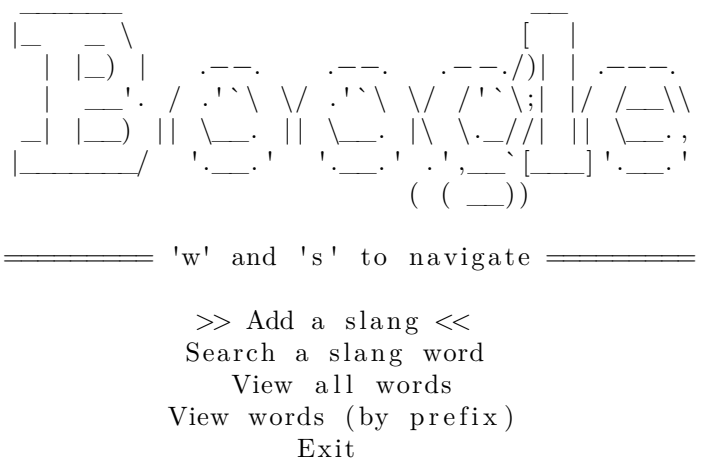
Data Structure AOL

1 Introduction

This C program implements a **Trie**, a tree-based data structure designed for efficient string storage and retrieval, commonly used in autocomplete, spell-checking, and dictionary applications. The program demonstrates core operations such as **insertion & search** while optimizing lookup performance through character-based node storage.

2 The Program

Boogle is a C program that serves as a slang dictionary, offering features to add, search, view, and search by prefix. It utilizes an efficient Trie data structure to simplify the management and retrieval of slang words, ensuring fast and effective access.



2.1 Main Menu

The **Boogle** program features a text-based interface, prominently displaying an ASCII art title. Users navigate the menu using the 'w' and 's' keys. The main menu presents five options:

1. Add a slang
2. Search a slang word
3. View all words
4. View words (by prefix)
5. Exit

The currently selected option is marked by '»' and '«' symbol for clear visibility.

```
void header(){

    puts(" ______                                __          ");
    puts("|_    _ \\                             [  |          ");
    puts("  | |_) |   .--.    .--.   .--./)| | .---.  ");
    puts("  |  __'. / .'`\\ \\/ .'`\\ \\/ /'`\\;| |/ /__\\\\ ");
    puts(" _| |__) || \\__. || \\__. |\\ \\._//| || \\__., ");
    puts("|_______/  '.__.'  '.__.' .',__`[___]'.__.' ");
    puts("                          ( ( __))           ");
    puts("");

}
```

```
void mainMenu1(){

    system("cls");
    header();
    puts("========= 'w' and 's' to navigate =========");
    puts("");
    puts("            >> Add a slang <<         ");
    puts("           Search a slang word             ");
    puts("              View all words               ");
    puts("          View words (by prefix)       ");
    puts("                  Exit                  ");

    char ch = getch();
    while(1){
        switch(ch){
            case 'w' : mainMenu5(); break;
            case 's' : mainMenu2(); break;
            case '\r' : addWord(); break;
            case '\n' : addWord(); break;
```

```
20              default : addWord(); break;
21          }
22      }
23
24  }
```

Explanation:
This function is responsible for displaying the first menu screen of the program. The menu guides the user to use the ‘’w‘‘ and ‘’s‘‘ keys to navigate between different menu options. In this screen, the currently selected option is "‘Add a slang‘", and if the user presses Enter, it directly leads to the ‘addWord()‘ function. Pressing ‘’s‘‘ navigates to the next menu (‘mainMenu2()‘), while pressing ‘’w‘‘ loops around to the last menu (‘mainMenu5()‘).

Output

```
 _______                                  __
|_    _ \                                [  |
  | |_) |   .--.     .--.    .--./)| | .---.
  |  __'. / .'`\ \/ .'`\ \/ /'`\;| |/ /__\\
 _| |__) || \__. || \__. |\ \._//| || \__.,
|_______/  '.__.'  '.__.' .',__`[___]'.__.'
                          ( ( __))

========= 'w' and 's' to navigate =========

            >> Add a slang <<
           Search a slang word
              View all words
          View words (by prefix)
                  Exit
```

2.2 Trie Structure

The Trie data structure is a tree-like structure that is used for efficient storage and retrieval of strings. Each node represents a character of a word, and the path from the root to any node forms a prefix of the word it represents.

```
struct Node {

    char description[100];
    struct Node *children[26];
    bool isEndOfWord;

};
```

