**Trie (Prefix Tree)**

**1) Definition**

A **Trie** is a special tree data structure used to store strings efficiently for **prefix based searching**.

**2) Node Structure**

Each node typically contains:

* **Children**
  + Up to 26 links for lowercase English letters
  + Or a map/dictionary for general character sets
* **End of Word Flag**
  + Boolean marker indicating a complete word ends at this node

**3) Key Properties**

* Designed for **prefix operations**
* Works like a dictionary structure
* Insert and search depend on word length, not number of words

**Time Complexity:**

* Insert: **O(L)**
* Search: **O(L)**
* Prefix search: **O(L)**  
  (L = length of the word)

**4) Uses / Advantages**

* Fast prefix based search
* Efficient handling of large dictionaries
* Ideal for string lookup tasks

**Common Applications:**

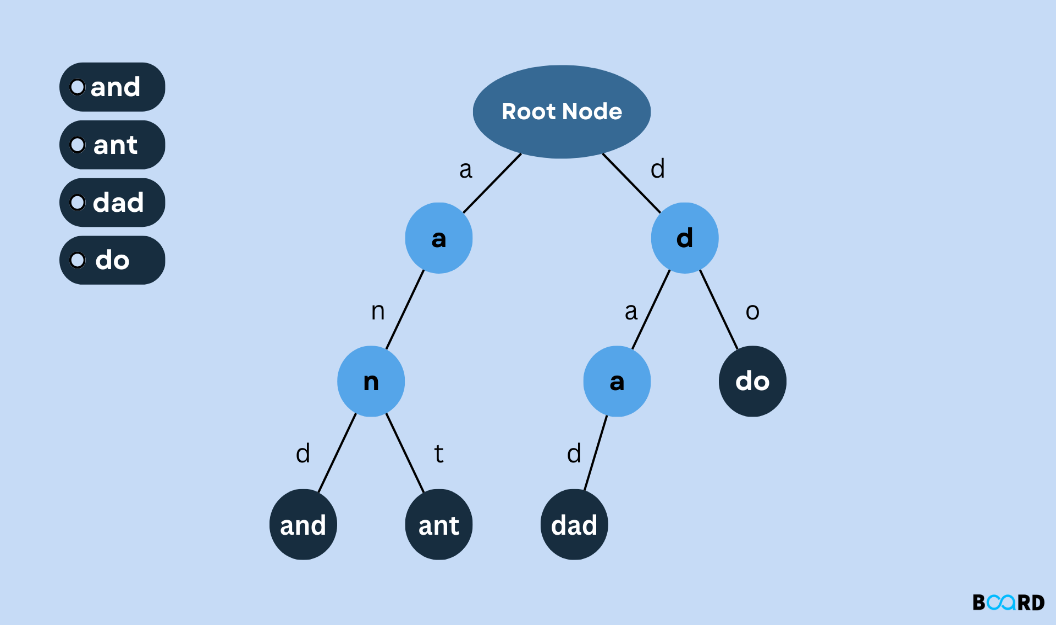
* Auto complete systems
* Spell checkers
* Word filtering
* Dictionary lookup
* Search suggestions

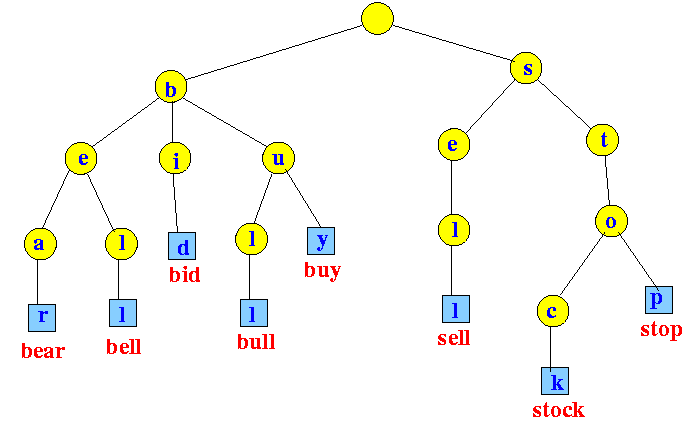
**5) Types of Tries**

* Basic Trie
* Compressed Trie (Radix Tree)
* Suffix Trie
* Word Search Trie
* Aho–Corasick Automaton (multi pattern matching)

**Types of Trie Structures**

**1) 🟢 Basic Trie (Prefix Tree)**





**Definition:**  
A tree structure used to store strings so that common prefixes are shared.

**Key Features**

* Each edge represents a character
* Path from root → node forms a prefix
* End of word flag marks complete words

