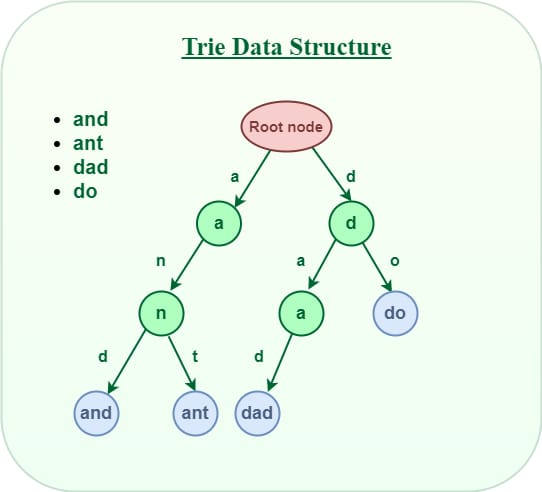
**Trie (Prefix Tree)**

A tree-like data structure used to store strings efficiently by sharing common prefixes. It consists of nodes connected by edges. Each node represents a character or a part of a string. The root node, the starting point of the Trie, represents an empty string. Each edge emanating from a node signifies a specific character. The path from the root to a node represents the prefix of a string stored in the Trie.

Operations like insertion, search and deletion are performed in ***O(n)***, where ***n*** is the length of the word.



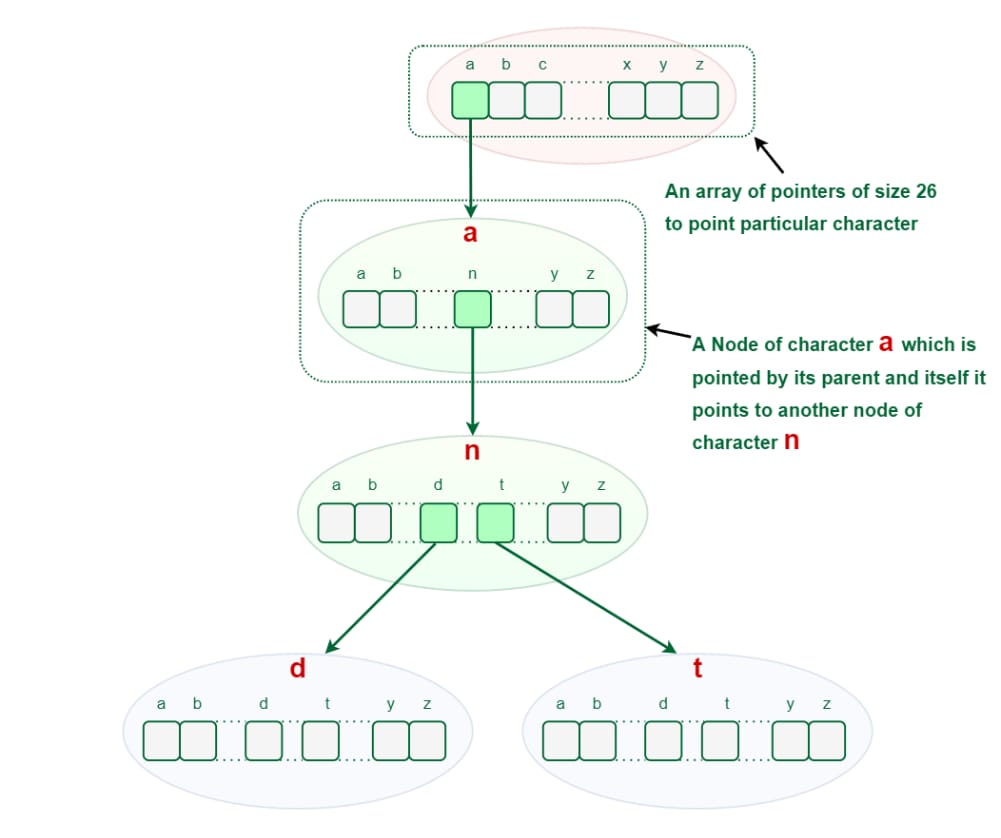
**Full ADT Specification of the Trie Abstract Data Type (ADT)**

**Operations:**

1. **Insert (Key): Adds a word into the trie.**

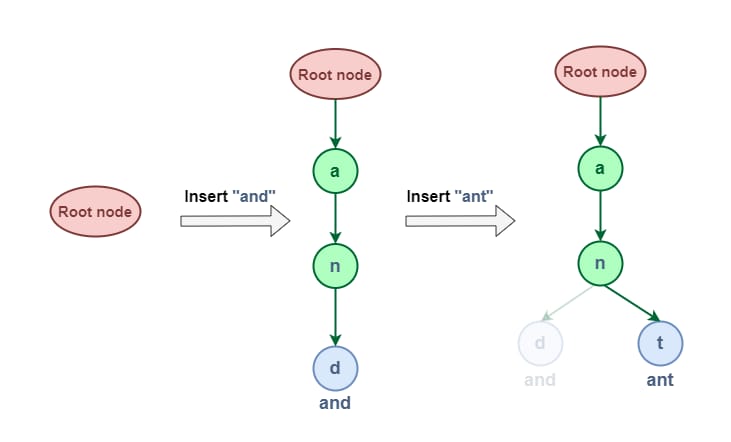
Inserting “and” in Trie data structure:

* Start at the root node: The root node has no character associated with it and its wordEnd value is 0, indicating no complete word ends at this point.
* First character “a”: Calculate the index using ‘a’ – ‘a’ = 0. Check if the child[0] is null. Since it is, create a new TrieNode with the character “a“, wordEnd set to 0, and an empty array of pointers. Move to this new node.
* Second character “n”: Calculate the index using ‘n’ – ‘a’ = 13. Check if child[13] is null. It is, so create a new TrieNode with the character “n“, wordEnd set to 0, and an empty array of pointers. Move to this new node.
* Third character “d”: Calculate the index using ‘d’ – ‘a’ = 3. Check if child[3] is null. It is, so create a new TrieNode with the character “d“, wordEnd set to 1 (indicating the word “and” ends here).



Inserting “ant” in Trie data structure:

* Start at the root node: Root node doesn’t contain any data but it keeps track of every first character of every string that has been inserted.
* First character “a”: Calculate the index using ‘a’ – ‘a’ = 0. Check if the child[0] is null. We already have the “a” node created from the previous insertion. So, move to the existing “a” node.
* First character “n”: Calculate the index using ‘n’ – ‘a’ = 13. Check if child[13] is null. It’s not, so move to the existing “n” node.
* Second character “t”: Calculate the index using ‘t’ – ‘a’ = 19. Check if child[19] is null. It is, so create a new TrieNode with the character “t“, wordEnd set to 1 (indicating the word “ant” ends here).



**Time Complexity:** *O(number of words \* maxLengthOfWord)*

**Auxiliary Space:** *O(number of words \* maxLengthOfWord)*

1. **Search (Key): Checks if a word exists in the trie.**

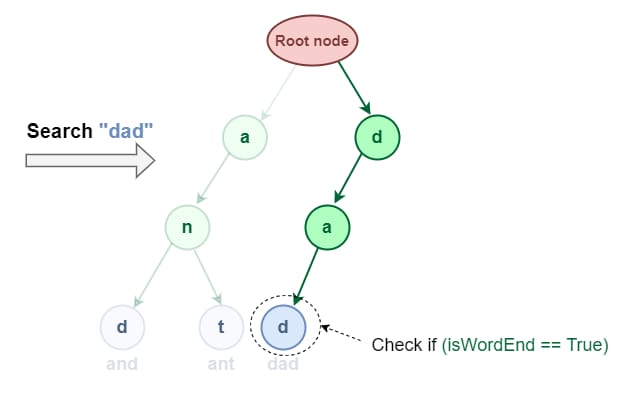
Searching for a key in Trie data structure is similar to its insert operation. However, It only compares the characters and moves down. The search can terminate due to the end of a string or lack of key in the trie.

Steps by step approach for searching in Trie Data structure:

* Start at the root node. This is the starting point for all searches within the Trie.
* Traverse the Trie based on the characters of the word you are searching for. For each character, follow the corresponding branch in the Trie. If the branch doesn’t exist, the word is not present in the Trie.
* If you reach the end of the word and the wordEnd flag is set to 1, the word has been found.
* If you reach the end of the word and the wordEnd flag is 0, the word is not present in the Trie, even though it shares a prefix with an existing word.

Searching word “dad” in the trie

* We start at the root node.
* We follow the branch corresponding to the character ‘d’.
* We follow the branch corresponding to the character a’.
* We follow the branch corresponding to the character ‘d’.
* We reach the end of the word and wordEnd flag is 1. This means that “dad” is present in the Trie.

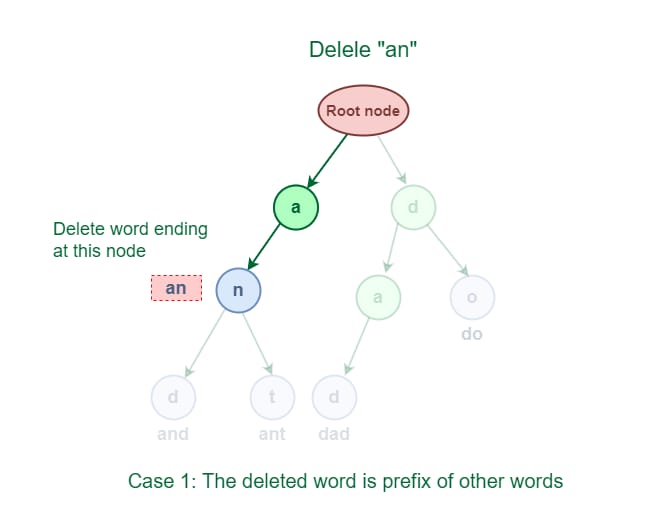


1. **Delete (Key): Removes a word from the trie if it exists.**

This operation is used to delete strings from the Trie data structure. There are three cases when deleting a word from Trie.