Explanation:
- The `struct Node` defines a node in the Trie.
- `description[100]`: This array holds a description of the word associated with the node. Each node corresponds to a character in a word and may have an associated description.
- `children[26]`: This array holds pointers to the child nodes. Since the Trie is designed for lowercase English letters, there are 26 children corresponding to the 26 letters of the alphabet ('a' to 'z').
- `isEndOfWord`: A boolean flag that marks whether the current node represents the end of a valid word in the Trie. If `isEndOfWord` is true, the node marks the end of a word.

```
struct Node *createNode() {

    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->isEndOfWord = false;
    strcpy(newNode->description, "");
    for (int i = 0; i < 26; i++) {
        newNode->children[i] = NULL;
    }
    return newNode;

}
```

Explanation:
- The `createNode` function creates a new node for the Trie.
- `struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));` dynamically allocates memory for a new node of type `struct Node`.
- `newNode->isEndOfWord = false;` initializes the `isEndOfWord` flag to false, as this is not the end of a word when the node is created.
- `strcpy(newNode->description, "");` sets the description of the new node to an empty string.
- The `for` loop initializes all the child nodes of the new node to `NULL`, indicating that no children are connected at the time of creation.
- Finally, the function returns the newly created node.

```
1 struct Node *root = createNode();
```

Explanation:
- `struct Node *root = createNode();` creates the root node of the Trie by calling `createNode`. The root node does not represent any word but serves as the starting point for all insertions and searches.

2.3 Add a Slang

```c
void insert(struct Node *root, const char *word){

    struct Node *current = root;
    for(int i = 0; word[i] != '\0'; i++){
        int index = word[i] - 'a';
        if(current->children[index] == NULL){
            current->children[index] = createNode();
        }
        current = current->children[index];
    }
    current->isEndOfWord = true;
    strcpy(current->description, word);

}

void addWord() {

    system("cls");
    header();
    printf("=============== Add a word ================\n\n");
    char word[100];
    char description[256];
    while (1) {
        printf("Enter the word [> 1 characters & no spacing]: ");
        fgets(word, sizeof(word), stdin);
        word[strcspn(word, "\n")] = '\0';
        int valid = 1;
        if (strlen(word) <= 1) {
            printf("Error: Word must be more than 1 character.\n");
            continue;
        }
        for (int i = 0; word[i]; i++) {
            if (!islower(word[i]) || isspace(word[i])) {
                valid = 0;
                break;
            }
        }
        if (!valid) {
            printf("Error: Word must contain only lowercase letters (a-z) and no
            ↪  spaces.\n");
            continue;
        }
        break;
    }
    while (1) {
        printf("Enter the description [> 2 words]: ");
        fgets(description, sizeof(description), stdin);
        description[strcspn(description, "\n")] = '\0';
```

```
48          int count = 0, inWord = 0;
49          for (int i = 0; description[i]; i++) {
50              if (!isspace(description[i]) && !inWord) {
51                  count++;
52                  inWord = 1;
53              } else if (isspace(description[i])) {
54                  inWord = 0;
55              }
56          }
57          if (count < 3) {
58              printf("Error: Description must contain more than 2 words.\n");
59          } else {
60              break;
61          }
62      }
63      struct Node *current = root;
64      for (int i = 0; word[i]; i++) {
65          int index = word[i] - 'a';
66          if (index < 0 || index >= 26) continue;
67          if (current->children[index] == NULL) {
68              current->children[index] = createNode();
69          }
70          current = current->children[index];
71      }
72      current->isEndOfWord = true;
73      strcpy(current->description, description);
74      printf("Word '%s' added successfully!\n", word);
75      printf("\n       >> Press Enter to continue <<");
76      while (1) {
77          char ch = getch();
78          if (ch == '\r') {
79              mainMenu1();
80              break;
81          }
82      }
83
84  }
```

Explanation:

- The `insert` function is used to insert a word into the Trie. It takes the root node of the Trie and the word to be inserted as parameters.
- `struct Node *current = root;` sets the current node to the root of the Trie, and the traversal of the Trie starts from there.
- The `for` loop iterates through each character of the word:
- `int index = word[i] - 'a';` calculates the index for the character `word[i]` by subtracting the ASCII value of 'a'. This gives an index between 0 and 25, corresponding to each letter of the alphabet.
- `if (current->children[index] == NULL)` checks if the child node corresponding to the current character exists. If it doesn't, a new node is created for that character.