**Operations**

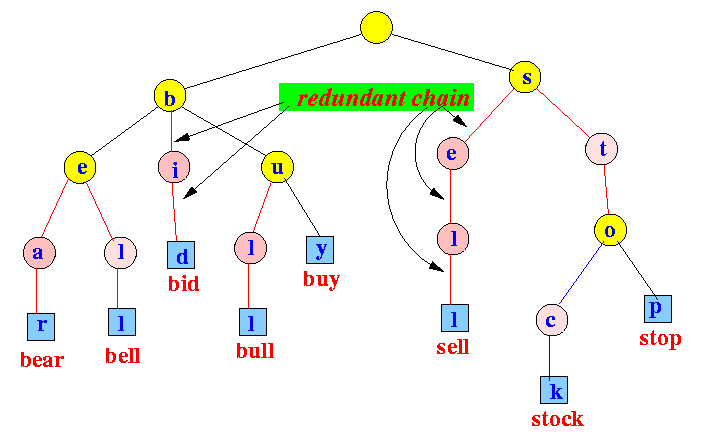
* Insert word
* Search full word
* Check prefix

**Time Complexity:** O(L) per operation (L = word length)

**Use Cases**

* Autocomplete
* Dictionary lookup
* Spell checking
* Prefix queries

**2) 🟡 Compressed Trie (Radix Tree)**



**Definition:**  
An optimized Trie where chains of single child nodes are merged into one edge labeled with a substring.

**Key Features**

* Reduces memory usage
* Faster traversal due to fewer nodes
* Also called Patricia Trie

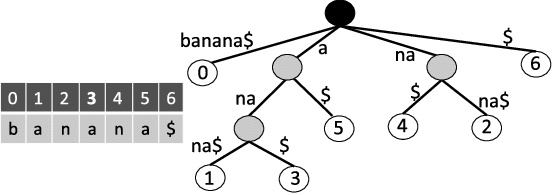
**Advantages**

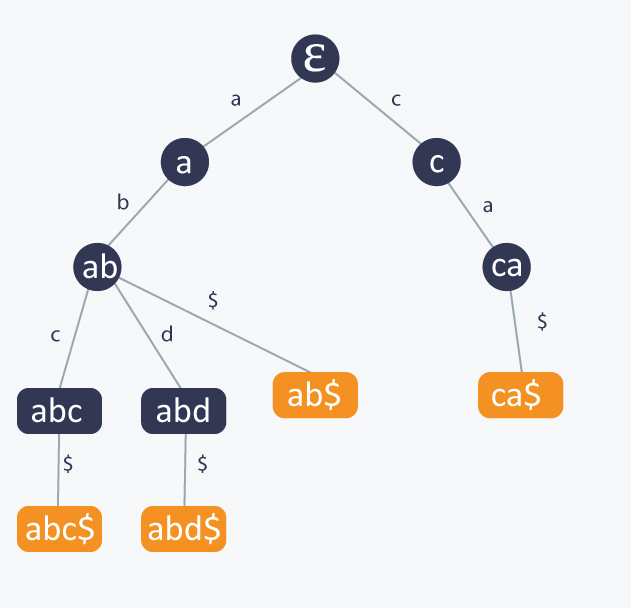
* Efficient for large datasets
* Avoids long single child paths

**Use Cases**

* IP routing tables
* Databases
* Search engines
* Memory constrained systems

**3) 🔵 Suffix Trie**





**Definition:**  
A Trie that stores all suffixes of a given string.

For string "banana" → stores:  
banana, anana, nana, ana, na, a

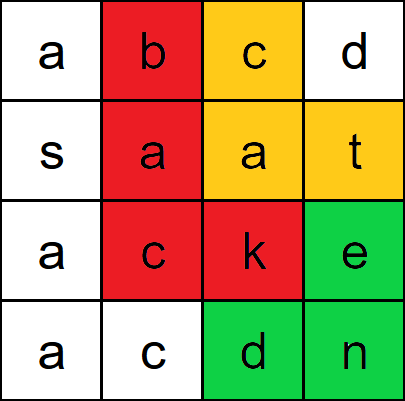
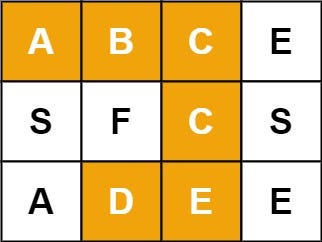
**Key Features**

