676. Implement Magic Dictionary

Medium | Depth-First Search | Design | Trie | Hash Table | String — Leetcode Link

Problem Description

The problem requires designing a data structure named `MagicDictionary` that can do two things: first, it should be able to store a list of distinct words. Second, it should provide a function to determine whether we can change exactly one character in a given string so that it matches any of the words stored in the dictionary.

When creating this data structure, we need to perform the following operations:

- `buildDict`: This method initializes the dictionary with an array of unique strings.
- `search`: This method takes a string `searchWord` and returns `true` if changing exactly one character in `searchWord` results in a word that is in the dictionary. It returns `false` otherwise.

The challenge lies in determining an efficient way to validate if a word with one character changed is in the dictionary, considering the dictionary can contain any number of words, and the search function might be called multiple times.

Intuition

The solution is to preprocess the dictionary in such a way that searching is efficient. Rather than checking every word in the dictionary during each search, the idea is to generate a pattern for each word by replacing each character in turn with a placeholder (e.g., an asterisk '*'). The patterns are then stored in a data structure with quick access time. By using this pattern-matching strategy:

1. We create all possible patterns from each word in the dictionary by replacing each character with a '*'. This yields a generalized form of the words.
2. When searching for a match of `searchWord`, we generate the same patterns for it.
3. We then verify if these patterns match any patterns from the dictionary:
   - If a pattern has a count greater than 1, we can be certain that `searchWord` can match a different word in the dictionary (since we store only unique words).
   - If a pattern has a count of 1, and the `searchWord` is not already in the set of dictionary words, we know that there is exactly one different character.
4. If any of the generated patterns of `searchWord` satisfy the conditions above, we can return `true`. Otherwise, `false`.

Utilizing a counter to track the number of occurrences of patterns and a set to keep track of the words ensures a quick look-up and decision during the search operation.

Solution Approach

To implement the `MagicDictionary`, we use a set and a `Counter` from Python's collections library. The set, `self.s`, stores the words in the dictionary to ensure the uniqueness of the elements and to provide quick access for checks. The Counter, `self.cnt`, tracks the count of all the possible patterns formed by replacing each character in the words with the placeholder asterisk '*'. Here's a breakdown of how each part of the implementation contributes to the solution:

- The `__init__` method initializes the data structures (`self.s` for the set and `self.cnt` for the Counter) used to hold the words and patterns.
- The `gen` method is a helper function that generates all possible patterns for a given word by replacing each character with ''. *For example, for the word "hello", it will produce ["ello", "hllo", "he*lo", "helo", "hell"].
- The `buildDict` method constructs the set with the given dictionary words and uses the `gen` method to create and count the patterns of each word. By iterating over the words in the dictionary and their generated patterns, we populate `self.cnt`.
- In the `search` method, for a given `searchWord`, we generate all possible patterns and then loop over them to verify if we can find a match:
  - If `self.cnt[p] > 1`, it means that there is more than one word in the dictionary that can match the pattern, hence we can have a match with exactly one character different from `searchWord`.
  - If `self.cnt[p] == 1` and the `searchWord` is not in `self.s`, the pattern is unique to one word in the dictionary, and since `searchWord` is not that word, this confirms we can match by changing exactly one character.

In terms of algorithms and patterns, this implementation uses a clever form of pattern matching that simplifies the search space. Instead of comparing the `searchWord` to every word in the dictionary, it only needs to check the generated patterns, significantly improving efficiency, especially when the `search` operation is called multiple times.

By using these data structures and algorithms, the `MagicDictionary` is able to quickly and efficiently determine whether there exists a word in the dictionary that can be formed by changing exactly one character in the `searchWord`.

Example Walkthrough

Let's illustrate the solution approach using a simple example. Suppose we want to initialize our `MagicDictionary` with the words `["hello", "leetcode"]`. Here's how the `MagicDictionary` would be built and utilized:

1. **Initialization**: The `MagicDictionary` is created with an empty set and counter.

2. **Building the Dictionary**:

   - The `buildDict` function is called with `["hello", "leetcode"]`.
   - The `gen` helper function generates patterns by replacing each character with an asterisk `'*'`.
     - For "hello", the generated patterns would be `["*ello", "h*llo", "he*lo", "hel*o", "hell*"]`.
     - For "leetcode", the generated patterns would be `["*eetcode", "l*etcode", "le*tcode", "lee*code", "leet*ode", "leetc*de", "leetcod*"]`.
   - These patterns are added to `self.cnt` counter, and the original words are stored in the set `self.s`.
3. **Search Operation**:

   - Now, let's say we want to search for the word "hullo" using the `search` function.
   - The generated patterns for "hullo" would be `["*ullo", "h*llo", "hu*lo", "hul*o", "hull*"]`.
   - Next, the algorithm goes through each of these patterns:
     - It finds that the pattern `"h*llo"` has a count of 1 in `self.cnt`, and since "hullo" is not in the set `self.s` (which contains "hello"), this means there is exactly one character different in a unique word in the dictionary that matches the pattern `"h*llo"`.
   - Since all conditions for a successful match are met, the `search` function will return `true` for the word "hullo".
4. **Result**:

   - The `search` operation effectively demonstrates the ability to determine if "hullo" can be transformed into a word in the dictionary by changing exactly one letter. It returns `true` without having to compare "hullo" against every word in the dictionary, showcasing the efficiency of the pattern matching approach.