- `current = current->children[index];` moves the `current` pointer to the newly created or existing child node.
- After the loop completes (i.e., after the last character of the word has been processed), the `current->isEndOfWord = true;` marks the current node as the end of a word.
- `strcpy(current->description, word);` copies the word into the `description` field of the node to store the word's description.

The `addWord` function allows users to add a new word and its description to the slang dictionary. This function performs input validation to ensure the word and description meet certain criteria before storing them in a Trie structure. Below is a detailed explanation of the code:

```
void addWord() {

    system("cls");
    header();
    printf("================ Add a word ================\n\n");
```

Explanation:
- `system("cls");` clears the screen.
- `header();` prints the header (as shown in the Main Menu section).
- `printf("================ Add a word ================");` prints a message indicating the beginning of the "Add a word" process.

```
    char word[100];
    char description[256];
```

Explanation:
- Two character arrays `word` and `description` are declared. `word` stores the slang word (max 100 characters), and `description` stores the word's description (max 256 characters).

```
    while (1) {
        printf("Enter the word [> 1 characters & no spacing]: ");
        fgets(word, sizeof(word), stdin);
        word[strcspn(word, "\n")] = '\0';
        int valid = 1;
```

Explanation:
- A `while(1)` loop is used to repeatedly ask for the word until it is valid.
- `fgets(word, sizeof(word), stdin);` reads the word input from the user.
- `word[strcspn(word, "")] = ";` removes the newline character if it is included by `fgets`.

```
        if (strlen(word) <= 1) {
            printf("Error: Word must be more than 1 character.\n");
            continue;
        }
```

Explanation:
- If the length of the word is less than or equal to 1, an error message is displayed, and the loop continues, prompting the user to enter a valid word.

```
        for (int i = 0; word[i]; i++) {
            if (!islower(word[i]) || isspace(word[i])) {
                valid = 0;
                break;
            }
        }
        if (!valid) {
            printf("Error: Word must contain only lowercase letters (a-z) and no
            ↪  spaces.\n");
            continue;
        }
        break;
    }
```

Explanation:
- The `for` loop iterates through each character in the word.
- If a character is not a lowercase letter or is a space, the word is marked as invalid.
- If the word is invalid, an error message is shown, and the loop continues.
- If the word is valid, the loop breaks, allowing the function to proceed.

```
    while (1) {
        printf("Enter the description [> 2 words]: ");
        fgets(description, sizeof(description), stdin);
        description[strcspn(description, "\n")] = '\0';
        int count = 0, inWord = 0;
```

Explanation:
- Another `while(1)` loop is used to prompt the user for a description.
- `fgets(description, sizeof(description), stdin);` reads the description.
- `description[strcspn(description, "")] = ";` removes the newline character from the description.

```
        for (int i = 0; description[i]; i++) {
            if (!isspace(description[i]) && !inWord) {
                count++;
                inWord = 1;
            } else if (isspace(description[i])) {
                inWord = 0;
            }
        }
        if (count < 3) {
            printf("Error: Description must contain more than 2 words.\n");
```

```
        } else {
            break;
        }
    }
```

Explanation:
- The `for` loop counts the number of words in the description. A word is considered as a sequence of characters that are not spaces.
- If there are fewer than 3 words in the description, an error message is shown. If the description is valid, the loop breaks.

```
    struct Node *current = root;
    for (int i = 0; word[i]; i++) {
        int index = word[i] - 'a';
        if (index < 0 || index >= 26) continue;
        if (current->children[index] == NULL) {
            current->children[index] = createNode();
        }
        current = current->children[index];
    }
    current->isEndOfWord = true;
    strcpy(current->description, description);
    printf("Word '%s' added successfully!\n", word);
    printf("\n       >> Press Enter to continue <<");
```

Explanation:
- A `Node *current = root;` initializes a pointer `current` to the root of the Trie.
- A `for` loop iterates over each character in the word, creating new nodes in the Trie as necessary.
- `current->isEndOfWord = true;` marks the end of the word in the Trie.
- `strcpy(current->description, description);` stores the description for the word.
- A success message is displayed, and the user is instructed to press Enter to continue.

```
    while (1) {
        char ch = getch();
        if (ch == '\r') {
            mainMenu1();
            break;
        }
    }

}
```

Explanation:
- The `while(1)` loop waits for the user to press Enter (represented by ˚).
- When Enter is pressed, `mainMenu1();` is called, which likely returns to the main menu, and the loop ends.

