Solution: Implement Trie

Let's solve the Implement Trie problem using the Trie pattern.

We'll cover the following

Statement

Trie is a tree-like data structure used to store strings. The tries are also called **prefix trees** because they provide very efficient prefix-matching operations. Implement a trie data structure with three functions that perform the following tasks:

- **Insert (word):** This inserts a word into the trie.
- **Search (word):** This searches the given word in the trie and returns TRUE, if found. Otherwise, return FALSE.
- **Search prefix (prefix):** This searches the given prefix in the trie and returns TRUE, if found. Otherwise, return FALSE.

Constraints:

- $1 \leq$ `word.length`, `prefix.length` ≤ 2000
- The strings consist only of lowercase English letters.
- At most, 3×10^3 calls in total will be made to the functions.

Pattern: Trie

There are multiple data structures that can be used to store strings. However, for a majority of them, searching a string requires a complete traversal of the stored data. A more efficient approach is to use a trie.

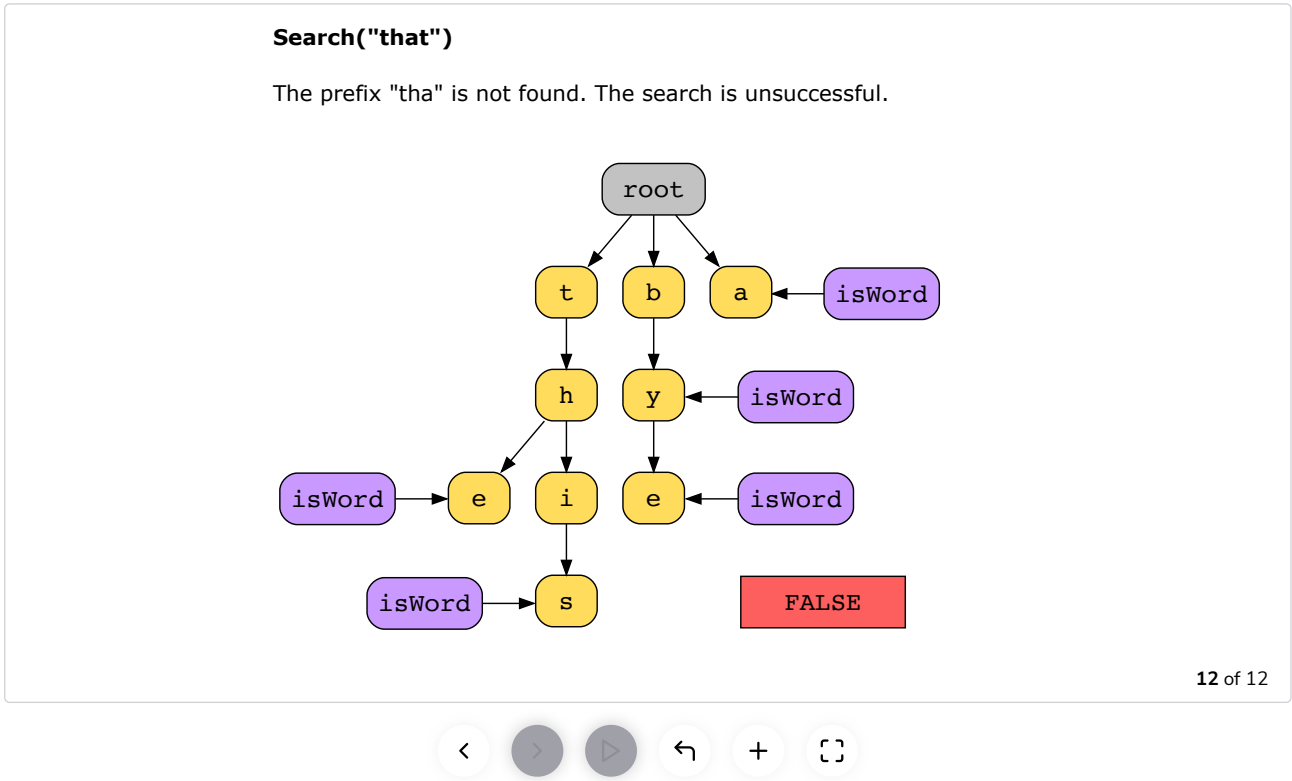
A trie is a tree of characters. The trie takes a word as a parameter and creates a new node for each character of that word. This way, the given string is searched character by character and does not require an exhaustive search. Therefore, this problem fits perfectly in the trie pattern.

