Matthew Poncini
J.B. Speed School of Engineering
University of Louisville

17 June 2024

—

Project 1: Trie Data Structure for Automatic Search Completion

Description of the Assignment

The goal of this assignment was to create a program that features automatic search completion using a Trie data structure. Part I of the project focuses on the creation of the Trie Data Structure. It closely models after the Binary Search Tree that was created in Assignment 3. Part II of the project shows the implementation of the Trie Structure in C++ program. Part III of the project is to allow for input editing so that the typos can be fixed.

Part I: The Trie Structure

TrieNode Construction

Each trienode stores a string value and has pointers to 27 child trienodes. Each trienode corresponds to one of the 26 letters of the alphabet with the last trienode representing a space. The constructor initializes the trienode's value and sets all child pointers to `nullptr`. The abc method returns the string value of the trienode.

TrieNode Construction

```
class TrieNode {
public:
    string value;
    const vector<string> dictionary = get_Dictionary();
    const vector<char> alphabet = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
        'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', ' '};

    vector<TrieNode*> abc_pointers;

    TrieNode(const string& val) : value(val), abc_pointers(27, nullptr) {}

    string abc() const {
        return value;
    }
    //... TrieNode class continues
```

Input Check and Word Check

The input_check function checks whether a given input string, insert, is a prefix of any word in dictionary.txt. The parameter of input_check takes is constant reference to a string and the function returns a boolean value. The function iterates over each word in the dictionary and checks if the prefix of a word matches the insert. If a match is found, the for loop breaks and the function returns true. This indicates that insert is a valid prefix. If no match is found after checking all words, the function returns false. This is important in the Trie Structure because it prevents the insert function from creating unwanted child trienodes.

The word_check function returns true if the input string, insert, matches any word in the dictionary. The word_check function is very similar to input_check. The only difference is that it checks the entire word. If the insert is a valid word, then the insert function can suggest this word to the user.

Insert Function

```
bool input_check(const string& insert) const {
    for (const auto& word : dictionary) {
        if (insert == word.substr(0, insert.size())) {
            return true;
        }
    }
    return false;
}

bool word_check(const string& insert) const {
    for (const auto& word : dictionary) {
        if (insert == word) {
            return true;
        }
    }
    return false;
}
    //... TrieNode class continues
```

The Insert Function

The insert function is adds trienodes to the Trie structure and collects suggested words for auto-completion based on the input. The insert function has three parameters: a pointer to a TrieNode (insert_node), a reference to a vector of strings (suggested_words), and an integer (depth). The depth integer is used to limit the amount of iterations that the insert function completes. This way, less suggestions will be outputted and the function will be faster.

The function first checks if the size of the insert_node's value, obtained from the abc method, is less than the specified depth, ensuring the recursive insertion process does not exceed the desired depth. If the depth condition is satisfied, the function uses the input_check method to determine if the current trienode's string value is a valid prefix of any word in the dictionary. If it is not, the function returns and stops further recursion for this branch.

If the current trienode's value is a valid prefix, the function will then use the word_check method to see if it is a complete and valid word in the dictionary. If it is, the function adds this word to the suggested_words vector. The function then iterates over the 27 possible child trienodes, each representing the letters of the alphabet (or a space). For each possible child trienode, it constructs a new string (new_string) by adding the current character from alphabet to the trienode's string value. It then checks if this new string is a valid prefix by using input_check. If it is, it creates a new TrieNode with new_string as its value and assigns it to the appropriate position in the abc_pointers vector of the current trienode. The function then recursively calls itself with the new trienode.

This recursive process ensures that all valid auto-completion suggestions up to the specified depth are collected in the suggested_words vector. The insert function leverages the Trie's structure to efficiently explore all potential completions and uses the dictionary to validate prefixes and complete words at each step.