Python Solution

```python
from collections import Counter

class MagicDictionary:

    def __init__(self):
        # Initialize attributes for storing words and counts of generic patterns
        self.words_set = set()
        self.pattern_count = Counter()

    def generate_patterns(self, word):
        # Generate all potential patterns by replacing each character with a '*'
        return [word[:i] + '*' + word[i + 1:] for i in range(len(word))]

    def buildDict(self, dictionary: List[str]) -> None:
        # Build the dictionary from a list of words and count the occurences of the generated patterns
        self.words_set = set(dictionary)
        self.pattern_count = Counter(pattern for word in dictionary for pattern in self.generate_patterns(word))

    def search(self, searchWord: str) -> bool:
        # Search if there is any word in the dictionary that can be obtained by changing exactly one character of the searchWord
        for pattern in self.generate_patterns(searchWord):
            count = self.pattern_count[pattern]
            # Check if more than one word matches the pattern
            # or if one word matches the pattern but it's not the searchWord itself
            if count > 1 or (count == 1 and searchWord not in self.words_set):
                return True
        return False


# Example of using the MagicDictionary class
# obj = MagicDictionary()
# obj.buildDict(["hello", "leetcode"])
# param_2 = obj.search("hhllo")  # Should return True, as hello is in the dictionary after changing one character
# param_3 = obj.search("hello")  # Should return False, as the word is in the dictionary and does not require a change

```

Java Solution

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

class MagicDictionary {
    // Using a set to store the actual words to ensure no duplicates
    private Set<String> wordSet = new HashSet<>();
    // Using a map to store placeholders for words with a character replaced by '*'
    private Map<String, Integer> placeholderCountMap = new HashMap<>();

    /** Constructor for MagicDictionary **/
    public MagicDictionary() {
        // nothing to initialize
    }

    /**
     * Builds a dictionary through a list of words
     *
     * @param dictionary An array of words to be added to the MagicDictionary
     */
    public void buildDict(String[] dictionary) {
        for (String word : dictionary) {
            wordSet.add(word); // add the word to the set
            // get placeholders for the word and update their count in the map
            for (String placeholder : generatePlaceholders(word)) {
                placeholderCountMap.put(placeholder, placeholderCountMap.getOrDefault(placeholder, 0) + 1);
            }
        }
    }

    /**
     * Search if there is any word in the dictionary that can be obtained by changing
     * exactly one character of the searchWord
     *
     * @param searchWord The word to search for in the dictionary
     * @return True if such word exists, False otherwise
     */
    public boolean search(String searchWord) {
        // generate placeholders for the search word
        for (String placeholder : generatePlaceholders(searchWord)) {
            int count = placeholderCountMap.getOrDefault(placeholder, 0);
            // if count is more than 1 or the word itself is not in the set and count is 1
            if (count > 1 || (count == 1 && !wordSet.contains(searchWord))) {
                return true;
            }
        }
        return false;
    }

    /**
     * Generate all possible placeholders for a word by replacing each
     * character with '*'
     *
     * @param word The word to generate placeholders for
     * @return A list of placeholders
     */
    private List<String> generatePlaceholders(String word) {
        List<String> placeholders = new ArrayList<>();
        char[] chars = word.toCharArray(); // convert word to char array for manipulation
        for (int i = 0; i < chars.length; ++i) {
            char originalChar = chars[i];
            chars[i] = '*'; // replace the i-th character with '*'
            placeholders.add(new String(chars)); // add to the list of placeholders
            chars[i] = originalChar; // revert the change to original character
        }
        return placeholders;
    }
}

// Sample usage is shown in the comment below:
/*
MagicDictionary obj = new MagicDictionary();
obj.buildDict(dictionary);
boolean param_2 = obj.search(searchWord);
*/

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <unordered_set>
#include <unordered_map>
using namespace std;

class MagicDictionary {
public:
    /** Initialize your data structure here. */
    MagicDictionary() {
        // Constructor is currently empty; no initialization is needed.
    }

    /** Builds the dictionary from a list of words. */
    void buildDict(vector<string> dictionary) {
        for (const string& word : dictionary) {
            // Insert the word into the set.
            wordsSet.insert(word);

            // For each possible generic representation with one letter replaced by a '*', update its count.
            for (const string& genericWord : generateGenericWords(word)) {
                genericWordCounts[genericWord]++;
            }
        }
    }

    /** Searches if there is any word in the dictionary that can be matched to searchWord after modifying exactly one character. */
    bool search(string searchWord) {
        for (const string& genericWord : generateGenericWords(searchWord)) {
            if (genericWordCounts[genericWord] > 1 ||
                (genericWordCounts[genericWord] == 1 && wordsSet.count(searchWord) == 0)) {
                return true;
            }
        }
        return false;
    }

private:
    unordered_set<string> wordsSet; // Set to store words.
    unordered_map<string, int> genericWordCounts; // Map to store the counts of all possible generic representations.