Output

```
 _______                                  __
|_    _ \                                [  |
  | |_) |   .--.     .--.    .--./)| | .---.
  |  __'. / .'`\ \/ .'`\ \/ /'`\;| |/ /__\\
 _| |__) || \__. || \__. |\ \._//| || \__.,
|_______/  '.__.'  '.__.' .',__`[___]'.__.'
                          ( ( __))

================ Add a word ================

Enter the word [> 1 characters & no spacing]: brain rot
Error: Word must contain only lowercase letters (a-z) and no spaces.
Enter the word [> 1 characters & no spacing]: b
Error: Word must be more than 1 character.
Enter the word [> 1 characters & no spacing]: brainrot
Enter the description [> 2 words]: non educational
Error: Description must contain more than 2 words.
Enter the description [> 2 words]: non educational content
Word 'brainrot' added successfully!

       >> Press Enter to continue <<
```

2.4 Search a Slang Word

```c
void searchWord() {

    system("cls");
    header();
    printf("============== Search a Word ==============\n\n");
    char word[100];
    while (1) {
        printf("Enter the word [> 1 characters & no spacing]: ");
        fgets(word, sizeof(word), stdin);
        word[strcspn(word, "\n")] = '\0';
        int valid = 1;
        if (strlen(word) <= 1) {
            printf("Error: Word must be more than 1 character.\n");
            continue;
        }
        for (int i = 0; word[i]; i++) {
            if (!islower(word[i]) || isspace(word[i])) {
                valid = 0;
                break;
            }
        }
        if (!valid) {
            printf("Error: Word must contain only lowercase letters (a-z) and no
            ↪  spaces.\n");
            continue;
        }
        break;
    }
    struct Node *current = root;
    for (int i = 0; word[i]; i++) {
        int index = word[i] - 'a';
        if (current->children[index] == NULL) {
            current = NULL;
            break;
        }
        current = current->children[index];
    }

    if (current != NULL && current->isEndOfWord) {
        printf("\nWord found!\n");
        printf("Word        : %s\n", word);
        printf("Description : %s\n", current->description);
    } else {
        printf("\nWord '%s' not found in dictionary.\n", word);
    }
    printf("\n       >> Enter to continue <<");
    char ch = getch();
    while (1) {
```

```
48          if (ch == '\r' || ch == '\n') {
49              mainMenu2();
50              break;
51          }
52      }
53
54  }
```

Explanation:
The `searchWord()` function allows the user to search for a slang word in a trie-based dictionary. It prompts the user to input a valid lowercase word without spaces, validates the input, then traverses the trie data structure to check if the word exists. If found, it displays the word along with its description; otherwise, it informs the user that the word is not in the dictionary. Finally, it waits for the user to press Enter before returning to the main menu.

```
1   void searchWord() {
2
3       system("cls");
4       header();
5       printf("============== Search a Word ==============\n\n");
```

Explanation:
- Clears the console screen (Windows specific) with `system("cls")`.
- Calls `header()` to print the program's header or title.
- Prints a section title indicating the start of the "Search a Word" feature.

```
1       char word[100];
2       while (1) {
3           printf("Enter the word [> 1 characters & no spacing]: ");
4           fgets(word, sizeof(word), stdin);
5           word[strcspn(word, "\n")] = '\0';
```

Explanation:
- Declares a character array `word` to hold the user's input (max 99 chars + null).
- Starts an infinite loop to repeatedly prompt until valid input is given.
- Prompts user to enter a word longer than one character with no spaces.
- Reads the input line safely with `fgets` from `stdin`.
- Removes the trailing newline from the input by replacing it with a null terminator.

```
1           int valid = 1;
2           if (strlen(word) <= 1) {
3               printf("Error: Word must be more than 1 character.\n");
4               continue;
5           }
```

Explanation:
- Initializes a flag `valid` to track input validity.
- Checks if the entered word length is 1 or less.
- If too short, prints an error and restarts the input loop.

```
        for (int i = 0; word[i]; i++) {
            if (!islower(word[i]) || isspace(word[i])) {
                valid = 0;
                break;
            }
        }
        if (!valid) {
            printf("Error: Word must contain only lowercase letters (a-z) and no
            ↪  spaces.\n");
            continue;
        }
        break;
    }
```

