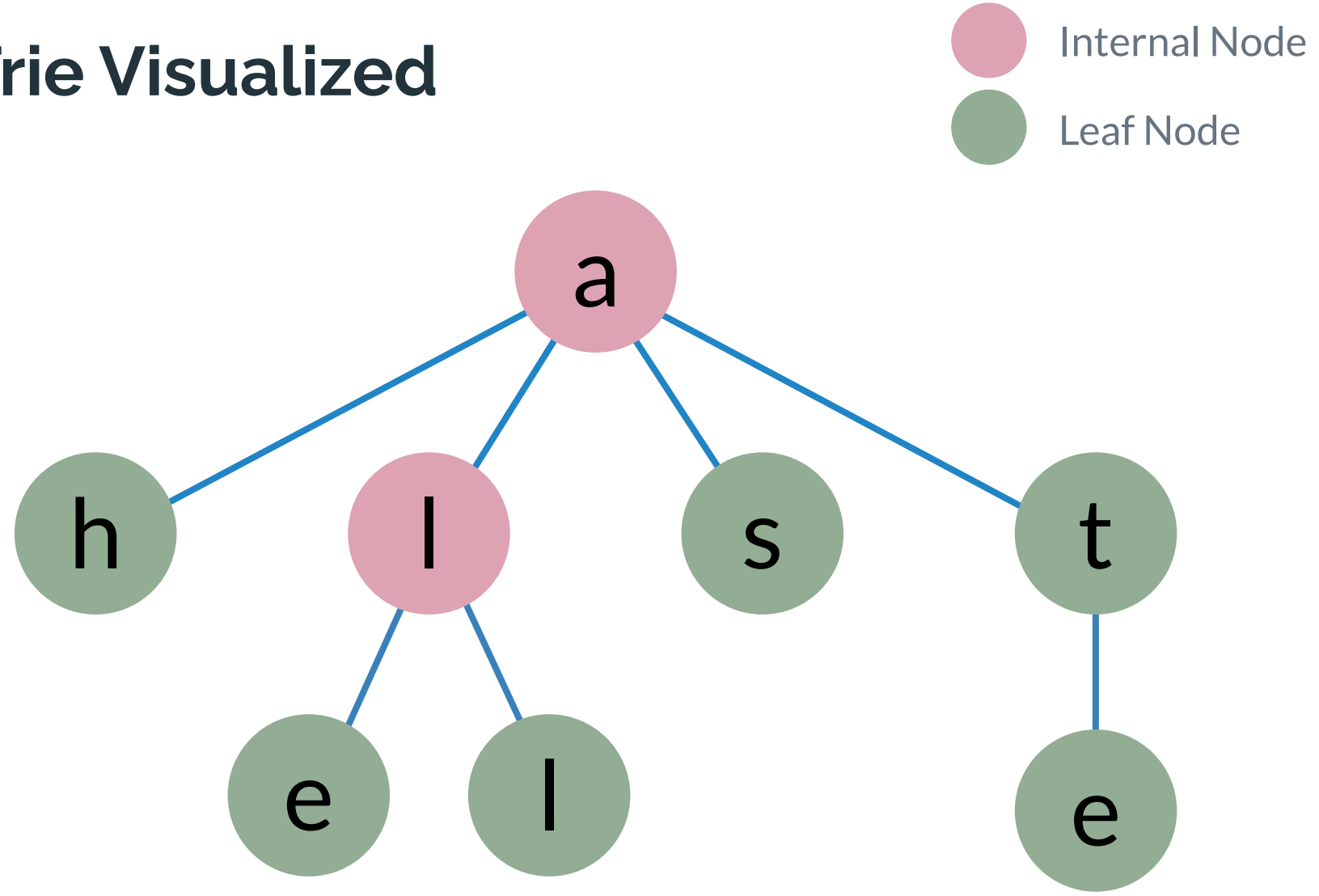
reTrieval Introduction

A trie is an extension of a tree where each node has 26 children, one for each letter of the alphabet

A leaf node in a trie corresponds to being the end of the word



Trie Visualized



Contains: `"ah"`, `"ale"`, `"all"`, `"as"`, `"at"`, `"ate"`

Node Structure

Each node stores two things:

1. A list of 26 children
    a. Each letter corresponds to a child
    b. If there is no child, the array will store None
2. A boolean flag for if the node is a child

```
class TrieNode:
    def __init__(self):
        # give it 26 children, initialized to None
        self.children = np.repeat(None, 26)
        # if the node is a leaf, it is the end of the word
        self.is_leaf = False
```

Trie Insert

To insert a word into our Trie, we follow this general algorithm:

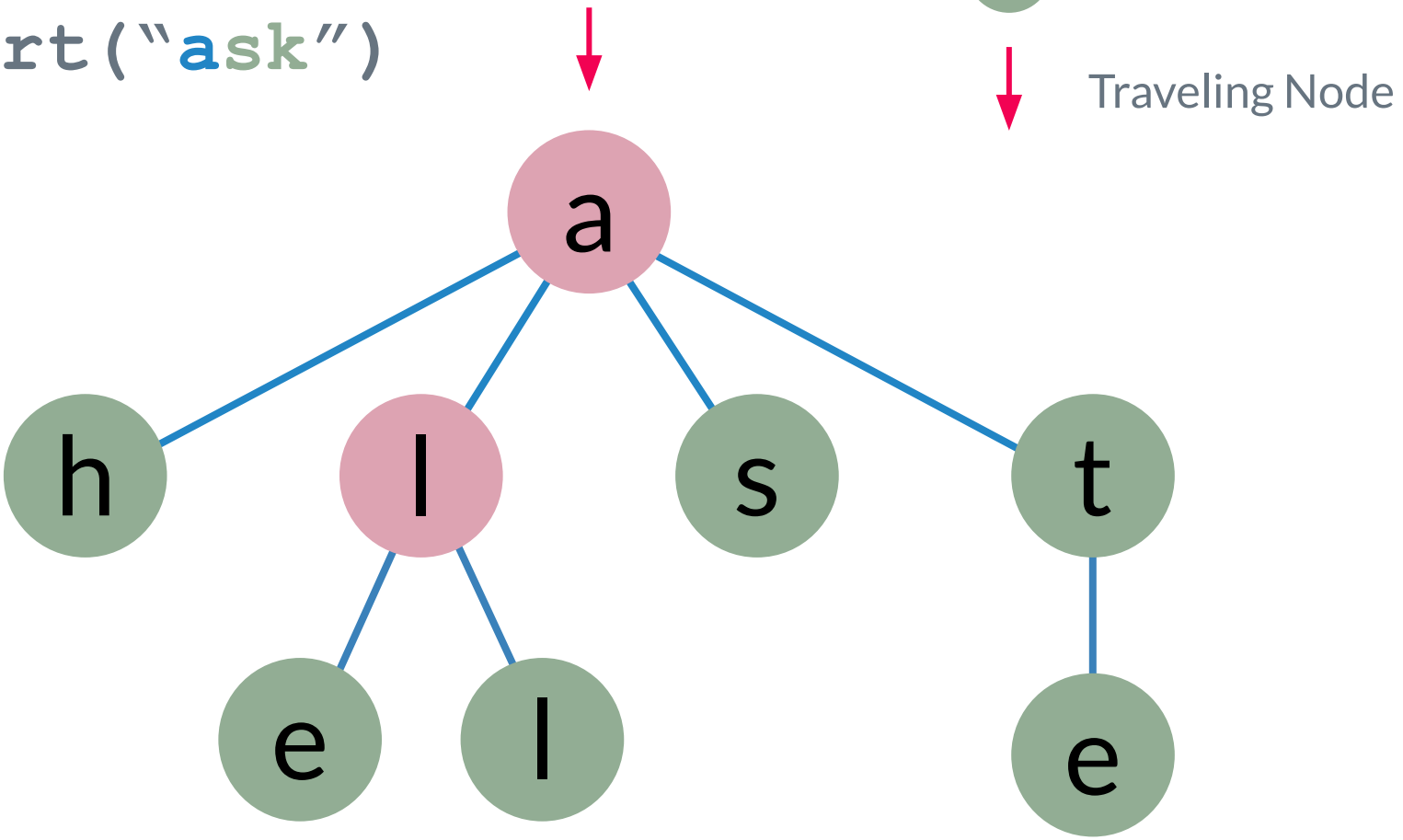
1. Set our “traveling” node to the Trie root

2. While our word is not the empty string:
    a. Get the location of the child, the current letter
    b. If our next child is None, create a new Node
    c. Set our “traveling” node to the next child
    d. Move to the next letter

3. Set is_leaf to True

Trie Insert Visualized:

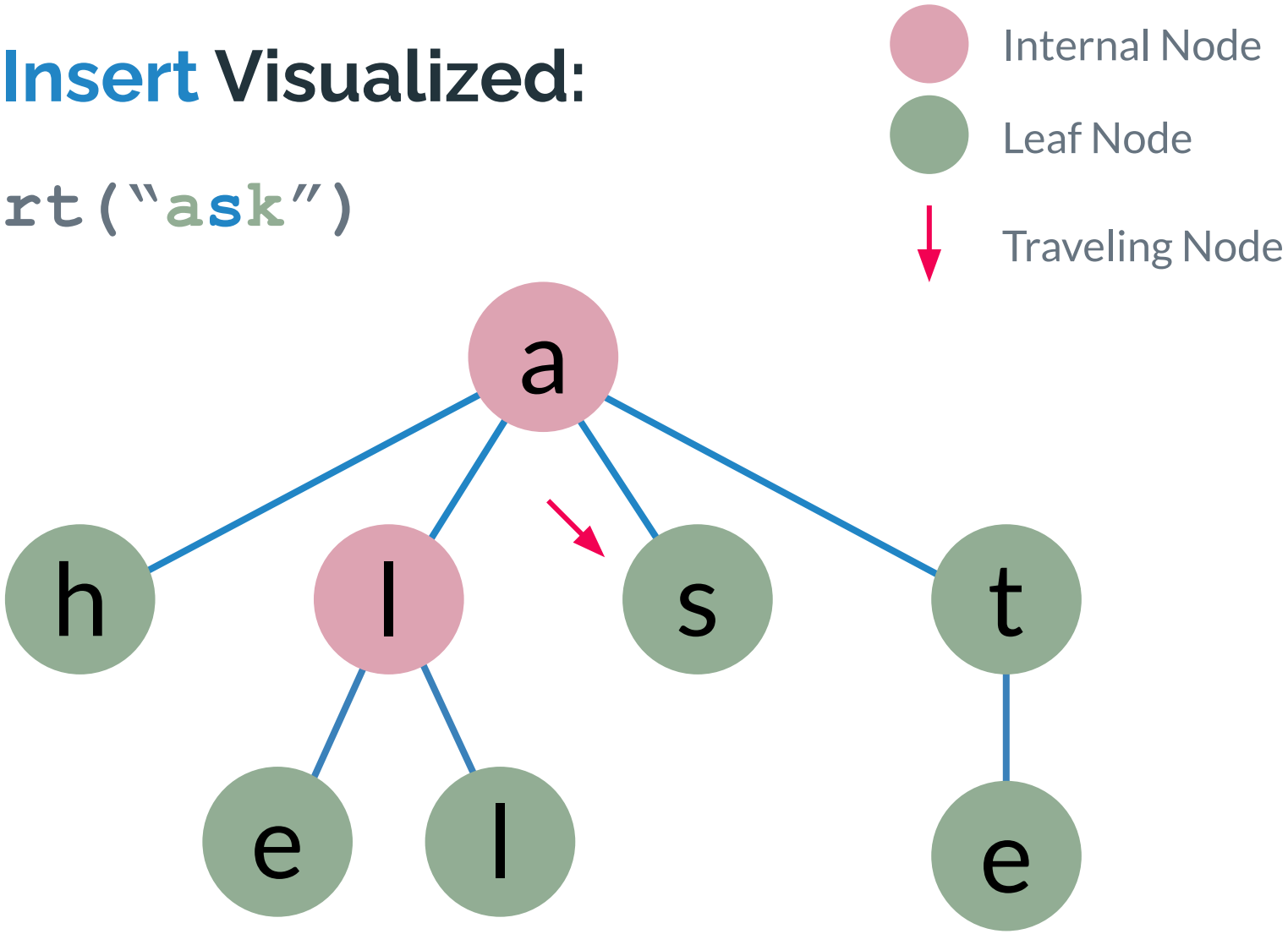
`insert("ask")`



We start at our root node, “a”

Trie Insert Visualized:

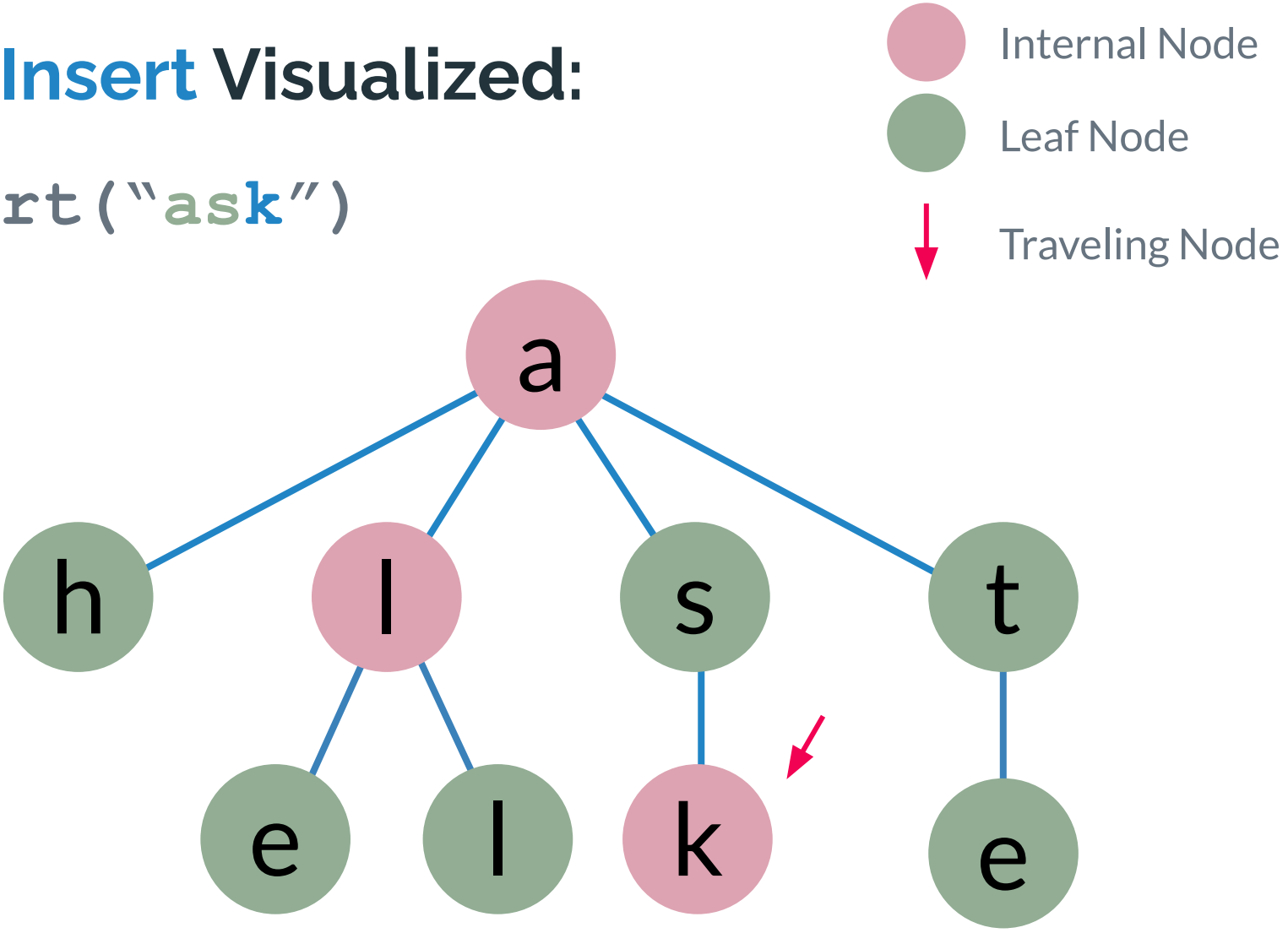
```
insert("ask")
```



We then move onto the second letter, “s”, which already exists

Trie Insert Visualized:

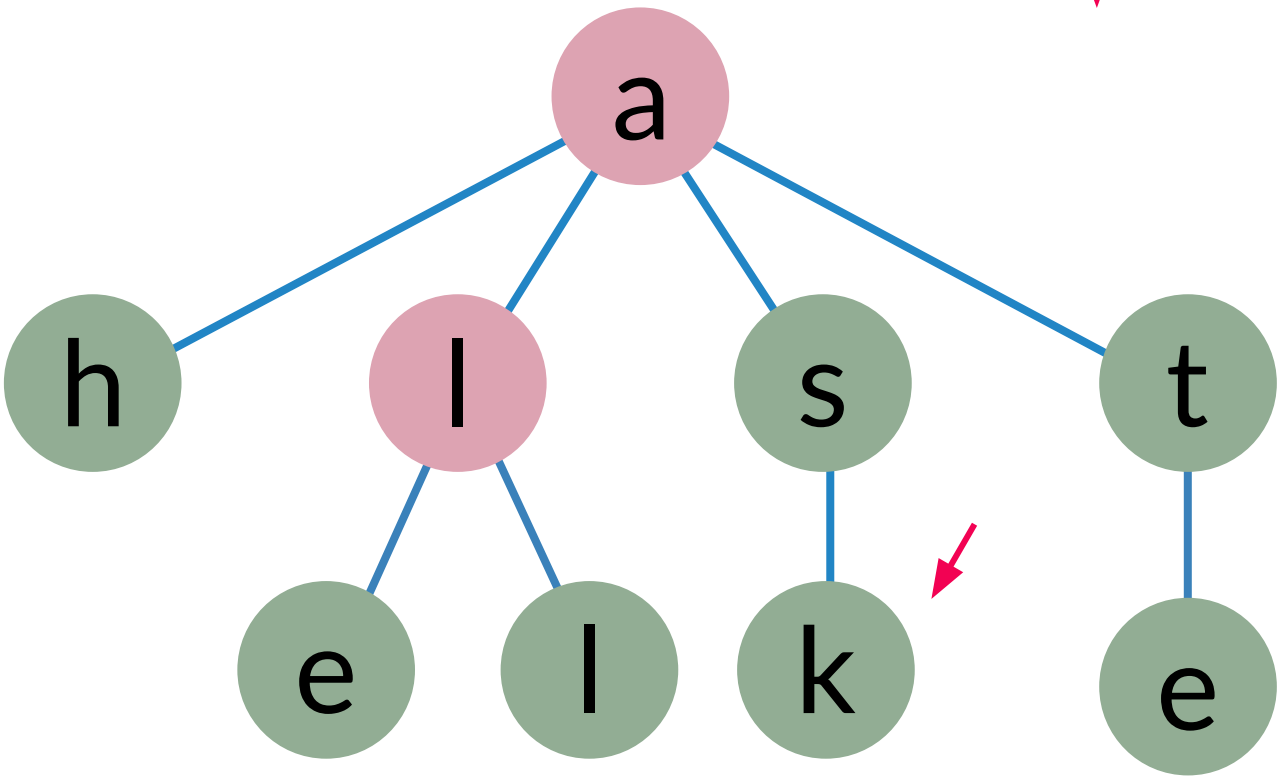
```
insert("ask")
```



Our next letter, “k”, doesn’t exist, so we have to make a new node

Trie Insert Visualized:

```
insert("ask")
```



We're done with our word, so we set the last node to a leaf node

Trie Find

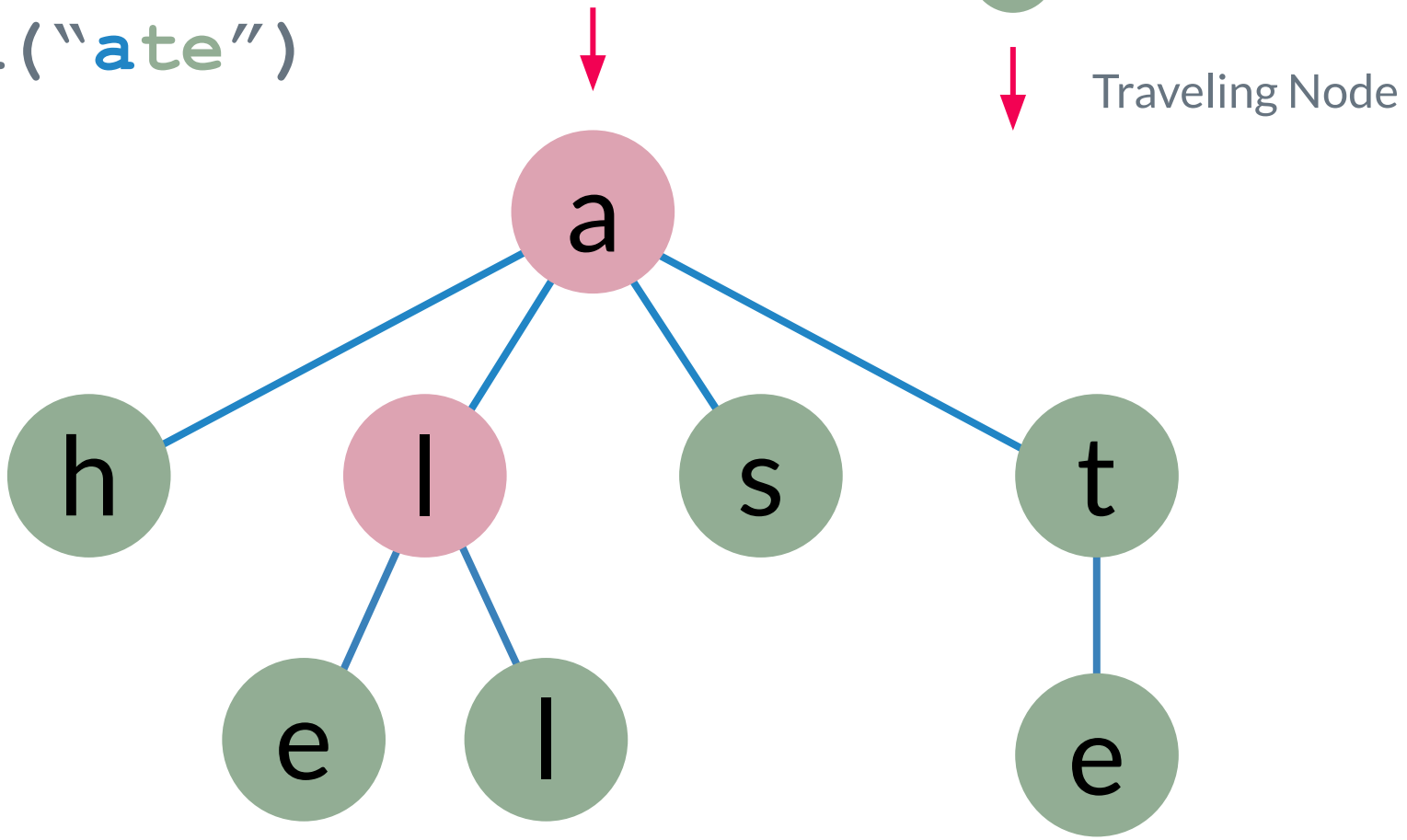
Finding if a word exists in a Trie is essentially the same as inserting