    /** Generates all possible generic words by replacing each letter of the word with a '*' one at a time. */
    vector<string> generateGenericWords(string word) {
        vector<string> genericWords;
        for (int i = 0; i < word.size(); ++i) {
            char originalChar = word[i];
            word[i] = '*';
            genericWords.push_back(word);
            word[i] = originalChar;
        }
        return genericWords;
    }
};

// The MagicDictionary class can be used as follows:
// MagicDictionary* obj = new MagicDictionary();
// obj->buildDict(dictionary);
// bool result = obj->search(searchWord);

```

Typescript Solution

```typescript
// Imports skipped since global definitions don't require import statements

// Global variables to store words and counts of their generic representations
const wordsSet: Set<string> = new Set();
const genericWordCounts: Map<string, number> = new Map();

/** Builds the dictionary from a list of words. */
function buildDict(dictionary: string[]): void {
    dictionary.forEach(word => {
        // Insert the word into the set
        wordsSet.add(word);

        // For each possible generic representation with one letter replaced by '*', update its count
        const genericWords = generateGenericWords(word);
        genericWords.forEach(genericWord => {
            const count = genericWordCounts.get(genericWord) || 0;
            genericWordCounts.set(genericWord, count + 1);
        });
    });
}

/** Searches if there is any word in the dictionary that can be matched to searchWord after modifying exactly one character. */
function search(searchWord: string): boolean {
    const genericWords = generateGenericWords(searchWord);

    for (const genericWord of genericWords) {
        const count = genericWordCounts.get(genericWord) || 0;
        if (count > 1 || (count === 1 && !wordsSet.has(searchWord))) {
            return true;
        }
    }

    return false;
}

/** Generates all possible generic words by replacing each letter of the word with '*' one at a time. */
function generateGenericWords(word: string): string[] {
    const genericWords: string[] = [];

    for (let i = 0; i < word.length; i++) {
        const originalChar = word[i];
        // Replace one letter with '*'
        word = word.substring(0, i) + '*' + word.substring(i + 1);
        genericWords.push(word);

        // Restore the original letter
        word = word.substring(0, i) + originalChar + word.substring(i + 1);
    }

    return genericWords;
}

// Example usage:
// buildDict(["hello", "leetcode"]);
// const result: boolean = search("hhllo"); // Should return true

```

Time and Space Complexity

Time Complexity

- `__init__`: O(1) since no operation is performed in the initialization process.
- `gen`: The time complexity is O(N) where N is the length of the word since we generate a new string for each character by omitting that character.
- `buildDict`:
  - The time complexity of constructing the set `s` is O(M * K) where M is the number of words in the dictionary and K is the average length of the words since we have to copy each word into the set.
  - The time complexity of creating the counter is O(M * N * K) where N is the average length of the words in the dictionary. For each word, we create N different patterns (by the `gen` function) and for each pattern we incur the cost of insertion into `cnt`. Note that insertion into a counter (which is basically a hash map) is typically O(1), so the main cost is in generating the patterns and iterating over the words.
- `search`:
  - The `search` function involves generating patterns from the `searchWord` whose time complexity is O(N), where N is the length of `searchWord`.
  - The time complexity of searching in the counter is potentially O(1) for each pattern. Since there are N patterns, the total complexity in the worst-case for searching is O(N).
  - The `searchWord not in self.s` check also has a time complexity of O(1) on average due to the set's underlying hash table.
  - Combining these, the total time complexity for `search` is O(N).

In summary, if `N` is the maximum length of a word, `M` is the number of words in the dictionary, and `searchWord` is the word to search for with length `N`, then the time complexity for `buildDict` is O(M * N * K) and for `search` is O(N).

Space Complexity

- `self.s`: Takes O(M * K) space to store each word in the dictionary.
- `self.cnt`: The counter could at most hold O(M * N) unique patterns (if every generated pattern from every word is unique).
- For `gen` function: Since it creates a list of strings, in the worst case, it could take O(N^2) space to store these strings if we consider the space used by all intermediate strings. However, in most practical scenarios where strings are immutable and the language runtime optimizes string storage, the actual additional space complexity incurred by `gen` is closer to O(N).

Overall, the space complexity is O(M * N + M * K) which is attributable to the size of the words in the dictionary and the patterns generated.

Note: The actual space used by some data structures like hash maps can be larger than the number of elements due to the need for a low load factor to maintain their performance characteristics. The complexities mentioned do not account for these implementation details.