Write a Program to Implement Search, insert, and Remove in Trie.

Mr. Akash Yadav
Assistant Professor
Artificial Intelligence & Data Science

Lesson Plan

Subject/Course	**Competitive Coding**
Lesson Title	**Write a Program to Implement Search, insert, and Remove in Trie.**

Lesson Objectives
To understand the concept and structure of Trie (Prefix Tree).
To implement the basic Insert, Search, and Remove operations in a Trie.
To efficiently store and retrieve strings with common prefixes.
To handle deletion carefully so shared prefixes among words are not lost.
To analyze the time and space complexity of Trie operations.

Problem Statement:

Write a program to implement a Trie (prefix tree) that supports:

- insert(String key) — insert a word into the trie
- search(String key) — return true if the exact word exists
- remove(String key) — delete a word from the trie (if it exists)

Concept

- A **Trie** is a tree-like data structure used to store a dynamic set of strings where keys are usually strings.
- Fast prefix-based operations: insert, search, and delete in time proportional to the length of the word O(L) where L is word length.
- Useful for auto-complete, spell-checkers, dictionary implementations, and IP routing (prefix matching).

Algorithm/Logic

Insert

1. Start at root.
2. For each character ch in the word:

- compute index (e.g., ch - 'a').
- if children[index] is null, create new node.
- move to children[index].

1. After last char, mark isEndOfWord = true.

Time: O(L) per insertion. Space: O(L) new nodes in worst case.

Algorithm/Logic

Search

1. Start at root.

    1. For each character ch in the word:

    - follow children[index].

    - if at any step child is null → return false.

    1. After last char, return node.isEndOfWord.

Time: O(L) per search.

Algorithm/Logic

Delete

1. Deleting a word requires careful handling to avoid removing nodes used by other words.
2. Use recursive helper that:
3. Traverses characters to the end.
4. Unmarks isEndOfWord at end.
5. On unwind, delete a child node if:
6. It has no children, and
7. isEndOfWord is false.
8. Return boolean to parent indicating whether child can be removed.
9. Time: O(L) per deletion.

Code Implementation

```java
class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEndOfWord = false;
}

public class Practical20_Trie {
    private TrieNode root;

    public Practical20_Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    public void insert(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (node.children[index] == null)
                node.children[index] = new TrieNode();
            node = node.children[index];
        }
        node.isEndOfWord = true;
    }
```

```java
    // Search a word in the Trie
    public boolean search(String word) {
        TrieNode node = root;
        for (char ch : word.toCharArray()) {
            int index = ch - 'a';
            if (node.children[index] == null)
                return false;
            node = node.children[index];
        }
        return node.isEndOfWord;
    }

    // Delete a word from the Trie
    public boolean delete(String word) {
        return deleteHelper(root, word, 0);
    }
```

```java
    private boolean deleteHelper(TrieNode node, String word, int depth) {
        if (node == null)
            return false;

        if (depth == word.length()) {
            if (!node.isEndOfWord)
                return false;
            node.isEndOfWord = false;

            // If no children, node can be deleted
            return isEmpty(node);
        }

        int index = word.charAt(depth) - 'a';
        if (deleteHelper(node.children[index], word, depth + 1)) {
            node.children[index] = null;
            return !node.isEndOfWord && isEmpty(node);
        }
        return false;
    }
```

```java
    private boolean isEmpty(TrieNode node) {
        for (TrieNode child : node.children)
            if (child != null)
                return false;
        return true;
    }

    // Main method
    public static void main(String[] args) {
        Practical20_Trie trie = new Practical20_Trie();
        trie.insert("cat");
        trie.insert("car");
        trie.insert("dog");

        System.out.println("Search 'car': " + trie.search("car"));
        System.out.println("Search 'cap': " + trie.search("cap"));

        trie.delete("car");
        System.out.println("After deleting 'car', search 'car': " + trie.search("car"));
        System.out.println("Search 'cat': " + trie.search("cat"));
    }
}
```

Output

```
Search 'car': true
Search 'cap': false
After deleting 'car', search 'car': false
Search 'cat': true
```

Time & Space Complexity

Time Complexity: O(L) per operation

Space Complexity: O(26 × L) (for children references)

Summary

- Efficient method to search pair adding up to target
- Uses HashMap for instant lookup
- Suitable for large inputs
- One-pass solution

Summary

✅ Advantage:

- Eliminates nested loops
- Guaranteed faster performance

📌 Use Cases:

- E-Commerce cart matching
- Financial security validation
- Data search operations

Practice Questions:

1️⃣ Implement Trie (Prefix Tree) — LeetCode #208

🔗 https://leetcode.com/problems/implement-trie-prefix-tree/

Concept: Build a Trie supporting insert(), search(), and startsWith() operations.

Why Practice: Directly matches this practical — helps master the basic structure and traversal logic of Trie.

Practice Questions:

2 Add and Search Word – Data Structure Design — LeetCode #211

🔗
https://leetcode.com/problems/add-and-search-word-data-structure-design/

Concept: Extend Trie to support wildcard searches using recursion (. can match any letter).

Why Practice: Strengthens understanding of Trie traversal and recursive pattern searching.

Practice Questions:

3️⃣ Replace Words — LeetCode #648

🔗 https://leetcode.com/problems/replace-words/

Concept: Use Trie to replace words in a sentence with the shortest root from a given dictionary.

Why Practice: Demonstrates how Trie can be applied in real-world tasks like text simplification and dictionary lookups.

Thanks