Data Structures Final Project Report

Student ID: 111062115 Name: 郭伊真

1. How you implement your code

I implemented my essay search by the data structure **Tries**.

(1) **Read .txt**

I use a while loop to go through every .txt and add up the variable ***i*** to determine which file to open. When encountering ***!fi.open()***, the loop breaks, which means we stop the "open file" section when the file is not exist.

(2) **Build Trie**

When we read a word of title or content, we put it into ***insert*** function as a parameter. For each character, we check if the corresponding ***child node*** exists in the ***Trie***. If not, then create a ***new TrieNode*** and establish the necessary ***parent-child relationship***. For subsequent search determination, we also mark the last node of the inserted word as a valid word by setting ***isWord*** to true and record them as ***leaf***s.

(3) **Handling words for searching**

To determine which search method to use, I first modified the ***word_parse*** function of the query so that it can capture specific symbols. In the ***search_mode*** function, I then evaluate the query word's head and tail to identify the associated symbols. Subsequently, I remove the first and the last char to obtain the word for search and its searching kind.

(4) **Exact Search**

In exact search, we traverse the ***Trie*** starting from the root. If any ***node==NULL*** or the required ***child node==NULL***, indicating the word is not present, then the function returns ***false***. If the traversal reaches the end of the word, it returns ***true*** only if the last node's ***isWord*** is true.

(5) **Prefix Search**

For prefix search, the process closely resembles exact search. The only difference is that it directly returns ***true*** after traversing a word, without checking if the node's ***isWord*** is true (as in prefix search, the last character may not be a ***leaf*** node in the Trie).

(6) **Suffix Search**

In suffix search, we first reverse the input word. Then, for each ***leaf*** node, we traverse the ***Trie*** upwards, comparing characters with the reversed word. If a match is found for the entire reversed word, the function returns ***true***, indicating the presence of the suffix in the Trie. Otherwise, it returns ***false***.

(7) Wildcard Search

In wildcard search, we use a recursively function ***wildcard_search_helper*** to implement the search. In this function, we use ***if-else*** to determine if the character is *******. If it is, it goes through all possibilities; if not, it continues the search by recursion to the next ***child node***. To go through all possibilities, if it matches zero characters(empty), we recursively call the function, incrementing ***word_idx*** to signify skipping. If it matches one or more characters, we use a ***for loop*** to iterate through all the ***child nodes*** of the ***Trie*** and call the function recursively.

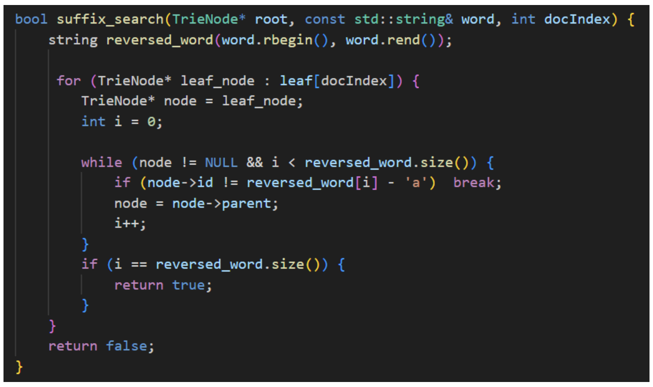
(8) operators

For operators ***+***, ***/*** and ***-***, we first modify the ***word_parse*** function of the query so that it can capture specific symbols. We store the previous search result as ***tmp_ans*** and do the corresponding ***operator*** operation based on ***q_word*** to obtain new ***ans***. We then continue updating ***ans*** with ***tmp_ans*** to facilitate multiple operator implementations.

2. Other implementations for optimization

(1) Initially, I was concerned about limited space, so I used a single ***query*** to run through all ***.txt files***, creating a ***Trie*** each time and scanning through each ***Trie***. However, I later realized that this approach was too slow. Given sufficient space, I modified the process to first scan through all ***.txt files*** and create all ***Tries***. Then, for each ***query***, I scanned through all the ***Tries***. This adjustment significantly improved the execution speed.

(2) For ***Suffix Search***, in my initial attempt, I traversed down the ***Trie*** from the ***root***. However, I find out a way of optimization. Therefore, I modified the code to store the ***leaf nodes*** when building Tries, reverse the word, and traverse from the ***leaf nodes*** upward. This modification also improved the speed!



```
bool suffix_search(TrieNode* root, const std::string& word, int docIndex) {
    string reversed_word(word.rbegin(), word.rend());

     for (TrieNode* leaf_node : leaf[docIndex]) {
        TrieNode* node = leaf_node;
        int i = 0;

        while (node != NULL && i < reversed_word.size()) {
            if (node->id != reversed_word[i] - 'a')  break;
            node = node->parent;
            i++;
        }
        if (i == reversed_word.size()) {
            return true;
        }
    }
    return false;
}
```

3. Challenges you encounter in this project, or Conclusion

In this project, I dedicated a large amount of time to optimizing runtime. Before it, I didn't have a strong concept of the ***execution speed*** for each line of code or logic. However, due to the necessity to carefully consider execution speed in this project, even though it required more time for contemplation, the effort in this project not only improved my coding skills but also gave me with valuable insights into the intricacies of optimizing performance. As a result, I emerged from this experience with an enhanced ability to create efficient and effective code.