PROJECT REPORT: USING DICTIONARY

SUBMITTED BY:

Anirudh S Nagekar 24PG00070

Suchithra p s 24PG00084

Varsha r a 24PG00087

Pradeep Burli 24PG00078

Under the guidance of

Prof. Usha Subramanian

Course: MCA&MSC DATA SCIENCE

Semester-1

School of engineering

TABLE OF CONTENTS

1.Introduction

Autocomplete systems have become essential features in many text-based tools such as search engines, messaging apps, and document editors. These systems assist users by suggesting possible completions for the word or phrase being typed, thus speeding up the process of entering text. Autocomplete systems typically work by matching the characters typed by the user with a predefined vocabulary and providing suggestions that start with the typed prefix.

When a user begins typing a word, the system searches its vocabulary for matching prefixes and offers a list of potential word completions. These suggestions are updated dynamically as the user continues typing. The system can be designed to return a limited number (k) of suggestions based on the entered prefix. As the user types each subsequent character, the list of suggestions narrows down, helping the user to quickly select the desired word or phrase.

In this project, the primary objective is to design and implement an autocomplete feature for a document management system, aimed at improving the speed and efficiency of researchers and writers. The autocomplete feature will help users by suggesting possible completions for words as they type, ultimately increasing their writing speed.

2.Methodology

The methodology for developing an Autocomplete System using a dictionary involves several steps aimed at creating an efficient, real-time word suggestion feature for a document management system. The approach is broken down into phases that cover the reading of input data, data structure implementation, user input handling, and system performance evaluation.

 File Reading: The first step in the methodology is to read the vocabulary data from a file. This file contains prefixes and their associated words. The file must be parsed, and each prefix along with its corresponding words will be stored in an appropriate data structure.

 Vocabulary Expansion: The system should allow for future vocabulary expansion. As new words are added to the vocabulary, the system should automatically update its internal data structure to accommodate these new terms

Dictionary Structure: A dictionary is chosen as the core data structure for storing prefixes and their associated words. A dictionary is well-suited for this task due to its efficiency in mapping keys (prefixes) to values (words).

Prefix and Word Association: Each node in the linked list will represent a word associated with a specific prefix. The dictionary's keys will be the prefixes, and the values will be linked lists that store the words starting with that prefix.

- **Prefix Input Handling:** As the user types, the system must monitor the typed input and match it with existing prefixes. This step involves progressively matching the typed prefix with entries in the dictionary.

- **Dynamic Suggestions:** The system will keep track of the user's typed characters and dynamically generate the k possible word completions based on the current prefix. For example, if the user types "abs," the system will suggest the first k words that begin with "abs."

- **Handling Partial Input:** As the user continues typing, the suggestions should update in real-time, displaying new words that match the increasingly specific prefix (e.g., "abse" leads to suggestions like "absent," "absently," and "absenteeism").

- **No Word Selection:** Since the system uses a terminal interface, there will be no ability for the user to select words interactively. Instead, the system will only display suggestions based on the input so far.

- **Suggestion Display:** After each keystroke by the user, the system will display the current suggestions, showing up to **k** words that match the typed prefix. For example, if the prefix "abs" is typed, the system might show words like "absent," "absorb," and "absurd."

- **Output of Prefix-Word Mapping:** At any given time, the system should display all the prefixes and their associated words that have been generated and stored during the execution of the program. This ensures that the user can see the available vocabulary and how it changes as they type.

- **Testing the System:** The final implementation will be tested with various inputs, ensuring that the system correctly matches prefixes and updates the suggestions in real time. Test cases will be created for different prefix lengths and vocabulary sizes to ensure robustness.

 Performance Considerations: The system's performance will be evaluated based on the time it takes to generate suggestions for increasingly complex input, and how it handles larger vocabularies. Given the choice of data structures, we expect good performance, especially when the dictionary size is moderate.

Enhanced Data Structures: In the future, the system could be enhanced by incorporating a Trie to improve the prefix matching speed. A trie would reduce the overhead of storing multiple versions of similar words, leading to a more memory-efficient and faster system.

GUI Integration: While the system currently operates via a terminal interface, future versions could include a GUI that allows users to interactively select words from the suggestions, improving usability.