The Insert Function

```
void insert(TrieNode* insert_node, vector<string>& suggested_words, int depth) {
    if (insert_node->abc().size() < depth) {
        if (input_check(insert_node->abc())) {
            if (word_check(insert_node->abc())) {
                suggested_words.push_back(insert_node->abc());
            }
            for (int i = 0; i < 27; i++) {
                string new_string = insert_node->abc() + alphabet[i];
                if (input_check(new_string)) {
                    insert_node->abc_pointers[i] = new TrieNode(new_string);
                    insert(insert_node->abc_pointers[i], suggested_words, depth);
                }
            }
        }
    }
}
// ... TrieNode class continues
```

Search History

The sh_input_check, sh_word_check, and sh_insert functions are necessary to separate the auto completion process for previously searched words. These functions mirror the general input_check, word_check, and insert functions but us vector that contains previously searched words instead of the dictionary vector.

The primary purpose of these functions is to offer personalized search suggestions by tailoring the auto-completion to the user's past searches. The sh_insert function parallels the main Trie insert function but is scoped to user data and does not have a depth limit. This enhances the relevance and personalising of the suggestions.

Search History Capability

```
void sh_insert(TrieNode* insert_node, vector<string>& suggested_words,
vector<string> user_search_history) {
    if (sh_input_check(insert_node->abc(), user_search_history)) {
        if (sh_word_check(insert_node->abc(), user_search_history)) {
            suggested_words.push_back(insert_node->abc());
        }
        for (int i = 0; i < 27; ++i) {
            string new_string = insert_node->abc() + alphabet[i];
            if (sh_input_check(new_string, user_search_history)) {
                insert_node->abc_pointers[i] = new TrieNode(new_string);
                sh_insert(insert_node->abc_pointers[i], suggested_words, user_search_history);
            }
        }
    }
}
// ... TrieNode class continues
```

Implementation

The main function implements a command-line interface for search auto-completion using a Trie structure. It begins by initializing a root Trie node and necessary variables to handle user input and vectors to store data. A predefined search_history vector is also initialized to show the program's capabilities. A while loop is controlled by the menu_on boolean. The program presents a menu with options to search or exit. This allows the user to turn off the while loop.

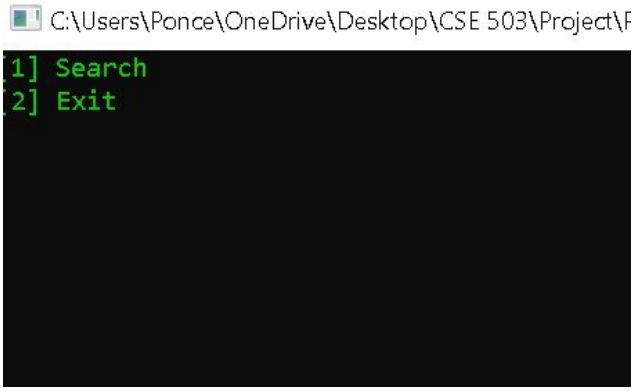

Figure 1: Main Menu

If the user chooses to search, they are prompted to input a query. The input is processed and trimmed so that the sh_insert and insert functions can handle them. The sh_insert function is called to generate suggestions based on the user's search history. After, the insert function is called to generate suggestions from the dictionary.

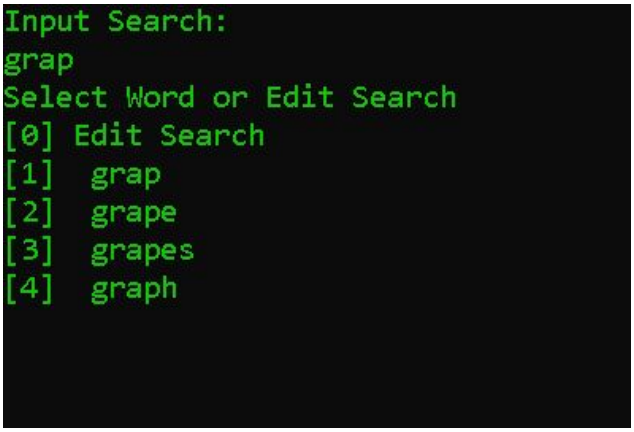

Figure 2: Search Auto Completion Example

The user is then presented with a list of auto-completed options as well as their original search query. Depending on their selection, the chosen query is added to the search_history vector. The loop continues until the user opts to exit. When this happens, the root Trie node is deleted, and the search history is printed out, showing all the searches made during the session.

