MapMyIndia – Assignment Round

Sarthak Gupta – 2019AAPS0290G

f20190290@goa.bits-pilani.ac.in || +91 9971565132

Code:

```
class DictionaryTrieNode:
    def __init__(self):
        self.children = {}
        self.is_word = False


class DictionaryTrie:
    def __init__(self):
        self.root = DictionaryTrieNode()

    def insert_word(self, word):
        # Inserts a word into the dictionary trie.
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = DictionaryTrieNode()
            node = node.children[char]
        node.is_word = True

    def search_word(self, word):
        # Searches for exact matches of a word in the dictionary trie.
        node = self._get_prefix_node(word)
        if not node:
            return []

        results = []
        self._dfs_traversal(node, word, '', results)
        return results

    def _get_prefix_node(self, prefix):
        # Traverses the trie to find the node representing the given prefix.
        node = self.root
        for char in prefix:
            if char not in node.children:
                return None
            node = node.children[char]
        return node

    def _dfs_traversal(self, node, remaining_word, current_word, results):
        # Performs a depth-first traversal to find all words matching the
prefix.
        if not remaining_word:
```

```
            if node.is_word:
                results.append(current_word)
            return

        char = remaining_word[0]
        if char in node.children:
            child_node = node.children[char]
            next_word = current_word + char
            self._dfs_traversal(child_node, remaining_word[1:], next_word,
results)

    def find_similar_words(self, word, max_length_diff=2, max_typos=2):
        # Finds similar words to the given word in the dictionary trie.
        all_words = self._get_all_words()
        similar_words = []
        for dict_word in all_words:
            if abs(len(dict_word) - len(word)) <= max_length_diff:
                distance = self._calculate_similarity(word, dict_word)
                if distance is not None and distance <= max_typos:
                    similarity_score = self._calculate_similarity_score(word,
dict_word, distance)
                    similar_words.append((dict_word, similarity_score))
        similar_words.sort(key=lambda x: x[1], reverse=True)
        return similar_words

    def _get_all_words(self):
        # Returns a list of all words in the dictionary trie.
        words = []
        self._dfs_collect_words(self.root, '', words)
        return words

    def _dfs_collect_words(self, node, current_word, words):
        # Performs a depth-first traversal to collect all words in the trie.
        if node.is_word:
            words.append(current_word)

        for char, child_node in node.children.items():
            next_word = current_word + char
            self._dfs_collect_words(child_node, next_word, words)

    @staticmethod
    def _calculate_similarity(word1, word2):
        # Calculates the Levenshtein distance between two words.
        len1 = len(word1)
        len2 = len(word2)
        dp = [[0] * (len2 + 1) for _ in range(len1 + 1)]

        for i in range(len1 + 1):
```

```
            dp[i][0] = i
        for j in range(len2 + 1):
            dp[0][j] = j

        for i in range(1, len1 + 1):
            for j in range(1, len2 + 1):
                cost = 0 if word1[i - 1] == word2[j - 1] else 1
                dp[i][j] = min(
                    dp[i - 1][j] + 1,       # Deletion
                    dp[i][j - 1] + 1,       # Insertion
                    dp[i - 1][j - 1] + cost  # Substitution
                )

                if i > 1 and j > 1 and word1[i - 1] == word2[j - 2] and
word1[i - 2] == word2[j - 1]:
                    dp[i][j] = min(dp[i][j], dp[i - 2][j - 2] + cost)  #
Transposition

        distance = dp[len1][len2]
        if distance is None:
            distance = float('inf')

        return distance

    @staticmethod
    def _calculate_similarity_score(word1, word2, distance):
        # Calculates the similarity score between two words based on their
distance.
        max_len = max(len(word1), len(word2))
        return (max_len - distance) / max_len


def build_dictionary():
    # Builds a dictionary trie from a data file.
    dictionary_trie = DictionaryTrie()
    with open("datafile.txt", "r") as file:
        for line in file:
            word = line.strip().lower()  # Remove leading/trailing whitespaces
and convert to lowercase
            dictionary_trie.insert_word(word)
    return dictionary_trie


def main():
    # Main function to search for words and find similar words in the
dictionary.
    dictionary = build_dictionary()
    search_term = input("Enter a word to search: ").lower()
```

```
    results = dictionary.search_word(search_term)
    similar_words = dictionary.find_similar_words(search_term)

    if results:
        print("Exact matches:")
        for word in results:
            print(word)

    if not results and similar_words:
        print("Showing similar words:")
        top_similar_words = sorted(similar_words, key=lambda x: x[1],
reverse=True)[:10]
        for word, similarity_score in top_similar_words:
            if similarity_score > 0.5:
                print(f"{word}")
                # print(f"{word} (Similarity Score: {similarity_score})") #
Uncomment to show similarity score

    if not results and not similar_words:
        print("No matches found.")


if __name__ == "__main__":
    main()
```

Explanation:

The Dictionary Trie is a versatile and efficient data structure that allows us to search for words and retrieve similar words with ease. It's particularly useful when we need to find words that match a given prefix or identify words similar to a target word. In this essay, we'll explore the inner workings of the Dictionary Trie and delve into its various functions, explaining their significance in a beginner-friendly manner.

1. Structure of the Dictionary Trie: The Dictionary Trie consists of two primary classes: DictionaryTrieNode and DictionaryTrie. The DictionaryTrieNode class represents a single node in the trie, while the DictionaryTrie class acts as a wrapper around the trie itself.
2. Inserting Words: The insert_word method in the DictionaryTrie class allows us to add words to the trie. It works by examining each character in a word and creating child nodes if they don't already exist. Once the traversal is complete, the last node is marked as a complete word by setting the is_word flag to True.
3. Searching for Exact Words: The search_word method is incredibly handy for finding exact matches of a word in the trie. It begins by traversing the trie based on the characters of the input word. If the traversal reaches the end of the word and the current node represents a complete word, the word is added to the results. This method provides an efficient way to retrieve all words that match a given prefix.

4. Retrieving Similar Words: One of the most exciting features of the Dictionary Trie is the find_similar_words method. It allows us to find words that are similar to a given target word based on specific criteria. Here's how it works:

4.1. Length and Typo Check: The algorithm starts by obtaining all the words stored in the trie using the _get_all_words method. It then proceeds to compare the length of the target word with each word from the dictionary. If the difference in lengths falls within a specified threshold, the algorithm calculates the Levenshtein distance between the target word and the current word using the _calculate_similarity method.

4.2. Similarity Score Calculation: To quantify the similarity between two words, the algorithm computes a similarity score using the _calculate_similarity_score method. This score takes into account the Levenshtein distance and the maximum length of the two words being compared. A higher similarity score indicates a closer match between the words.

4.3. Filtering and Sorting: Next, the algorithm filters out words with similarity scores below a specific threshold, discarding less similar words. It then sorts the remaining similar words in descending order based on their similarity scores. This ensures that the most similar words appear at the beginning of the list.

5. Interacting with the User: The main function serves as the entry point for user interaction with the Dictionary Trie. It prompts the user to enter a word to search and subsequently calls the relevant methods to find exact matches and similar words. The results are then displayed to the user, providing a seamless and intuitive experience.

Conclusion: The Dictionary Trie offers an efficient and adaptable solution for word search and retrieval of similar words. Its underlying trie structure enables quick prefix-based searching, while the similarity algorithm empowers us to find words closely related to a target word. By understanding the concepts behind the Dictionary Trie's implementation, developers gain a powerful tool for effectively managing word-related operations in various applications, such as autocomplete systems and spell checkers.