3. System Design/Project Planning

The autocomplete feature is being integrated into a document management system, which is intended to assist researchers and writers by speeding up their writing process. The system will use a dictionary data structure, supported by a linked list implementation, to offer word suggestions as the user types. The goal is to create a system that dynamically generates and displays a list of k possible word completions based on a prefix (a partial sequence of letters) entered by the user.

1. Input Data Handling

- **File Input**: The system will read data from a file that contains prefixes and their associated words. This data will be parsed and used to populate the dictionary.
- **File Format**: The data is expected to be structured such that each prefix is followed by a list of words starting with that prefix.

2. Data Structure Design

- **Dictionary**: A dictionary will be used as the main data structure to store the prefixes and their associated words. The key in the dictionary will be the prefix, and the value will be a linked list of words that start with that prefix.
- **Linked List**: Each linked list will hold words starting with a particular prefix. This allows for efficient storage and retrieval of words and supports dynamic word addition.
- **Why Dictionary?**: A dictionary allows fast lookups by prefix, ensuring the autocomplete feature provides near-instant

suggestions. A linked list is chosen for storing words because it allows for easy insertion and deletion of words, making it suitable for dynamic vocabulary expansion.

3. Input Handling and Word Suggestions

- **User Input**: The system will take input from the user as they type a prefix (e.g., "abs"). As the user types each character, the system will match it against the stored prefixes in the dictionary.
- **Suggestion Generation**: Once the system identifies a matching prefix, it will return the first **k** words associated with that prefix. For example, if the prefix is "abs" and **k = 3**, the system may suggest "absent", "absorb", and "absurd".
- **Updating Suggestions**: As the user continues typing, the list of suggestions should update in real-time to reflect the newly entered prefix. For instance, after typing "abse", the system may suggest "absent", "absently", and "absenteeism".

4 .Implementation Details

1. Data Structures

a. WordList

- Stores a list of words associated with a prefix.
- Contains:
    - words: A 2D array to store words.
    - count: The number of words in the list.

```
typedef struct WordList {
    char words[MAX_WORDS][MAX_WORD_LENGTH];
    int count;
} WordList;
```

b. Node

- Represents a node in the linked list.
- Contains:
    - prefix: The prefix string.
    - wordList: A list of words starting with the prefix.
    - next: A pointer to the next node.

```
typedef struct Node {
    char prefix[MAX_WORD_LENGTH];
    WordList wordList;
    struct Node* next;
} Node;
```

2. Key Functions

a. createNode

- Creates a new node with the given prefix.
- Initializes the word list and sets the next pointer to NULL.

```
Node* createNode(char* prefix) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    strcpy(newNode->prefix, prefix);
    newNode->wordList.count = 0;
    newNode->next = NULL;
    return newNode;
}
```

b. addWordToNode

- Adds a word to the word list of a node.
- Ensures the word list does not exceed the maximum capacity.

```
void addWordToNode(Node* node, char* word) {
    if (node->wordList.count < MAX_WORDS) {
        strcpy(node->wordList.words[node->wordList.count], word);
        node->wordList.count++;
    }
}
```

c. insertNode

- Inserts a new node into the linked list or updates an existing node.

- If the prefix already exists, the word is added to the existing node's word list.
- If the prefix does not exist, a new node is created and inserted in the correct position.

```
void insertNode(Node** head, char* prefix, char* word) {
    Node* temp = *head;
    Node* prev = NULL;

    while (temp != NULL && strcmp(temp->prefix, prefix) < 0) {
        prev = temp;
        temp = temp->next;
    }

    if (temp != NULL && strcmp(temp->prefix, prefix) == 0) {
        addWordToNode(temp, word);
    } else {
        Node* newNode = createNode(prefix);
        addWordToNode(newNode, word);

        if (prev == NULL) {
            newNode->next = *head;
            *head = newNode;
        } else {
            newNode->next = prev->next;
            prev->next = newNode;
        }
    }
}
```

d. buildDictionary

- Reads data from a CSV file and builds the linked list.
- Tokenizes each line to extract prefixes and words.
- Inserts the data into the linked list using the insertNode function.

```c
void buildDictionary(Node** head, char* filename) {
    FILE* file = fopen(filename, "r");
    if (file == NULL) {
        printf("Error opening file.\n");
        return;
    }

    char line[256];
    while (fgets(line, sizeof(line), file)) {
        line[strcspn(line, "\n")] = '\0';

        char* token = strtok(line, ",");
        if (token == NULL) continue;

        char prefix[MAX_WORD_LENGTH];
        strcpy(prefix, token);

        token = strtok(NULL, ",");
        while (token != NULL) {
            insertNode(head, prefix, token);
            token = strtok(NULL, ",");
        }
    }

    fclose(file);
}
```