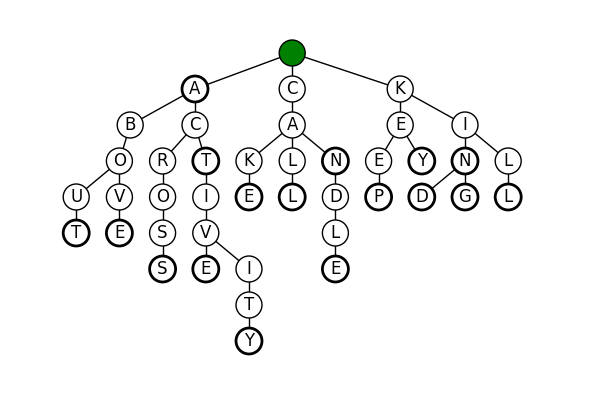
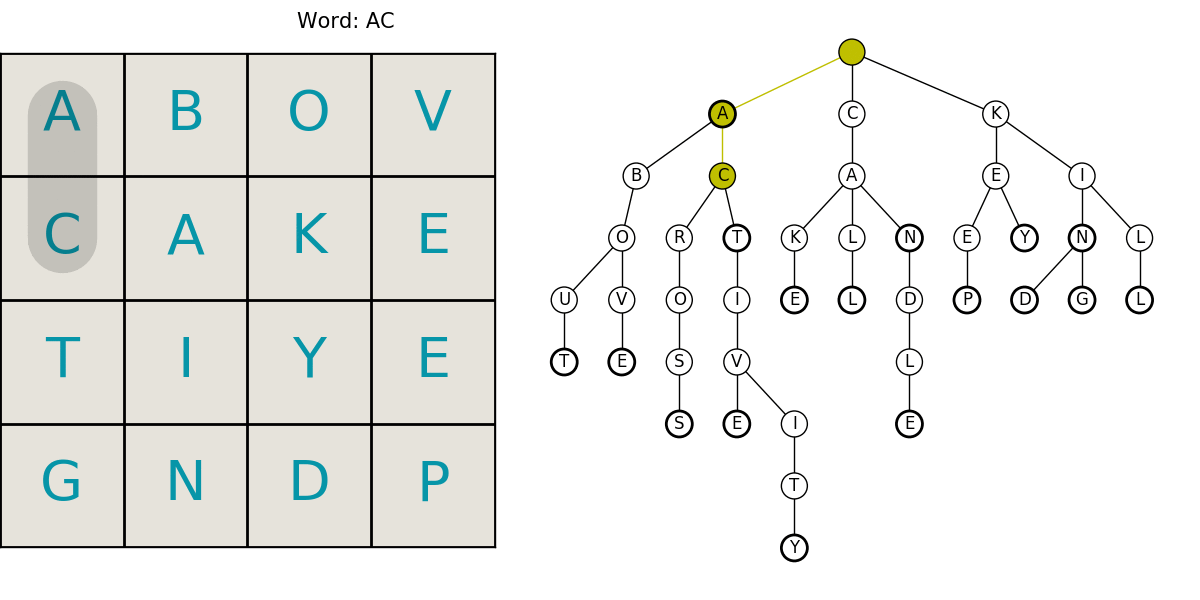
* Supports substring queries efficiently
* Large memory usage (can be O(n²))

**Use Cases**

* Pattern matching
* Longest repeated substring
* Bioinformatics (DNA analysis)
* Text processing

**4) 🟣 Word Search Trie**



**Definition:**  
A Trie built from a dictionary to search words efficiently in a grid or matrix.

**How It Works**

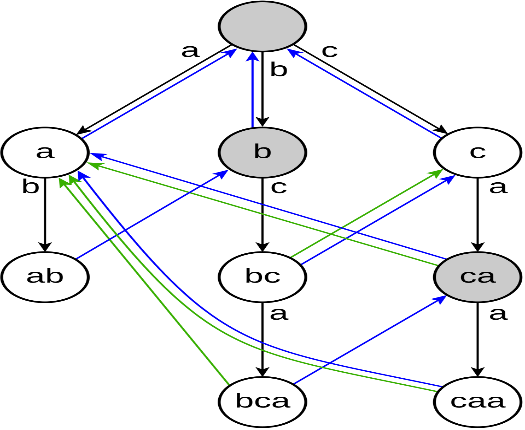
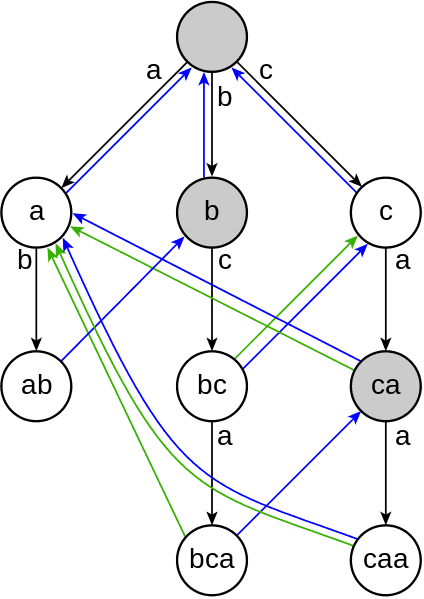
* Insert all dictionary words into Trie
* Perform DFS/BFS on grid
* Use Trie to prune invalid paths