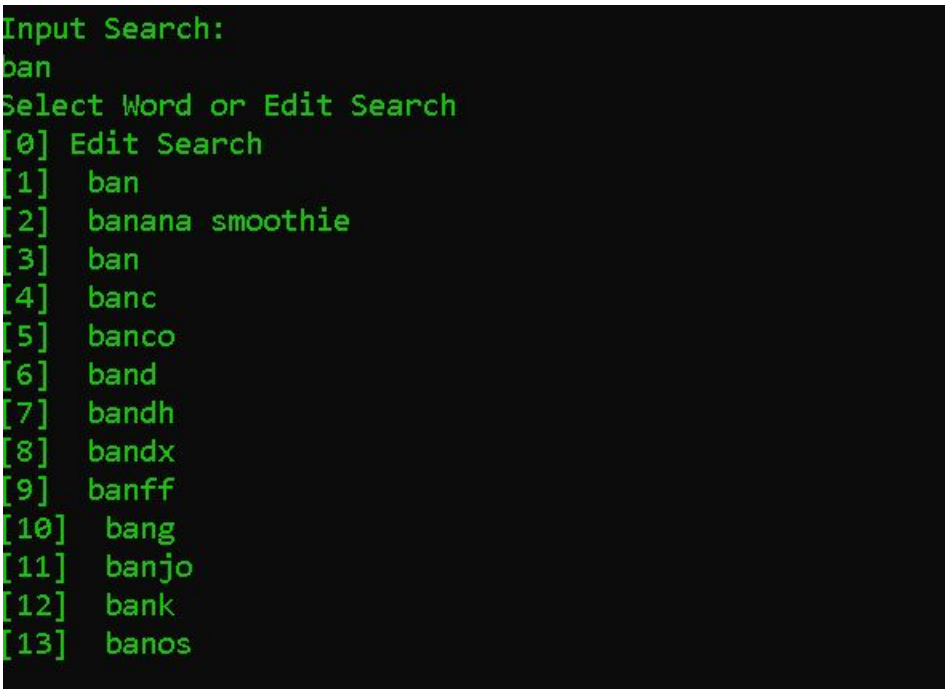

Figure 4: Searched Word Prioritization

The main function always suggests previously searched entries first. It is capable of searching entries that contain multiple words, or fragments of words split by a space.

