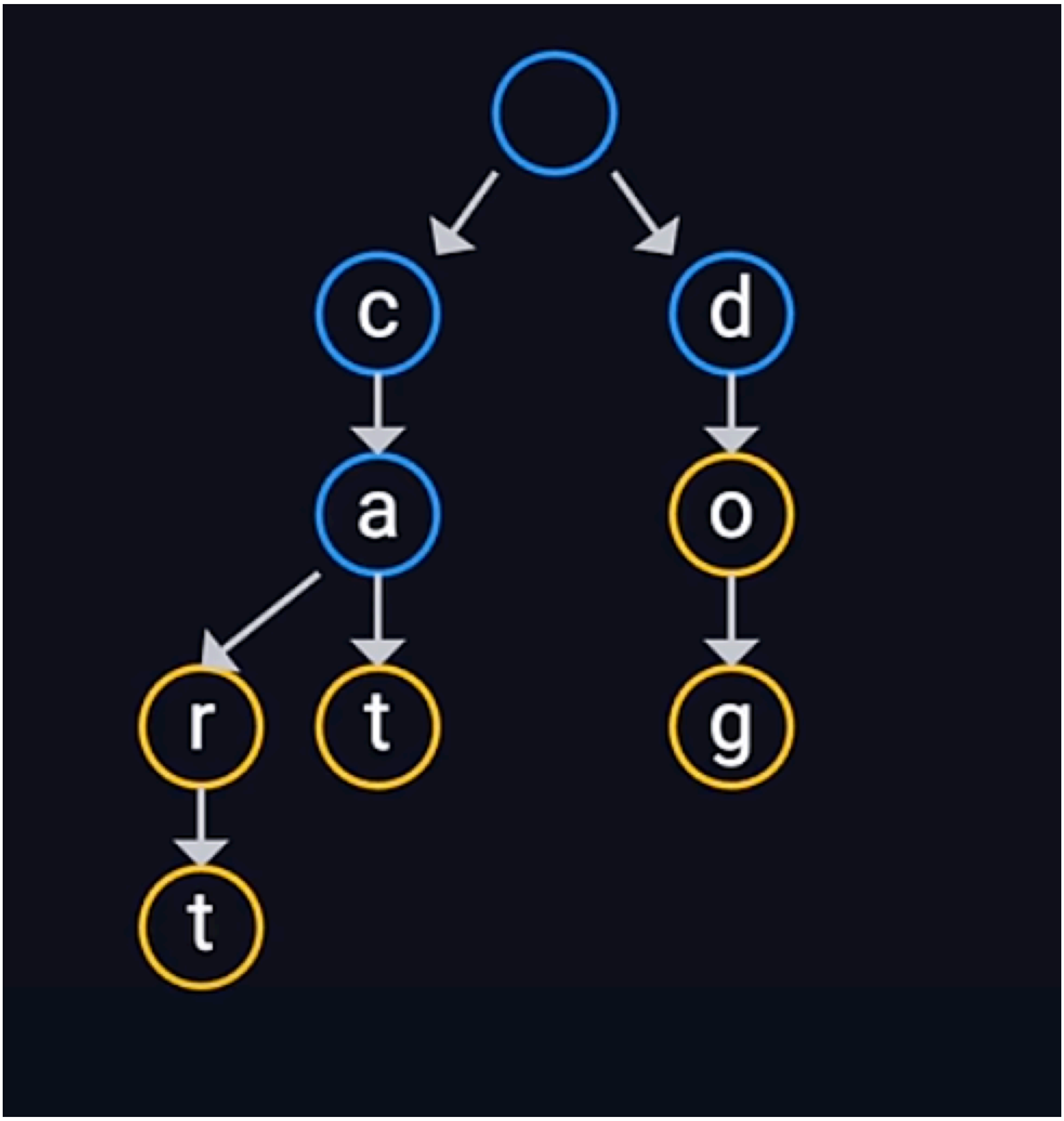
Day 20 Trie

ITSRUNTYM

What is a Trie?

A **Trie** , also known as a **Prefix Tree** or **Digital Tree**, is a **tree-like data structure** used to store a **dynamic set of strings**, where each node represents a **character** of a string.

