# Documentation: WordDictionary Class (Add and Search Words Data Structure)

## Overview:

- The WordDictionary class is designed to support adding words and finding whether a given word (or a pattern containing wildcard characters .) matches any previously added word in the data structure. The wildcard character. can match any letter, and the structure efficiently handles such searches.
- *This class is particularly useful for scenarios where:*
  - You want to store a large collection of words.
  - You need to search for words with potential wildcard characters.

## Key Components:

### 1. TrieNode Class:

- Each node in the Trie structure.
- ***Stores:***
  - A dictionary (children) where each key is a character, and the value is the child TrieNode corresponding to that character.
  - A boolean (is_end) indicating if the node marks the end of a valid word in the Trie.

### 2. WordDictionary Class:

- Manages the root of the Trie and provides methods to add words and search for them.

## Methods:

1. **__init__():**
   - ***Description:*** Initializes the WordDictionary object by creating the root node of the Trie.
   - ***Purpose:*** Sets up an empty Trie to which words will be added and searched.
   - ***Attributes:***
     - *root:* The root node of the Trie (an instance of TrieNode).

2. **addWord(word: str) -> None:**

- ***Description:*** Adds a word to the Trie.
- ***Parameters:***
  - *word (str):* The word to be added. It consists of lowercase English letters.
- ***Process:***
  - Starts from the root of the Trie.
  - *For each character in the word:*
    - ✓ Checks if the character exists as a child of the current node.
    - ✓ If the character is not found, a new node is created.
    - ✓ Proceeds to the next character in the word.
  - Marks the last node as the end of a word.
- ***Complexity:***
  - *Time complexity:* O(L), where L is the length of the word being added.

3. **search(word: str) -> bool:**

- ***Description:*** Searches for a word or a pattern in the Trie. The word may contain wildcard characters (.) which can match any letter.
- ***Parameters:***
  - *word (str):* The word or pattern to be searched. It consists of lowercase English letters and/or dots (.).
- ***Process:***
  - Utilizes a recursive Depth-First Search (DFS) strategy.
  - *For each character in the word:*
    - ✓ If the character is a regular letter, the search continues along the corresponding child node.
    - ✓ If the character is . (wildcard), the search explores all possible child nodes at that level of the Trie.
  - Returns True if a word matches the pattern exactly (reaching the end of the word and the is_end flag of the final node is True), otherwise returns False.
- ***Wildcard Handling:***
  - When the character . is encountered, all child nodes are explored, and the search continues for each possible branch.
- ***Complexity:***
  - *Time complexity:*
    - ✓ Best case: O(L), where L is the length of the word.
    - ✓ Worst case: O(N  L), where N is the number of possible branches (wildcards) and L is the word length.

## Example Use Cases:

1. **Adding Words:**

- wordDictionary.addWord("bad") adds the word "bad" to the Trie.
- wordDictionary.addWord("dad") adds the word "dad" to the Trie.
- wordDictionary.addWord("mad") adds the word "mad" to the Trie.

2. **Searching for Words:**
    - wordDictionary.search("pad") returns False since the word "pad" was not added.
    - wordDictionary.search("bad") returns True because "bad" exists in the Trie.
    - wordDictionary.search(".ad") returns True because the pattern ".ad" can match "bad," "dad," or "mad."
    - wordDictionary.search("b..") returns True because the pattern "b.." can match "bad."

## Edge Cases:

1. **Word Not Found:**
    - If a word that doesn't exist in the Trie is searched, the search method will return False.

2. **Handling Wildcards:**
    - If . is used as a wildcard, all possible child nodes at that level are explored. This can increase the complexity but allows flexibility in matching patterns.

3. **Word Length:**
    - The word can be up to 25 characters long, but the Trie structure ensures efficient handling of such lengths.

## Constraints:

- The length of the word to be added or searched is between 1 and 25 characters.
- Words consist of lowercase English letters.
- Search words can contain up to 2 wildcards (.).
- There can be up to $10^4$ operations (additions and searches).

## Performance Considerations:

- **Efficiency:** The Trie structure provides efficient word insertion and lookup. The recursive DFS approach effectively handles searching for patterns with wildcards.

- **Scalability:** The design is optimized to handle up to 10^4 operations, ensuring it works well even with large datasets of words.

## Conclusion:

- The WordDictionary class, using a Trie data structure, efficiently supports the addition and search of words with or without wildcard characters. It is well-suited for handling scenarios where pattern matching is required on a large set of words.