Proposal for the Project: Dynamic Word Suggestion System with Efficient Lookup Using C++

Introduction

The Dynamic Word Suggestion System is a cutting-edge application designed to enhance user experience by providing real-time word suggestions as users type into a search bar. Built using C++ and the Qt framework, the system integrates advanced data structures like Tries, Hash Tables, and Bloom Filters to ensure optimal performance. This project aims to provide a scalable and efficient solution for handling large datasets while maintaining a responsive graphical interface for users.

Objective

The primary objective of this project is to design and implement a system that offers:

1. Real-time word suggestions to assist users with completing partially typed queries.
2. Efficient lookups using optimized data structures to handle large datasets with minimal delay.
3. A scalable and user-friendly interface for both personal and enterprise use.

Project Scope

1. **Core Functionalities**:
    - Typing suggestions based on prefixes.
    - Efficient integration of in-memory data storage using vectors for predefined words.
    - Real-time interaction with a graphical user interface.
2. **Key Features**:
    - Interactive dropdown menus for dynamic suggestions.
    - Scalability to handle millions of words.
    - Extensibility for multiple languages or specialized dictionaries.
3. **Constraints**:
    - Completion within 14 days.
    - The solution must be memory-efficient and computationally optimized.

Technology Stack

1. **Programming Language**:
    - C++: For implementing core logic, data structures, and algorithms.
2. **Data Storage**:
    - Vectors: For storing predefined word datasets directly in memory.
3. **GUI Framework**:
    - Qt: To build a responsive and intuitive interface.

Data Structures Utilized

1. **Trie (Prefix Tree)**:
    - **Purpose**: Efficient storage and lookup of words based on their prefixes.
    - **Reason for Use**: Provides O(k) complexity for prefix searches, where *k* is the prefix length. This ensures fast suggestions as the user types.
    - **Consequence Without It**: Without a Trie, the system would rely on linear searches through the vector, significantly increasing response times for large datasets.
2. **Hash Table (Hash Set)**:
    - **Purpose**: Quick validation of word existence.
    - **Reason for Use**: Hash tables allow O(1) average time complexity for exact word lookups.
    - **Consequence Without It**: Validating word existence would require linear searches, degrading performance for large datasets.
3. **Bloom Filter (Optional)**:
    - **Purpose**: Memory-efficient probabilistic testing of word existence.
    - **Reason for Use**: Reduces redundant checks by quickly identifying if a word might exist in the dataset.
    - **Consequence Without It**: Increased resource consumption due to additional lookups.

Vector-Based Word Storage

Each vector will store predefined words categorized by their starting alphabet. For instance:

- `vector<string> a_words` will store all words starting with 'A'.
- `vector<string> b_words` will store all words starting with 'B', and so on.
  This organization will minimize the search space, allowing faster prefix matching and retrieval when paired with the Trie structure.

System Features

1. **Search Bar with Real-Time Suggestions**:
    - Displays suggestions dynamically as the user types.
2. **In-Memory Word Storage**:
    - Uses vectors for efficient management and retrieval of predefined words.
3. **Interactive GUI**:
    - Built using Qt for seamless user experience.
4. **Scalability**:

- Handles millions of words without performance degradation.

Implementation Timeline

Day	Task	Description
1-2	Requirement Analysis & Design	Finalize structure, data flow, and GUI wireframe.
3-5	Implement Core Classes	Develop Trie, Hash Table, and optional Bloom Filter.
6-7	Word Storage Implementation	Populate vectors with predefined word datasets.
8-10	Develop GUI	Build an interactive search bar and dropdown.
11-12	Integration and Testing	Integrate components and perform functional testing.
13	Optimization	Enhance efficiency and optimize memory usage.
14	Final Testing and Documentation	Conduct final testing and prepare documentation.

Program Structure

Classes

Class	Purpose	Public Members	Private Members
`TrieNode`	Represents a node in the Trie	`is_end_of_word, children`	None
`Trie`	Stores words and supports prefix-based searches.	`insert_word(), search_prefix()`	`root (pointer to the root node)`
`HashTable`	Handles quick word existence lookups.	`insert_word(), search_word()`	`table (array of buckets)`
`BloomFilter`	(Optional) Memory-efficient membership testing.	`add_word(), check_word()`	`bit_array, hash_functions`
`WordManager`	Manages word storage using vectors.	`add_word(), get_words()`	`vectors for each alphabet`
`GUIManager`	Handles GUI interactions and integrates the search system.	`display_suggestions(), get_user_input()`	`ui_elements`

Class Dependencies

1. **Trie**:
    - Utilized for prefix-based lookups in GUIManager.

    - Stores words fetched from the vectors managed by WordManager.
2. **HashTable**:
    - Used by WordManager for validating word existence before insertion.
3. **BloomFilter (Optional)**:
    - Used to pre-filter checks within WordManager.
4. **WordManager**:
    - Interfaces with vectors for in-memory storage and retrieval of word datasets.
5. **GUIManager**:
    - Displays suggestions retrieved from Trie and handles user interactions.

Conclusion

The Dynamic Word Suggestion System is a robust, scalable application designed for fast and accurate word suggestions. By leveraging advanced data structures and in-memory word storage using vectors, the system achieves optimal performance and scalability. The user-friendly GUI, coupled with efficient backend processing, ensures a seamless experience for users. With careful planning and structured development, this project will be completed within 14 days, providing a valuable tool for real-time word suggestions.