Explanation:
- Loops through each character in the input word.
- Checks if any character is not a lowercase letter or is a space.
- If invalid character found, sets `valid` to 0 and breaks the loop.
- If invalid, prints an error message and restarts input loop.
- If valid, breaks out of the input loop to continue.

```
    struct Node *current = root;
    for (int i = 0; word[i]; i++) {
        int index = word[i] - 'a';
        if (current->children[index] == NULL) {
            current = NULL;
            break;
        }
        current = current->children[index];
    }
```

Explanation:
- Sets a pointer `current` to the root of the trie (dictionary).
- Iterates over each character in the word.
- Calculates the index for the trie children array by subtracting 'a'.
- Checks if the child node exists for this character; if not, sets `current` to `NULL` and breaks (word not found).
- Otherwise, moves `current` to that child node.

```c
    if (current != NULL && current->isEndOfWord) {
        printf("\nWord found!\n");
        printf("Word        : %s\n", word);
        printf("Description : %s\n", current->description);
    } else {
        printf("\nWord '%s' not found in dictionary.\n", word);
    }
```

Explanation:
- Checks if `current` is not `NULL` and marks the end of a valid word.
- If so, prints that the word was found, showing the word and its description.
- Otherwise, prints that the word was not found in the dictionary.

```c
    printf("\n       >> Enter to continue <<");
    char ch = getch();
    while (1) {
        if (ch == '\r' || ch == '\n') {
            mainMenu2();
            break;
        }
    }

}
```

Explanation:
- Prompts user to press Enter to continue.
- Uses `getch()` to wait for a key press without echoing it.
- Waits in a loop until the Enter key (carriage return or newline) is pressed.
- Once Enter is detected, calls `mainMenu2()` to return to the main menu and exits the function.

Output

```
 _______                                  __
|_    _ \                                [  |
  | |_) |   .--.     .--.    .--./)| | .---.
  |  __'. / .'`\ \/ .'`\ \/ /'`\;| |/ /__\\
 _| |__) || \__. || \__. |\ \._//| || \__.,
|_______/  '.__.'  '.__.' .',__`[___]'.__.'
                          ( ( __))

========== Search a Word ==========

Enter the word [> 1 characters & no spacing]: brainrot

Word found!
Word        : brainrot
Description : non educational content

       >> Enter to continue <<
```

2.5 View All Words

```c
int count = 1;

void traverse(struct Node *current, char *buffer, int depth){

    if (current->isEndOfWord) {
        buffer[depth] = '\0';
        printf("%-4d %-20s %-50s\n", count++, buffer, current->description);
    }

    for (int i = 0; i < 26; i++) {
        if (current->children[i]) {
            buffer[depth] = 'a' + i;
            traverse(current->children[i], buffer, depth + 1);
        }
    }

}

int isTrieEmpty(struct Node *root){

    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            return 0;
        }
    }
    return 1;

}

void viewWord(){

    system("cls");
    header();
    puts("============= View all words ==============\n");

    if (isTrieEmpty(root)) {
        puts("Dictionary is empty.\n");
    } else {
        puts("No.  Word\t\t  Description\n");
        char buffer[100];
        count = 1;
        traverse(root, buffer, 0);
    }
    printf("\n       >> Press Enter to continue <<");
    while (1) {
        char ch = getch();
        if (ch == '\r') {
            mainMenu3();
```

```
            break;
        }
    }

}
```

Explanation:
The `viewWord()` function displays all words stored in a trie dictionary. If the trie is empty, it notifies the user. Otherwise, it traverses the trie recursively using the `traverse()` function, printing each word and its description in a formatted list. After listing all entries, it waits for the user to press Enter before returning to the main menu.

```
int count = 1;
```

Explanation:
- A global counter used to number each word displayed in the output.

```
void traverse(struct Node *current, char *buffer, int depth){
```

Explanation:
- Defines a recursive function to traverse the trie.
- Parameters: current node, character buffer to build words, and current depth (index in the buffer).

```
    if (current->isEndOfWord) {
        buffer[depth] = '\0';
        printf("%-4d %-20s %-50s\n", count++, buffer, current->description);
    }
```

Explanation:
- Checks if the current node marks the end of a word.
- Adds null terminator to complete the string in `buffer`.
- Prints the word number, word, and description in formatted columns.
- Increments the counter.

```
    for (int i = 0; i < 26; i++) {
        if (current->children[i]) {
            buffer[depth] = 'a' + i;
            traverse(current->children[i], buffer, depth + 1);
        }
    }
```

