

Implement Trie

Prefix tree from scratch — 26-pointer node design ■

● MEDIUM

■ Hook

Google's autocomplete. Phone contact search. Spell checkers. IP routing tables. All of these use a Trie (pronounced 'try') internally. A Trie is a tree where **each path from root to a node spells a prefix**. It answers 'does this word exist?' and 'does any word start with this prefix?' in $O(L)$ time — where L is the word length, completely independent of how many words are stored.

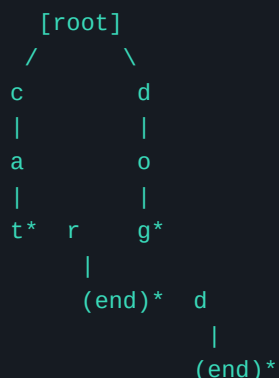
■ Core Intuition — Sharing Prefixes

Instead of storing each word independently, words that share a prefix share nodes. 'cat', 'car', 'card' all share the path $c \rightarrow a$. Each node has up to 26 children (one per letter) and an **isEnd** flag marking complete words.

■ Why $O(L)$ search regardless of n words?

In a HashMap you hash the full string. In a Trie you traverse one character at a time. Both are $O(L)$, but the Trie additionally supports prefix queries — HashMap can't tell you 'how many words start with pre' efficiently.

■ Visual — Insert: cat, car, card, dog



* = isEnd flag set to true

'cat', 'car', 'card', 'dog' all inserted.

Search 'car': $c \rightarrow a \rightarrow r \rightarrow$ found (isEnd=true) ✓

StartsWith 'ca': $c \rightarrow a \rightarrow$ node exists ✓

Search 'ca': $c \rightarrow a \rightarrow$ isEnd=false ✗ (not a full word)

■ ■ How It Works

1

TrieNode structure

26 children[] array (or HashMap). boolean isEnd flag.

2

Insert(word)

Walk from root, creating nodes as needed. Set isEnd=true at last char.

3

Search(word)

Walk from root following each char. Return isEnd at last node.

4

StartsWith(prefix)

Walk from root following each char. Return true if path exists (don't check isEnd).

■ Code

```
class Trie {
    private TrieNode root;
    static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEnd = false;
    }
    public Trie() { root = new TrieNode(); }
    public void insert(String word) {
        TrieNode node = root;
        for (char c : word.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null)
                node.children[idx] = new TrieNode();
            node = node.children[idx];
        }
        node.isEnd = true;
    }
    public boolean search(String word) {
        TrieNode node = traverse(word);
        return node != null && node.isEnd;
    }
    public boolean startsWith(String prefix) {
        return traverse(prefix) != null;
    }
    private TrieNode traverse(String s) {
        TrieNode node = root;
        for (char c : s.toCharArray()) {
            int idx = c - 'a';
            if (node.children[idx] == null) return null;
            node = node.children[idx];
        }
        return node;
    }
}
```

■ TIME

$O(L)$ per op

■ SPACE

$O(N \times L)$ total

L = word length, N = number of words. Each operation visits at most L nodes.

■ ■ Using HashMap vs TrieNode[26]

children[26] is faster (direct index) and simpler. HashMap is better for large alphabets (Unicode, DNA). For lowercase English always use the array.

■ Practice

#	Problem	Diff	Pattern
1	Implement Trie (LC 208)	Medium	Trie
2	Design Add and Search Words (LC 211)	Medium	Trie + DFS
3	Replace Words (LC 648)	Medium	Trie prefix
4	Index Pairs of a String (LC 1065)	Easy	Trie
5	Longest Word in Dictionary (LC 720)	Medium	Trie

- ✓ TrieNode: TrieNode[26] children + boolean isEnd
- ✓ Insert: create path, set isEnd=true at end
- ✓ Search: follow path, return isEnd at last node
- ✓ StartsWith: follow path, return whether it exists
- ✓ $O(L)$ per operation, $O(N \times L)$ total space

■ MENTAL MODEL — Remember This Forever

A Trie is like a library filing system where books are shelved letter by letter. To find 'cat': go to shelf 'c', then sub-shelf 'a', then sub-sub-shelf 't'. Words sharing prefixes share shelves — saving space and making prefix queries instant.