Project Phase2 Deliverable2: Proof of Concept Implementation

Pavani Chavali

Department of Computer Science, University of The Cumberlands

MSCS-532-B01: Algorithms and Data Structures

Professor. Brandon Bass

Date: 11/16/2025

The implementation of the code is basically split into two files trie.py and demo.py:

Trie.py contains the Trie Node ad Trie class definitions representing the core data structure.

Demo.py is a simple script to demonstrate the key functionality of insertion and auto completion.

The Proof of concept focuses on implementing the core functionality of the design: the Trie data structure and the auto-completion feature. The more complex spell-correction is implemented in the Next steps.

1. Partial Implementation Overview

The Proof of Concept will consist of implementing the two primary classes mentioned in design phase of the documentation: TrieNode and Trie.

TrieNode Class:

```
class TrieNode:
  def __init__(self):
    self.children = {} #this acts as hashmap that maps a character to a correspinding child trienode object
    self.is_end_of_word = False #this becomes True if the path from root to the node represents a complete word
```

This class will be implemented as described in the Phase1 report.

self.children: This is class iterable to store words and a Python dictionary hash map that maps a character (like 'a') to another TrieNode object.

self.is_end_of_word: A Boolean flag, initialized to False, that will be set to True if the node represents the end of a complete word.

Trie Class:

```
class Trie:
  def __init__(self):
    self.root = TrieNode()

  def insert(self,word):
    #Implements the insertion algorithm to add a word to the Trie

    word = word.lower()
    current_node = self.root

    for char in word:
      #checking if the character exists as a key
      if char not in current_node.children:
        #if not found create a new Trie node
        current_node.children[char] = TrieNode()
      #update current node to point to the children
      current_node = current_node.children[char]
    current_node.is_end_of_word= True

  def get_autocomplete_suggestions(self,prefix):
    prefix = prefix.lower()
    suggestions = []

    current_node = self.root
    for char in prefix :
      if char not in current_node.children:
        #if prefix path is not present return empty list
        return []
      current_node = current_node.children[char]
    self.find_words_from_node(current_node, prefix, suggestions)
    return suggestions

  def find_word_from_node(self,node, current_prefix, suggestions):
    #helper method to perform recursive DFS
```

```
    if node.is_end_of_word:
        suggestions.append(current_prefix)

    for char,child_node in node.children.items():
        self.find_word_from_node(child_node, current_prefix + char, suggestions)
```

This class will encapsulate the Trie's logic.

self.root: It will be initialized with a single TrieNode instance.

Implemented Method: insert(word)

* This method will implement the core insertion algorithm as you defined it.
* It will iterate, character by character, creating new TrieNode objects if the path does not exist.
* It will set the is_end_of_word flag to True on the final node.

Implemented Method: get_autocomplete_suggestions(prefix)

* This method will implement the two-phase process from your design.

Phase 1: Traverse to the prefix node.

It will iterate through the prefix, and if any character is not found, it will return an empty list.

Phase 2: Collect words.

Once at the prefix node, it will call a private helper method (find_words_from_node) to perform a Depth-First Search (DFS) on the entire subtree below that node, collecting all words.

Excluded Functionality for PoC:

* The get_spelling_corrections(word, max_distance) method will not be implemented in this PoC. As it requires a separate Levenshtein distance algorithm, and its exclusion keeps the PoC focused on the core data structure's primary function.

2. Demonstration and Testing

A simple Python script demo.py will be created to demonstrate the PoC's functionality.

Script Logic:

1. Initialize a Trie object.
2. Create a small, sample word list (e.g., ["car", "cargo", "cart", "cat", "apple", "app"]).
3. Insert each word into the trie using the insert() method.
4. Run a series of test cases using `get_autocomplete_suggestions()` and print the results.

Test Cases:

Standard Case: Basic prefix search

Input: "ca"

Output: [‘car', 'cargo', 'cart', 'cat']

Partial Match: Shows that both a full word and it prefixes are found

Input: "app"

Output: ['apple', 'app']

Full Word Match: Shows a prefix that is also a complete word

Input: "apple"

Output: ['apple']

Edge Case 1: Tests a prefix that is not in the trie.

No Match: "z"

Output: []

Edge Case 2: Tests the case of an empty prefix (should return all words).

Empty String: ""

Ouput: ['car', 'cargo', 'cart', 'cat', 'apple', 'app']

3. Implementation Challenges and Solutions

Challenge 1: Recursive DFS for Suggestions

Problem: The get_autocomplete_suggestions method needs to perform a complex DFS from the prefix node.