Sample Outputs

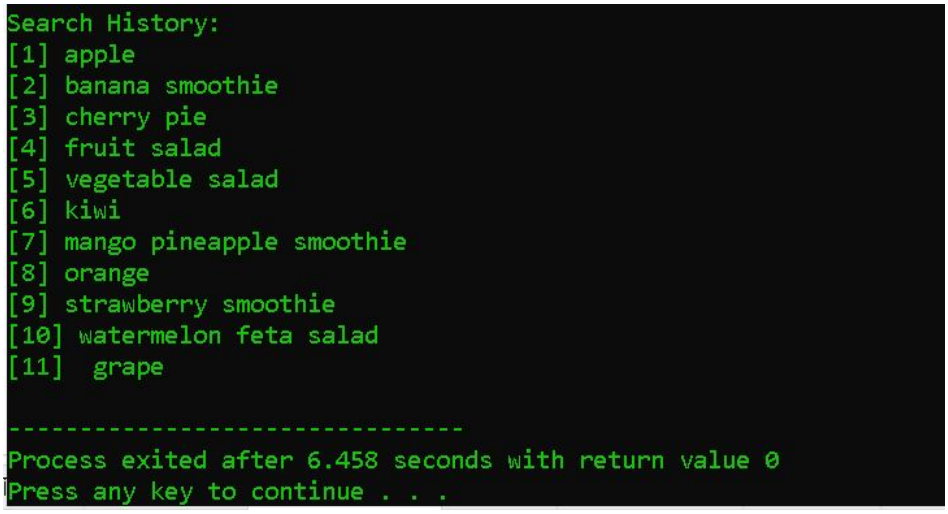

Figure 3: Search History

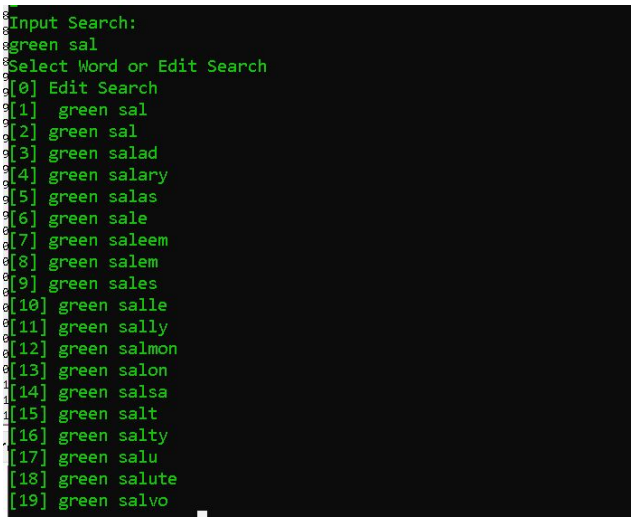

Figure 5: Searching Multiple Words

Part III: Auto Correct

Part III of the project calls for search entries that do not fall into the Trie Structure to be handled. The function created below corrects three common spelling errors of words that are misspelled by one letter. This handles the majority of typos written by users.

Search History Capability

```
void bad_search(string input, vector<string>& suggestion_list)  {
    string edit_input = input;
    int counter = 0;
    while(counter < 3){
        if (! word_check(input) )   {
            for (int i = 0; i < input.size(); i++) {
                for (char letter : alphabet) {
                    edit_input[i] = letter;
                    if (word_check(edit_input)) {
                        suggestion_list.push_back(edit_input);
                        counter++;
                    }
                }
                edit_input[i] = input[i];
            }
            for (int i = 0; i < input.size(); i++) {
                for (char letter : alphabet) {
                    edit_input = input.substr(0, i+1) + letter + input.substr(i + 1);
                    if (word_check(edit_input)) {
                        suggestion_list.push_back(edit_input);
                        counter++;
                    }
                }
            }
            for (int i = 0; i < input.size(); i++) {
                edit_input = input.substr(0, i) + input.substr(i + 1);
                if (word_check(edit_input)) {
                    suggestion_list.push_back(edit_input);
                    counter++;
                }
            }
        }
    }
}
```

The first spelling error that the bad_search() function checks is when the user types the wrong letter for the correct letter.

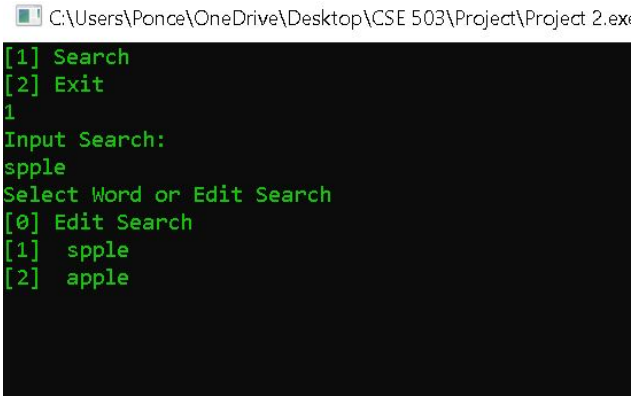

Figure 6: Corrects spple to apple

The second type of spelling error the function handles is for missing letters.