Explanation:
- Iterates through each child (a–z) of the current node.
- If a child exists, appends the corresponding character to the buffer.
- Recursively calls `traverse()` to continue building the word.

```
int isTrieEmpty(struct Node *root){
```

Explanation:
- Checks whether the trie is empty by examining all 26 children of the root node.

```
    for (int i = 0; i < 26; i++) {
        if (root->children[i]) {
            return 0;
        }
    }
    return 1;
```

Explanation:
- If any child exists, the trie is not empty (returns 0).
- If no children are found, returns 1 (empty).

```
void viewWord(){
```

Explanation:
- Main function to display all words in the dictionary.

```
    system("cls");
    header();
    puts("============== View all words ==============\n");
```

Explanation:
- Clears the console screen.
- Prints the header and section title for viewing all words.

```
    if (isTrieEmpty(root)) {
        puts("Dictionary is empty.\n");
    } else {
        puts("No.  Word\t\t  Description\n");
        char buffer[100];
        count = 1;
        traverse(root, buffer, 0);
    }
```

Explanation:
- If the trie is empty, displays a message indicating so.
- Otherwise, prints table headers.
- Initializes the buffer and resets counter.
- Calls `traverse()` to print all words and descriptions.

```
    printf("\n       >> Press Enter to continue <<");
    while (1) {
        char ch = getch();
        if (ch == '\r') {
            mainMenu3();
            break;
        }
    }

}
```

Explanation:
- Prompts the user to press Enter to continue.
- Uses `getch()` to wait for key press.
- If Enter key is pressed, calls `mainMenu3()` to return to the menu and exits the function.

Output

```
 _______                                  __
|_    _ \                                [  |
  | |_) |   .--.     .--.    .--./)| | .---.
  |  __'. / .'`\ \/ .'`\ \/ /'`\;| |/ /__\\
 _| |__) || \__. || \__. |\ \._//| || \__.,
|_______/  '.__.'  '.__.' .',__`[___]'.__.'
                          ( ( __))

========== View all words ==========

No.  Word                 Description

1    bloop                a giant whale
2    brah                 the word bro
3    braingrowth          educational form content
4    brainrot             non educational content
5    bromine              bro mine chemistry
6    dawg                 the word dog
7    fr                   for real real
8    pmo                  piss me off

      >> Press Enter to continue <<
```

2.6 View Words (by Prefix)

```c
void displayWordsByPrefix(struct Node* current, char* prefix, int depth, int*
    counter){

    if (current->isEndOfWord) {
        printf("%d. %s\n", ++(*counter), prefix);
    }

    for (int i = 0; i < 26; i++) {
        if (current->children[i] != NULL) {
            char nextChar = 'a' + i;
            char newPrefix[100];
            strcpy(newPrefix, prefix);
            newPrefix[depth] = nextChar;
            newPrefix[depth + 1] = '\0';
            displayWordsByPrefix(current->children[i], newPrefix, depth + 1,
                counter);
        }
    }

}

void viewWordByPrefix(){

    char prefix[100] = "";
    int i = 0;
    while (1) {
        system("cls");
        header();
        printf("========= View a word (by prefix) =========\n");
        printf(">> %s\n\n", prefix);
        if (i > 0) {
            struct Node* current = root;
            int found = 1;

            for (int j = 0; j < i; j++) {
                int index = prefix[j] - 'a';
                if (current->children[index] == NULL) {
                    found = 0;
                    break;
                }
                current = current->children[index];
            }

            if (found) {
                int counter = 0;
                displayWordsByPrefix(current, prefix, i, &counter);
            } else {
                printf("\nNo words found with the prefix '%s'.\n", prefix);
```

```
            }
        }
        char ch = _getch();
        if (ch == '\r' || ch == '\n') {
            break;
        } else if (ch == 8) {
            if (i > 0) {
                i--;
                prefix[i] = '\0';
            }
        } else if (isalpha(ch)) {
            prefix[i++] = ch;
            prefix[i] = '\0';
        }
    }
    printf("\n       >> Press Enter to continue <<");
    while (_kbhit()) _getch();
    while (1) {
        char ch = getch();
        if (ch == '\r') {
            mainMenu4();
            break;
        }
    }
    _getch();

}
```