Workaround: A private helper method,find_words_from_node(self, node, current_prefix)`, will be created. This recursive function will:

1. Check if the node.is_end_of_word is True and, if so, add current_prefix to a results list.
2. Iterate through the node.children and recursively call itself for each child, appending the new character to current_prefix.

Challenge 2: Case Insensitivity

Problem: Your report identifies case insensitivity as a practical challenge. A user might type "app" but expect "Apple" (if it were in the dictionary).

Workaround : To keep the PoC simple, all inputs will be normalized. The insert() method will convert all words to lowercase before insertion. The get_autocomplete_suggestions() method will likewise convert the input prefix to lowercase before searching.

4. Next Steps

This PoC lays the foundation for the full application. The next steps will be:

1. Implement Spell Correction:Implement the get_spelling_corrections method. This involves researching and integrating a Levenshtein distance algorithm to generate candidate words and then using the existing trie to validate them.

2. Optimize for Memory: The current implementation is a naive Trie. As noted in your report's analysis, this has significant memory overhead. The next phase will focus on optimizing this by implementing a Compressed Trie like Radix Tree to reduce node count and pointer overhead.

3. Scale and Test: The PoC will be tested with a small word list. The next step is to load a large-scale dictionary and benchmark the insert and get_autocomplete_suggestions performance to prepare for the end analysis.

5. Documentation

The final report will include these critical code snippets with detailed explanations:

TrieNode Class: The full, simple class definition.

Trie.insert() Method:The full method, showing the character-by-character traversal and node creation logic.

Trie.get_autocomplete_suggestions() and its _find_words_from_node()helper:This pair of methods is the most complex and demonstrates the core logic of the PoC.

Attaching demo code and output:

```
from bfjr import Trie

def run_demo():

    # 1. Initialize the Trie data structure
    trie = Trie()

    # 2. Inserting a sample word list to populate the dictionary
    word_list = ["car", "cargo", "cart", "cat", "apple", "app", "application"]
    print(f"Populating Trie with: {word_list}")
    for word in word_list:
        trie.insert(word)
    print("Insertion complete\n")

    test_prefixes = ["ca", "app","z"]

    for prefix in test_prefixes:
        print(f"Test Case: Auto-complete for prefix '{prefix}'")
        suggestions = trie.get_autocomplete_suggestions(prefix)

        if suggestions:
            print(f"Found {len(suggestions)} suggestions: {suggestions}")
        else:
            print("No suggestions found.")
        print("next input \n")
```

```
run_demo()
```

output:

practice > demo.py > ...

```python
from bfjr import Trie

def run_demo():

    # 1. Initialize the Trie data structure
    trie = Trie()

    # 2. Inserting a sample word list to populate the dictionary
    word_list = ["car", "cargo", "cart", "cat", "apple", "app", "application"]
    print(f"Populating Trie with: {word_list}")
    for word in word_list:
        trie.insert(word)
    print("Insertion complete\n")

    test_prefixes = ["ca", "app","z"]

    for prefix in test_prefixes:
        print(f"Test Case: Auto-complete for prefix '{prefix}'")
        suggestions = trie.get_autocomplete_suggestions(prefix)

        if suggestions:
            print(f"Found {len(suggestions)} suggestions: {suggestions}")
        else:
            print("No suggestions found.")
        print("next input \n")

run_demo()
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS PVS-STUDIO Code

```
/usr/bin/python3 "/Users/pavanichavali/Desktop/practice/practice/demo.py"
pavanichavali@Pavanis-MacBook-Pro practice % /usr/bin/python3 "/Users/pavanichavali/Desktop/practice/practice/demo.py"
Populating Trie with: ['car', 'cargo', 'cart', 'cat', 'apple', 'app', 'application']
Insertion complete

Test Case: Auto-complete for prefix 'ca'
Found 4 suggestions: ['car', 'cargo', 'cart', 'cat']
next input

Test Case: Auto-complete for prefix 'app'
Found 3 suggestions: ['app', 'apple', 'application']
next input

Test Case: Auto-complete for prefix 'z'
No suggestions found.
next input
```

References:

https://www.geeksforgeeks.org/dsa/trie-insert-and-search/

https://www.codecademy.com/article/trie-data-structure-complete-guide-to-prefix-trees

Githublink:

https://github.com/PavaniChavali135/AlgoandDatastructures/tree/main/week4/phase2