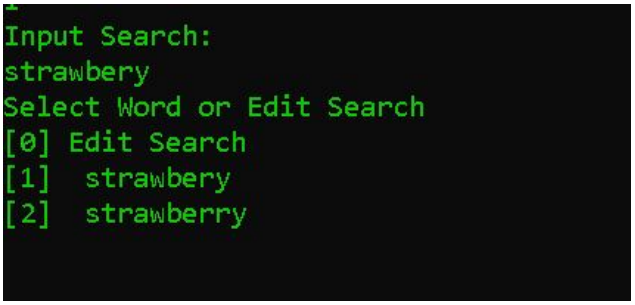

Figure 7: strawbery to strawberry

The third type of spelling error the function handles is for extra letters.

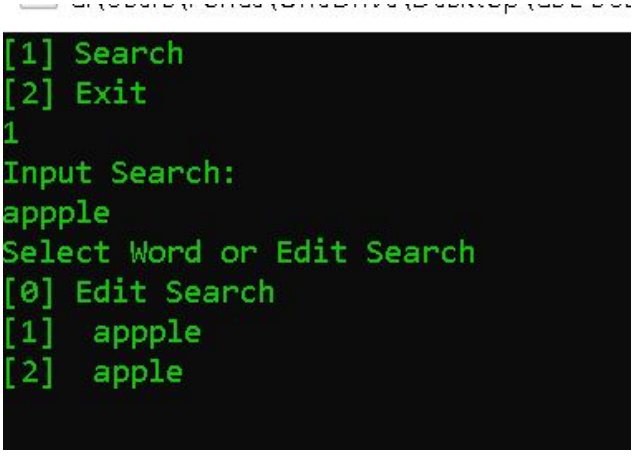

Figure 8: Corrects appple to apple

The bad search function limits itself to only ever offer at most 3 suggestions.

Program Analysis

The overall runtime of this program is complicated because of the number of different functions and inputs. Let's start with the insert() function.

```
void insert(TrieNode* insert_node, vector<string>& suggested_words, int depth);

int depth = splt_user_input[1].size() * 2;
```

The runtime of insert() relies on depth, which is a function of user input, and the number of words in the dictionary which is a constant.

d = depth
C = Number of Words in Dictionary
N = Length of User Input

Since there are 27 pointers for each TrieNode and each insertion compares the prefix and whole word against every word in the dictionary, the runtime of this function is $O(C \times 27^{d-N+1})$. Since d is a function of N the runtime of our function is simply $O(C \times 27^{N+1})$. Using proper Big-Oh Notation, this is simply $O(C \times 27^N)$. Not only is $O(27^N)$ large, but there are over 28,000 words in the dictionary. I think that the easiest way to make this function run faster is to always make the depth just one character longer than the inputted string length. This would then make the runtime $O(C)$.

There are several other ways to make this program more efficient. $O(27^N)$ is large, but N is usually always going to small since words are on average between 5 and 6 characters. We can increase efficiency by changing the depth size based on the input length. For example when a 4 letter input is being searched, the function is working the hardest because more words and prefix edges exist at 5 and 6 characters. This creates more and more branches which exponentially impacts run time. This is you might notice that run times for four letter inputs tend to be longer than run times for 8 letter inputs. Although the depth is larger for the 8 letter string, there will be way less edges in the TrieNode structure for the program to sort through; this is just the nature of the dictionary.

Another thought is to have the entire dictionary, and every existing prefix, stored already in the TrieNode structure. The dictionary is being stored as a list, which means that the insert function is limited in runtime by assessing if an edge to a new node exists. This would improve the efficiency because a new structure would not need to be built each time. However, the upfront cost of making this structure would take some time.

The other expensive function in the program is the very similar search history insert function:

```

void sh_insert(TrieNode* insert_node, vector<string>& suggested_words,
        vector<string> user_search_history)
```

The sh_insert() function isn't limited by depth in the same way the normal insert() is. If there is a 100 character string stored in the search history, the depth of the function will be a 100. When the search history is large and contains many strings with large number of characters, this will be an expensive function. I did leave it this way because I think it's important to have previously searched inputs as suggestions, regardless to their lengths.

Submitted by Matthew Poncini on 17 June 2024.