**Dynamic Word Suggestion System: Class Overview**

This document provides an in-depth overview of the key classes used in the **Dynamic Word Suggestion System**. Each class is detailed with its purpose, roles, members, and methods. Additionally, the reasoning behind each data structure and the logical implementation is explained.

**1. Trie Class**

**Purpose:**

The **Trie class** is designed to enable fast and efficient retrieval of words based on prefixes, making it the cornerstone of real-time suggestions.

**Members:**

**Private Members:**

* struct TrieNode: Represents a node in the Trie.
  + bool is\_end: Marks the end of a word.
  + unordered\_map<char, TrieNode\*> children: Maps characters to their respective child nodes.
* TrieNode\* root: The root node of the Trie structure.

**Public Members:**

* Trie(): Constructor to initialize the Trie.
* void insert\_word(const string& word): Inserts a word into the Trie.
* vector<string> search\_prefix(const string& prefix): Retrieves all words in the Trie that start with a given prefix.
* bool word\_exists(const string& word): Checks whether a specific word exists in the Trie.

**Rationale:**

* **Efficiency:** Supports O(k) complexity for prefix-based lookups, where *k* is the length of the prefix.
* **Without a Trie:** Searching for all words starting with a prefix in a large dataset would require linear scans, making real-time suggestions impractical.

**2. HashTable Class**

**Purpose:**

The **HashTable class** provides fast exact-match word lookups, ensuring efficient validation of word existence.

**Members:**

**Private Members:**

* unordered\_set<string> word\_set: Stores words for quick O(1) average-time lookups.

**Public Members:**

* HashTable(): Constructor to initialize the hash table.
* void add\_word(const string& word): Adds a word to the hash table.
* bool word\_exists(const string& word): Checks if a word exists in the hash table.

**Rationale:**

* **Efficiency:** Provides O(1) average time complexity for lookups.
* **Without it:** Validating word existence would require linear searches, slowing down the system for larger datasets.

**3. BloomFilter Class (Optional)**

**Purpose:**

The **BloomFilter class** is a probabilistic data structure designed to save memory and provide fast membership checks for large datasets.

**Members:**

**Private Members:**

* vector<bool> bit\_array: The bit array that implements the Bloom Filter.
* int size: Size of the bit array.
* int num\_hashes: Number of hash functions used.

**Public Members:**

* BloomFilter(int size, int num\_hashes): Constructor to initialize the Bloom Filter.
* void add\_word(const string& word): Adds a word to the Bloom Filter.
* bool word\_exists(const string& word): Checks if a word might exist.

**Rationale:**

* **Efficiency:** Uses minimal memory to check for probable existence, reducing the need for expensive checks in other structures.
* **Without it:** Larger memory and computational resources would be needed for datasets with millions of words.

**4. WordManager Class**

**Purpose:**

The **WordManager class** organizes predefined words into vectors based on their starting alphabet, optimizing search performance.

**Members:**

**Private Members:**

* vector<string> alphabet\_vectors[26]: An array of 26 vectors for storing words based on their starting letters (e.g., alphabet\_vectors[0] for words starting with 'A').

**Public Members:**

* WordManager(): Constructor to initialize the word storage.
* void add\_word(const string& word): Adds a word to the appropriate vector based on its starting letter.
* vector<string> get\_words(const char& prefix): Retrieves all words from the vector corresponding to the prefix's starting letter.

**Rationale:**

* **Efficiency:** Reduces the search space by categorizing words by their starting letter.
* **Without it:** Linear scans through the entire dataset would be required for every prefix lookup.

**5. GUIManager Class**

**Purpose:**

The **GUIManager class** is responsible for user interaction, displaying suggestions, and coordinating backend logic.

**Members:**

**Private Members:**

* QLineEdit\* search\_bar: The input field where users type queries.
* QListWidget\* suggestion\_list: Dropdown displaying suggestions.
* Trie\* trie: Pointer to the Trie for prefix-based lookups.
* HashTable\* hash\_table: Pointer to the hash table for exact-match validation.
* WordManager\* word\_manager: Pointer to the WordManager for accessing predefined words.

**Public Members:**

* GUIManager(): Constructor to set up GUI components.
* void handle\_user\_input(const QString& input): Processes user input, fetching and displaying suggestions.
* void add\_word\_to\_system(const QString& word): Adds a word to the system and updates the Trie, HashTable, and WordManager.

**Rationale:**

* **Central Role:** Serves as the bridge between user actions and backend logic.
* **Without it:** There would be no unified mechanism to manage user interaction and integrate various components.

**Class Dependencies**

**Interaction Summary:**

* **Trie Class:** Used by GUIManager for prefix-based lookups.
* **HashTable Class:** Works with GUIManager for exact-match validation.
* **BloomFilter Class:** Used in conjunction with WordManager to optimize membership checks.
* **WordManager Class:** Interfaces with the Trie for storing and retrieving predefined words.
* **GUIManager Class:** Coordinates the interaction between the user, backend logic, and data structures.

**Logical Reasoning for Data Structures**

1. **Trie:**
   * Chosen for its efficiency in prefix-based lookups with O(k) complexity.
   * Alternative: Linear scans through vectors would degrade performance.
2. **HashTable:**
   * Provides O(1) average time complexity for word existence checks.
   * Alternative: Slower linear searches would compromise real-time performance.
3. **BloomFilter:**
   * Optimizes memory usage for large datasets and avoids redundant checks.
   * Alternative: Increased latency and memory overhead without its probabilistic checks.
4. **Vectors in WordManager:**
   * Categorizes words for quick prefix-specific searches.
   * Alternative: Searching through a single, unsorted dataset would increase computational overhead.

**Conclusion**

The **Dynamic Word Suggestion System** is built on a robust foundation of well-chosen data structures and logical design principles. The synergy between classes ensures efficient word storage, retrieval, and real-time suggestions while maintaining scalability and a user-friendly interface. This modular design also supports extensibility for future enhancements, such as multi-language support or additional data sources.