We simply travel down the Trie until we run out of letters. However, we need to check if the node we are on is None. If the node is None, it means the word does not exist in the Trie

When we're done traveling, we only have to check if the ending node is not None, and if it is a leaf

Trie Find Visualized:

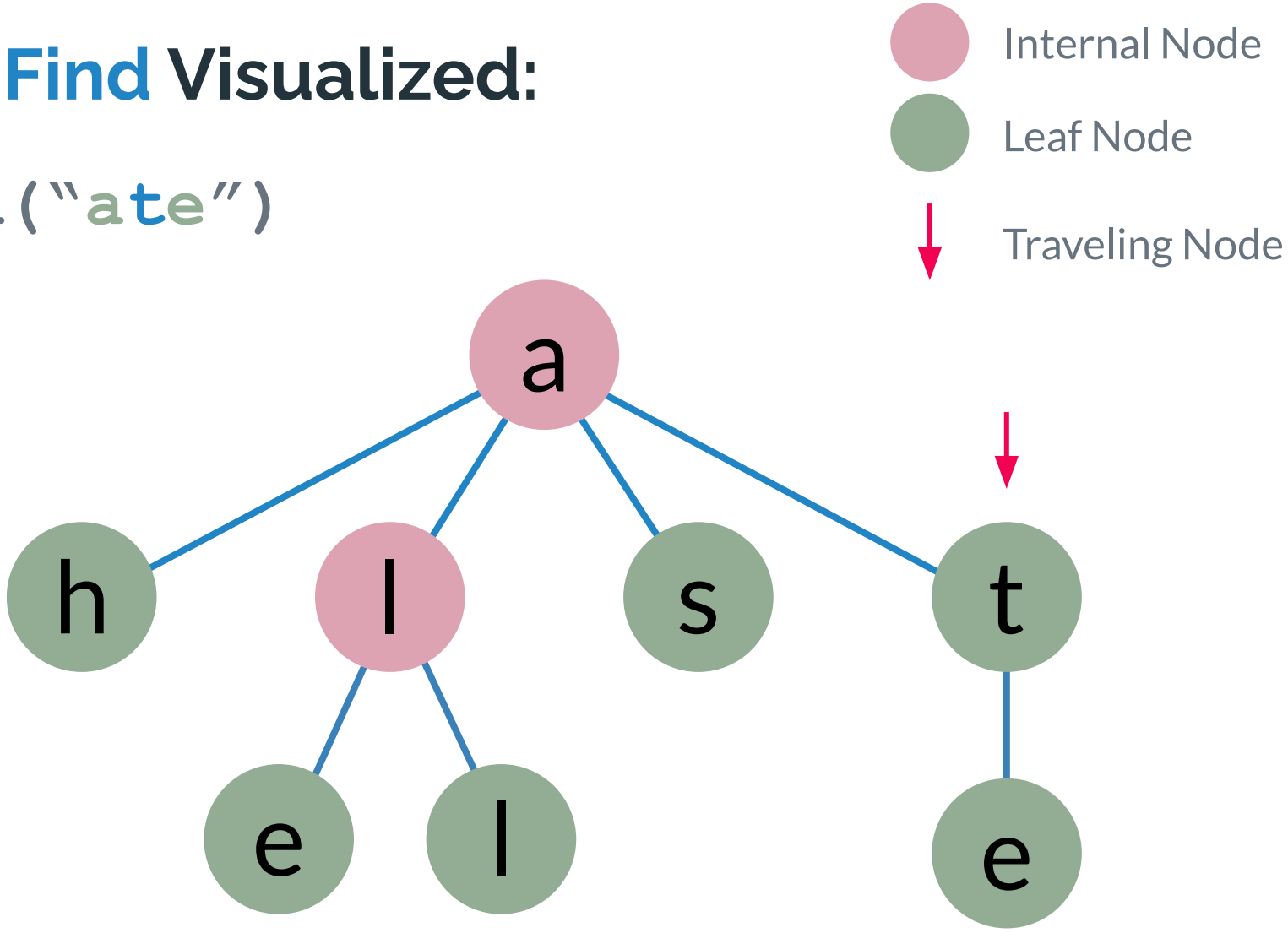
`find("ate")`



We start at our root node, “a”