Solution

To implement a `Trie` class, we'll initialize the root node of the tree in the constructor. This node will be of the `Node` type. The `Node` class contains a dictionary of nodes and a boolean variable, which determines if the character is at the end of a word.

Let's discuss the following functions now:

- **Insert(word):** In this function, we take a word as input. We begin from the root node and iterate over the string one character at a time. At each node, we check whether or not a child node with the character is present. If it's not present, a new node is initialized. For the last character of the word, we also set the boolean variable to TRUE for the corresponding node.
- **Search(word):** In this function, we start traversing from the root node and check if the first character is the child of the root node. If so, we move on to that node and check its children for the next character. If, at any point, we do not find the node corresponding to a character, we return FALSE. Alternatively, we return FALSE when we find the whole word, but the boolean variable is not TRUE for the last node, signaling that the word doesn't end here. We only return TRUE when all the characters match, and the word ends at that point.
- **Search prefix(prefix):** This function is mostly the same as the search function. The only exception is that we do not check if the boolean variable is also set to TRUE in the last-found node because we aren't looking for a complete word but just a prefix. In the trie above, for instance, `searchPrefix("by")` should return TRUE.



Let's look at the code for this solution below:

Java

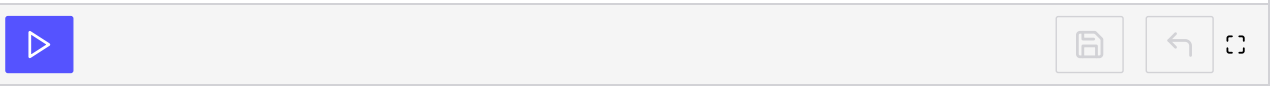
main.java

TrieNode.java

Print.java

```
23        }
24
25        // Once the a complete word is added to the trie,
26        // set boolean variable to true.
27        trieNode.isWord = true;
28    }
29
30    // A function to search for a word in the trie.
31    public boolean search(String word) {
32        TrieNode trieNode = this.root;
```

```
33
34        // Iterate over the word character by character.
35        for (int i = 0; i < word.length(); i++) {
36            char c = word.charAt(i);
37            // If there is no any child of a node,
38            // return false as we do not find the word.
39            if (!trieNode.children.containsKey(c)) {
40              return false;
41            }
42
43          trieNode = trieNode.children.get(c);
44        }
45
46        // Return the word after traversing all nodes as it is found.
47        return trieNode.isWord;
48    }
49
50    // A function to search for a prefix of a word in the trie.
```



Implement Trie

To recap, the solution to this problem can be divided into the following parts:

- **Insert()**: Add words to a trie by iterating character by character and checking if each character has a corresponding node. If it does not, a new node is created. The boolean variable is set to TRUE for the last node.
- **Search()**: Search by starting from the root and going down the trie, checking character by character. For a complete word, check if the boolean variable is set to TRUE.
- **Search prefix()**: Search by starting from the root and going down the trie. However, the boolean variable doesn't necessarily have to be TRUE.

Time complexity

- **Insert()**: The time complexity is $O(l)$, where l is the length of the word being inserted.
- **Search():** The time complexity is $O(l)$, where l is the length of the word that we need to search in the trie.
- **Search prefix():** The time complexity is $O(l)$, where l is the length of the prefix that we need to search in the trie.

Space complexity

- **Insert():** The space complexity is $O(l)$ because, in the worst case, we will add l nodes in the trie.
- **Search():** The space complexity is $O(1)$ because constant space is used while searching the trie.
- **Search prefix():** The space complexity is $O(1)$ because constant space is used while searching the trie.