Explanation:
This feature allows users to interactively type a prefix and display all the words in the Trie that start with that prefix. As users type, the program navigates the Trie accordingly and lists matching words. If no matches are found, a corresponding message is shown. It supports real-time user input, backspace handling, and displays results neatly.

```
void displayWordsByPrefix(struct Node* current, char* prefix, int depth, int*
↪  counter){

    if (current->isEndOfWord) {
        printf("%d. %s\n", ++(*counter), prefix);
    }

    for (int i = 0; i < 26; i++) {
        if (current->children[i] != NULL) {
            char nextChar = 'a' + i;
            char newPrefix[100];
            strcpy(newPrefix, prefix);
            newPrefix[depth] = nextChar;
            newPrefix[depth + 1] = '\0';
```

```
            displayWordsByPrefix(current->children[i], newPrefix, depth + 1,
            ↪  counter);
        }
    }

}
```

Explanation:
- `displayWordsByPrefix()` recursively displays all words in the trie that begin with a given prefix.
- If the current node is the end of a word, the full prefix is printed with numbering.
- The function continues exploring all valid child nodes, appending characters to the current prefix and calling itself recursively.

```
void viewWordByPrefix(){
```

Explanation:
- This function handles the user interface for viewing words by a typed prefix.
- It continuously reads input from the user and updates the display with matching words.

```
    char prefix[100] = "";
    int i = 0;
```

Explanation:
- `prefix` stores the current prefix typed by the user.
- `i` tracks the length of the prefix string.

```
    while (1) {
        system("cls");
        header();
        printf("========= View a word (by prefix) =========\n");
        printf(">> %s\n\n", prefix);
```

Explanation:
- Enters a loop where the screen is cleared and the current prefix is displayed as the user types.

```
        if (i > 0) {
            struct Node* current = root;
            int found = 1;

            for (int j = 0; j < i; j++) {
```

```
                int index = prefix[j] - 'a';
                if (current->children[index] == NULL) {
                    found = 0;
                    break;
                }
                current = current->children[index];
            }
```

Explanation:
- If a prefix has been typed, the code searches for that prefix in the trie.
- If any character does not match a child node, the prefix is not found.

```
            if (found) {
                int counter = 0;
                displayWordsByPrefix(current, prefix, i, &counter);
            } else {
                printf("\nNo words found with the prefix '%s'.\n", prefix);
            }
        }
```

Explanation:
- If the prefix is found in the trie, `displayWordsByPrefix()` is called to display matching words.
- Otherwise, a message indicates no matches found.

```
        char ch = _getch();
        if (ch == '\r' || ch == '\n') {
            break;
        } else if (ch == 8) {
            if (i > 0) {
                i--;
                prefix[i] = '\0';
            }
        } else if (isalpha(ch)) {
            prefix[i++] = ch;
            prefix[i] = '\0';
        }
    }
```

Explanation:
- Reads a character from the user.
- If Enter is pressed, exits the loop.
- If Backspace is pressed, removes the last character.
- If an alphabet character is typed, it is added to the prefix.

```
    printf("\n        >> Press Enter to continue <<");
    while (_kbhit()) _getch();
    while (1) {
        char ch = getch();
        if (ch == '\r') {
            mainMenu4();
            break;
        }
    }
    _getch();
}
```

Explanation:
- After exiting the typing loop, prompts the user to press Enter.
- Flushes any extra key presses.
- Waits for Enter to return to the main menu via `mainMenu4()`.

Output

```
 _______                                  __
|_    _ \                                [  |
  | |_) |   .--.     .--.    .--./)| | .---.
  |  __'. / .'`\ \/ .'`\ \/ /'`\;| |/ /__\\
 _| |__) || \__. || \__. |\ \._//| || \__.,
|_______/  '.__.'  '.__.' .',__`[___]'.__.'
                         ( ( __))

========== View a word (by prefix) ==========
>> br

1. brah
2. braingrowth
3. brainrot
4. bromine

       >> Press Enter to continue <<
```

2.7 Exit

```c
void exitProgram(){

    system("cls");
    header();
    puts("          Exiting the program...");
    Sleep(2000);
    exit(0);

}
```

Explanation:
- `exitProgram()` is called when the user chooses to exit the application.
- `system("cls")` clears the screen.
- `header()` displays the program header (likely a title or banner).
- `puts()` shows the exit message.
- `Sleep(2000)` pauses the program for 2 seconds (2000 milliseconds) to give the user time to read the message.
- `exit(0)` terminates the program with a success status code.

Output