Trie Find Visualized:

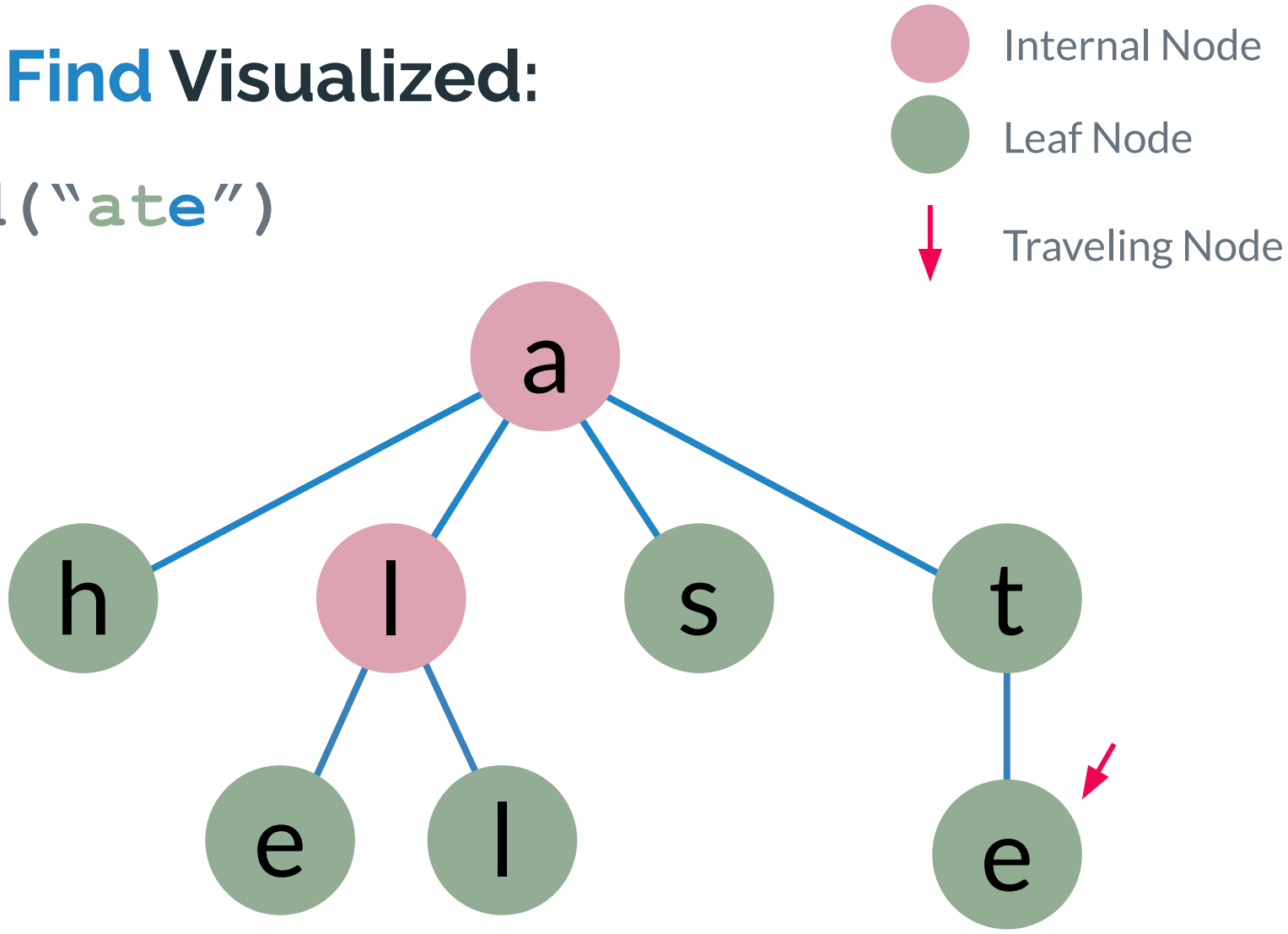
```
find("ate")
```



We then move to our next letter, “t”

Trie Find Visualized:

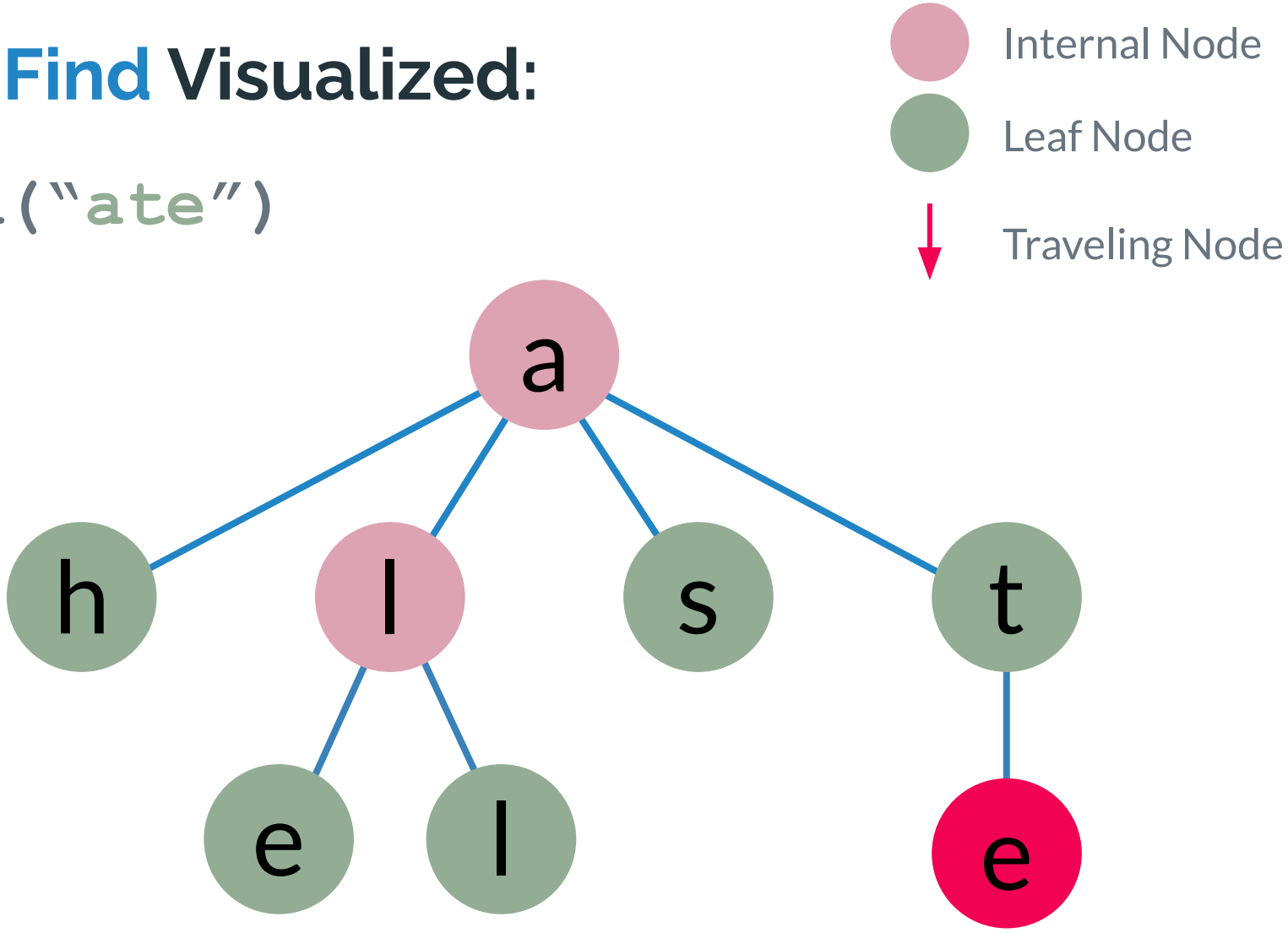
```
find("ate")
```



We move to our final letter, “e”

Trie Find Visualized:

```
find("ate")
```



We return true because “e” is a leaf node and our path is done

Why a Trie?

To reiterate, there are some things that Tries can do that other data structures cannot do efficiently, including:

- Auto-completion
- Word unscrambling
- Alphabetic order printing

We look at some of these things in the notebook

This data structure could be extremely useful for some of your future projects!

Trie Disadvantages

While a Trie can do a lot of cool things, it has **horrible** space complexity

It's important to keep this in mind if you want to use it in a large scale project!

Tasks to Complete

1) Work on the notebook (**trie.zip)** in the google drive folder https://drive.google.com/drive/folders/1mm-55M-VivpktAde0InbPMGFf760hjVZ?usp=sharing

Try to work on it collaboratively! You might meet some people you could do a project with in the future

As always, let us know if you need any help!


SIG
NLL