Main Loop: 3. Main Function

The main function drives the program. It:

1. Initializes the linked list.
2. Builds the dictionary from the CSV file.
3. Takes user input character by character.
4. Displays suggestions based on the current prefix.
5. Exits when the user types "exit".

```c
int main() {
    Node* head = NULL;
    char filename[] = "data.csv";
    buildDictionary(&head, filename);

    char input[MAX_WORD_LENGTH] = "";
    printf("Start typing (type 'exit' to quit):\n");

    while (1) {
        printf("Current input: %s\n", input);
        printf("Suggestions: ");

        WordList* suggestions = findWordsWithPrefix(head, input);
        if (suggestions != NULL) {
            for (int i = 0; i < suggestions->count; i++) {
                printf("%s ", suggestions->words[i]);
            }
        } else {
            printf("No suggestions found.");
        }
        printf("\n");

        char nextChar;
        printf("Enter next character: ");
        scanf(" %c", &nextChar);

        if (nextChar == '\n' || nextChar == '\0') continue;
        if (nextChar == 'e' && strcmp(input, "exi") == 0) break;

        strncat(input, &nextChar, 1);
    }

    printf("\nFinal Dictionary:\n");
    displayDictionary(head);

    return 0;
}
```

5.OUTPUT

```
┌──(katoki㉿Rapheal)-[~/project]
└─$ gcc reading.c

┌──(katoki㉿Rapheal)-[~/project]
└─$ ./a.out
-press \ for exiting-


b
ba
bal [ Suggestions: bal balancing balcony bale baleful ]
ball [ Suggestions: ball ballboy ballet ballroom ]
ballo
balloo
balloon
balloon
balloon i
balloon in
balloon in
balloon in t
balloon in th
balloon in the
balloon in the
balloon in the c
balloon in the ca
balloon in the can [ Suggestions: can canal canals candid candle ]
balloon in the cana [ Suggestions: canal canals ]
balloon in the canal [ Suggestions: canal canals ]
balloon in the canal
balloon in the canal l
balloon in the canal lo
balloon in the canal los
balloon in the canal lost

┌──(katoki㉿Rapheal)-[~/project]
└─$
```

6.Runtime Performance

Runtime Performance Analysis

The runtime performance of the autocomplete system is a critical factor, especially as the dataset grows. In this section, we analyze the performance of the current implementation using a **linked list** and discuss potential optimizations.

---

1. Performance of Linked List

a. Time Complexity

1. **Insertion**:
    - **Best Case**: O(1) (inserting at the head of the list).
    - **Worst Case**: O(n) (inserting at the end of the list or maintaining sorted order).
    - **Average Case**: O(n).
2. **Search**:
    - **Best Case**: O(1) (prefix matches the first node).
    - **Worst Case**: O(n) (prefix matches the last node or is not found).
    - **Average Case**: O(n).
3. **Traversal**:
    - **Time Complexity**: O(n) (traversing the entire list to display all prefixes and words).

b. Space Complexity

- **Space Complexity**: O(n * m), where:
    - n is the number of prefixes.

- m is the average number of words per prefix.

c. Performance Issues

- **Inefficient Search**:
  - Searching for a prefix requires traversing the entire list, leading to **O(n)** time complexity.
  - This becomes a bottleneck for large datasets.
- **No Prefix Optimization**:
  - Linked lists do not take advantage of shared prefixes among words.
  - Each word is stored independently, leading to redundant storage and increased memory usage.
- **Scalability**:
  - Performance degrades significantly as the dataset grows, making the system unsuitable for real-world applications.

---

2. Example Runtime Analysis

Dataset:

- Number of prefixes: 1,000.
- Average words per prefix: 10.
- Total words: 10,000.