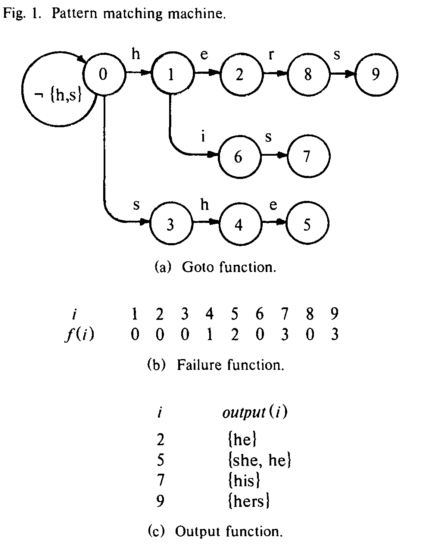
**Key Benefit**  
Avoids checking each word independently.

**Use Cases**

* Word Search puzzles
* Boggle games
* Matrix word problems

**5) 🔴 Aho–Corasick Automaton (Multi Pattern Matching)**



**Definition:**  
An advanced structure built on Trie that enables simultaneous searching of multiple patterns in a text.

**Key Components**

* Trie of patterns
* Failure links (like KMP fallback)
* Output links for matches

**Time Complexity**  
O(text length + number of matches)

**Advantages**

* Searches many patterns at once
* Extremely efficient for large text

**Use Cases**

* Spam filtering
* Intrusion detection systems
* DNA sequence matching
* Search engines
* Text editors

**🏆 Summary Comparison**

| **Type** | **Memory** | **Speed** | **Best For** |
| --- | --- | --- | --- |
| Basic Trie | Medium | Fast | Prefix search |
| Compressed Trie | Low | Faster | Large datasets |
| Suffix Trie | Very High | Fast queries | Substring problems |
| Word Search Trie | Medium | Efficient | Grid word search |
| Aho–Corasick | Medium | Very Fast | Multi pattern matching |

**6) Benefits**

* Eliminates repeated prefix storage
* Extremely fast prefix operations
* Widely used in competitive programming
* Efficient for dictionary based queries
* Helps solve complex string problems easily

**7) How to Recognize Trie Problems**

Look for problems involving:

* Prefix matching
* Suggesting completions
* Words starting with given pattern
* Dictionary lookup
* Searching efficiently in word lists

**8) Typical Applications**

* Auto complete systems
* Word dictionaries
* Spell checkers
* Prefix based filtering
* Word search in grids/matrices
* IP routing tables
* Search engine suggestions

**09) Steps to Solve Using Trie**

1. **Build the Trie** from given words
2. **Insert all dictionary words**
3. Perform required operation:
   * Full word search
   * Prefix search
   * DFS/BFS traversal (for suggestions or grid problems)
4. Return results or filtered matches

**Trie Problems — Easy → Hard (LeetCode)**

**Easy Level**

These focus on basic prefix operations and simple Trie usage.

* **208 — Implement Trie (Prefix Tree)** (Must Do)
* **14 — Longest Common Prefix** *(Trie optional)*
* **720 — Longest Word in Dictionary**
* **389 — Find the Difference** *(character counting, Trie optional concept)*
* **953 — Verifying an Alien Dictionary** *(prefix ordering idea)*

**Medium Level**

Require combining Trie with hashing, sorting, or DFS.

* **211 — Design Add and Search Words Data Structure**
* **648 — Replace Words**
* **677 — Map Sum Pairs**
* **386 — Lexicographical Numbers** *(Trie conceptually)*
* **421 — Maximum XOR of Two Numbers in an Array** *(Bitwise Trie)*
* **1707 — Maximum XOR With an Element From Array**
* **1268 — Search Suggestions System**
* **692 — Top K Frequent Words** *(Trie + heap optional)*

**Hard Level**

Advanced problems combining Trie with DFS, backtracking, or complex logic.

* **212 — Word Search II** (Classic Hard)
* **642 — Design Search Autocomplete System**
* **745 — Prefix and Suffix Search**
* **425 — Word Squares**
* **336 — Palindrome Pairs**
* **472 — Concatenated Words**
* **1032 — Stream of Characters**
* **140 — Word Break II** *(Trie + DP optional)*

**Recommended Learning Order**

If preparing for product-based companies:

1. 208 → Build foundation
2. 211 → Wildcard search
3. 648 → Prefix replacement
4. 421 → Bitwise Trie
5. 1268 → Auto complete
6. 212 → Grid + Trie
7. 745 / 642 → Advanced design