The deleted word is a prefix of other words in Trie.

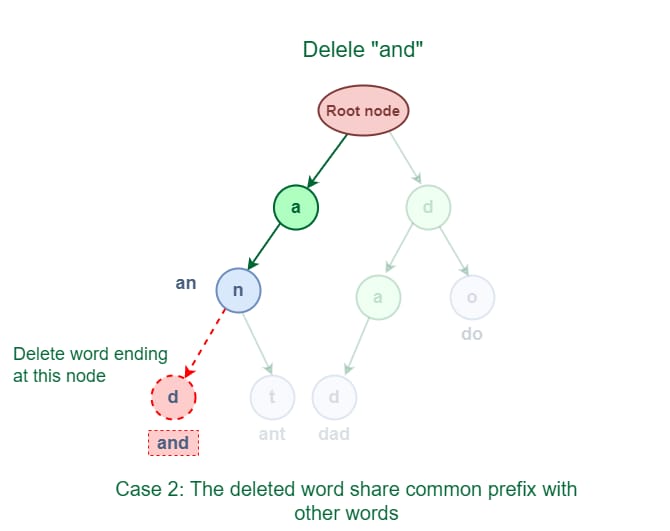
As shown in the following figure, the deleted word “an” share a complete prefix with another word “and” and “ant“.



An easy solution to perform a delete operation for this case is to just decrement the wordCount by 1 at the ending node of the word.

The deleted word shares a common prefix with other words in Trie.

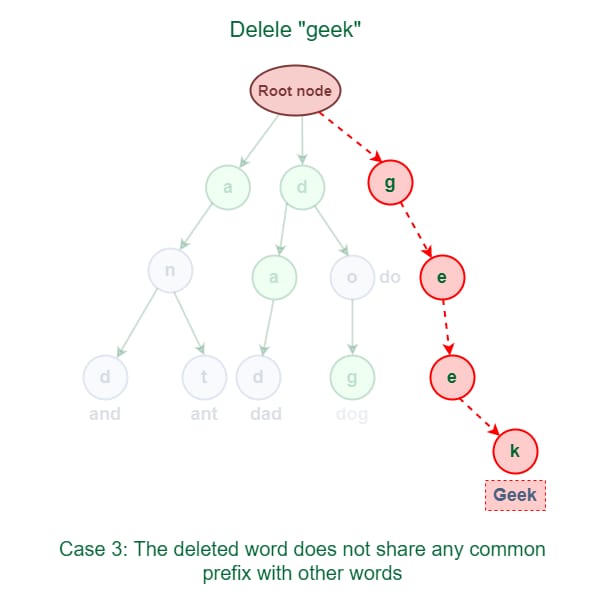
As shown in the following figure, the deleted word “and” has some common prefixes with other words ‘ant’. They share the prefix ‘an’.



The solution for this case is to delete all the nodes starting from the end of the prefix to the last character of the given word.

The deleted word does not share any common prefix with other words in Trie.

As shown in the following figure, the word “geek” does not share any common prefix with any other words.



The solution for this case is just to delete all the nodes.

**Properties:**

* Each Trie has an empty root node, with links (or references) to other nodes
* Each node of a Trie represents a string and each edge represents a character.
* Every node consists of hashmaps or an array of pointers, with each index representing a character and a flag to indicate if any string ends at the current node.
* It can contain any number of characters including alphabets, numbers and special characters.
* Each path from the root to any node represents a word or string.
* Case-sensitive or case-insensitive depending on the implementation.

**Application Domain That Suits Trie Domain: Autocomplete Systems**

Tries are widely used in search engines or text editors for implementing autocompletion. Example: As a user types a query, the system suggests possible completions based on stored words.

**Implementation of the Application for the Autocomplete Domain**

**Python Implementation for Autocomplete:**

class TrieNode:

def \_\_init\_\_(self):

self.children = {}

self.is\_end\_of\_word = False

class Trie:

def \_\_init\_\_(self):

self.root = TrieNode()

def insert(self, word):

node = self.root

for char in word:

if char not in node.children:

node.children[char] = TrieNode()

node = node.children[char]

node.is\_end\_of\_word = True

def search(self, word):

node = self.root

for char in word:

if char not in node.children:

return False

node = node.children[char]

return node.is\_end\_of\_word

def starts\_with(self, prefix):

node = self.root

for char in prefix:

if char not in node.children:

return []

node = node.children[char]

return self.\_find\_words\_from(node, prefix)

def \_find\_words\_from(self, node, prefix):

results = []

if node.is\_end\_of\_word:

results.append(prefix)

for char, child in node.children.items():

results.extend(self.\_find\_words\_from(child, prefix + char))

return results

# Example Usage

trie = Trie()

trie.insert("apple")

trie.insert("app")

trie.insert("application")

print(trie.starts\_with("app")) # Output: ['apple', 'app', 'application']