Key Properties:

- Nodes store **characters**.
- Paths from root to leaf represent **words**.
- It supports **fast insertion, search, and prefix matching**.
- It's especially powerful when dealing with **strings with shared prefixes**.
- 

Real-Life Use Cases

- **Autocomplete**
- **Spell checkers**
- **IP routing (longest prefix match)**

Basic Trie Operations

Operation	**Purpose**	**Time Complexity**
insert(word)	Add a word to the trie	O(L)
search(word)	Check if a word exists	O(L)
startsWith(prefix)	Check if any word starts with prefix	O(L)

Here, L = Length of the word or prefix

Time & Space Complexity

Operation	**Time Complexity**	**Space Complexity**
Insert	O(L)	O(26 × L × N)
Search	O(L)	-
Prefix Search	O(L)	-

Where:

- L = Length of the word/prefix
- N = Number of words
- 26 = For lowercase English letters

Advantages of Trie

- Faster than HashMap/Set for prefix lookups.
- Keeps data in a sorted-like structure.
- Great for dictionary-based problems.

Limitations

- **High memory usage** (due to 26 children per node).
- **Worse for small datasets** where HashSet might be more efficient.

1. insert(String word)

Goal:

Add a word to the Trie so it can be searched later.

Step-by-step:

1. Start at the root node.
2. For each character ch in the word:
    - Calculate the index: int index = ch - 'a'; (0 for 'a', 1 for 'b', ..., 25 for 'z').
    - If children[index] is null, create a new TrieNode.
    - Move to that child node.
3. After the loop, mark the last node as isEndOfWord = true.

2. search(String word)
Goal:
Check if a complete word exists in the Trie.

Step-by-step:

1. Start at the root node.
2. For each character ch in the word:
    - Calculate index.
    - If children[index] == null, return false (word not present).
    - Move to that child.
3. After the loop, **check if current node has isEndOfWord = true**:
    - If yes → return true
    - If not → return false (it's just a prefix, not a full word)

3. startsWith(String prefix)
Goal:
Check if **any word** in the Trie starts with a given prefix.

Step-by-step:

1. Start at the root.
2. For each character ch in the prefix:
    - Calculate index.
    - If children[index] == null, return false (prefix not found).
    - Move to that child.
3. After the loop, return true (prefix exists, whether or not it's a full word).

4. **getSuggestions(String prefix)**
Phase 1: Traverse to End of Prefix

1. Start from the root node.
2. For each character in the prefix:
    - Convert character to index (0–25).
    - Move to the corresponding child node.

   - If any character is missing, return an empty list (prefix not found).

Phase 2: Collect Words Using DFS

1. Once the prefix node is reached, initialize an empty result list.
2. Perform a depth-first search (DFS) from this node.
3. If a node marks the end of a word, add the word to the result list.
4. Recursively visit all non-null children (a–z) of the current node.
5. After DFS completes, return the result list.

#	Problem Name	Difficulty	Concept	Link
1	**Implement Trie (Prefix Tree)**	Medium	Basic Trie insert/search/prefix	🔗 LeetCode 208
2	**Design Add and Search Words Data Structure**	Medium	Trie with wildcard search (.)	🔗 LeetCode 211
3	**Replace Words**	Medium	Dictionary Trie + Replace	🔗 LeetCode 648
4	**Longest Word in Dictionary**	Medium	Trie + DFS for longest valid path	🔗 LeetCode 720
5	**Word Search II**	Hard	Trie + Backtracking	🔗 LeetCode 212
6	**Prefix and Suffix Search**	Hard	Combined prefix/suffix Trie	🔗 LeetCode 745
7	**Palindrome Pairs**	Hard	Trie + Reverse words + Palindrome logic	🔗 LeetCode 336
8	**Search Suggestions System**	Medium	Trie + Autocomplete	🔗 LeetCode 1268
9	**Stream of Characters**	Hard	Trie with reversed word stream	🔗 LeetCode 1032

10	**Sum of Prefix Scores of Strings**	Medium	Trie + prefix frequency sum	🔗 LeetCode 2416