Operations:

1. **Building the Dictionary**:
   - Time Complexity: O(n * m) = O(1,000 * 10) = O(10,000).
   - Explanation: Each prefix and its associated words are inserted into the linked list.

2. **Searching for a Prefix**:
    - Time Complexity: O(n) = O(1,000).
    - Explanation: The list is traversed to find the matching prefix.
3. **Displaying the Dictionary**:
    - Time Complexity: O(n * m) = O(1,000 * 10) = O(10,000).
    - Explanation: The entire list is traversed to display all prefixes and words.

7.Results and Analysis

The dictionary-based autocomplete system efficiently provides word suggestions by mapping prefixes to word lists, making it ideal for search engines and messaging platforms. It dynamically updates suggestions as the user types and supports vocabulary expansion. While effective for small to medium datasets, a trie structure could offer better performance for large vocabularies due to its efficient prefix-based searches. Despite this, the dictionary approach remains simple, scalable, and easy to implement.

8.Annexure

Annexure : we are taking file which contains the words as an input file.

Prefix	Word 1	Word 2	Word 3	Word 4	Word 5	Word 6	Word 7	Word 8	Word 9	Word 10	Word 11	Word 12	Word 13	Word 14
can	cane	cannot	candy	cans	candid	cannon	cannons	canal	canals	cantonme	candle			
dorm	dormitory	dorms	dormant	dormitories										
stand	stands	standup	standee	standings	standard	standish	standout	standby	standardize					
bal	balking	balcony	balm	ball	ballet	balk	balkiness	balancing	bale	ballboy	ballroom	baleful	bales	balmy

During the development of the autocomplete system, several key lessons were learned that provided valuable insights into data structures, algorithm efficiency, and user experience design.

1. **Importance of Data Structures:**
    - Linked lists offer a flexible way to store prefixes and words dynamically, but they can be less efficient in terms of search speed compared to other structures like Tries.
2. **File Handling and Data Management:**
    - Efficiently reading and storing dictionary data from a file is crucial for scalability.
    - Proper parsing and handling of input data ensures the integrity of stored information.
3. **Dynamic Memory Management:**
    - Allocating and deallocating memory properly prevents memory leaks, which are critical in linked list implementations.
    - Using malloc() and free() effectively ensures smooth performance without excessive memory usage.

4. **Real-Time User Input Processing:**
    - Handling user input dynamically requires efficient searching to maintain real-time performance.
    - The system should balance speed and accuracy when displaying word suggestions.
5. **Scalability Considerations:**
    - While linked lists work well for small datasets, **larger datasets require more optimized approaches** to avoid slow searches.
    - Implementing **indexed data structures** like Tries or Hash Tables can help in large-scale applications.
6. **Error Handling and Robustness:**
    - The system must handle **invalid inputs, file errors, and edge cases** (e.g., prefixes with no matching words).
    - Defensive programming techniques improve system reliability and prevent crashes.
7. **Usability and User Experience:**
    - Providing suggestions **as the user types** improves usability and makes text input faster.
    - A command-line-based implementation is simple but may lack the convenience of GUI-based selection.

Future Improvements

- Implementing a **Trie-based approach** for faster searches.
- Enhancing the **user interface** with a GUI for better interaction.
- Improving **search optimization** to handle larger datasets efficiently.

9. Conclusion

The execution of the autocomplete system using a dictionary has proven effective in enhancing text input speed, offering quick and relevant suggestions as users type. By mapping prefixes to corresponding word lists, the system provides dynamic suggestions and supports vocabulary expansion. The use of linked lists ensures flexibility, while limiting suggestions to a manageable number for a streamlined user experience.

While the dictionary approach works well for small to medium-sized datasets, alternative data structures like tries may offer better performance for handling larger vocabularies. In conclusion, the dictionary-based autocomplete system successfully meets the requirements for speed, accuracy, and scalability, providing a solid foundation for text input in terminal-based applications.

10.References

1.https://www.geeksforgeeks.org/implementation-on-map-or-dictionary-data-structure-in-c/

2.https://stackoverflow.com/questions/4384359/quick-way-to-implement-dictionary-in-c

3.https://www.reddit.com/r/learnprogramming/comments/n6y4qu/dictionary_in_c/?rdt=53079

4.chatgpt,openai.(2025)

- .