KATOUZIAN Pouria s192865

Project 1

INFO0027 : Smart Keyboard

Université de Liège
Faculté des Sciences Appliquées
Année académique 2023-2024

1 Introduction

as working such as software engineer for the SNCB, the famous belgian railway company. They observe longer and longer queues behind each ticket vending machines, yielding frustrating wait times for hurried travelers. I am responsible for fixing this issue. Investigating vending machines data, I notice that a significant amount of time is lost, on the one hand due to spelling mistakes while entering the destination station, on the other because the validity check, which happens at each key press, is slow. by utilization the right data structures and algorithms, I ensure that, a good modular and robust program will fix the issue.

2 Implementation description

The smartkeyboard program is an efficient and robust solution for managing and searching through a dataset of words using a Trie data structure. This program consists of several different functions, with the most important ones being Word Insertion, Word Completion, Word Suggestion, and functions that play the role of memory management. Additionally, this program is robust and handles errors very carefully and effectively. Another solution for this problem could be using a Ternary Search Tree (TST). TSTs offer similar functionality to Tries but with some differences in their structure and implementation.

The Trie data structure is chosen over TST for the smart keyboard program because it is more memory-efficient, flexible, and easier to scale for larger datasets.

Hash tables and BSTs may not be a good choice for this project due to their complexity and limitations. Hash tables may lack certain functionalities such as inherent support for alphabetical ordering or prefix-based traversal, making them less suitable for autocomplete functionalities. On the other hand, BSTs may not efficiently support prefix-based searches, resulting in inefficiencies, especially for large datasets. Traversing BSTs for prefix matches can be particularly inefficient due to their structure.

In a nutshell, an alternative data structure to implement this program could be a TST, which provides better time complexity on average and the same complexity as Trie in the worst case. Using a TST could be more efficient if storing additional keys in each node would be necessary or helpful for different demanded operations.

3 Complexity

3.1 Time complexity

3.1.1 Suggest Words

Function	**Best Case**	**Average Case**	**Worst Case**
suggestWords()	O(1)	O(K)	O(M)
printSuggestWords()	O(1)	O(C)	O(ALPHABET_SIZE)
Overall Suggestion	**O(1)**	**O(K + C)**	**O(M * ALPHABET_SIZE)**

3.1.2 Complet Words

Function	**Best Case**	**Average Case**	**Worst Case**
completWord()	O(M)	O($M + H$) (balanced)	O($M \times H$) (unbalanced)
printWordComplet()	$O(1)$	$O(H)$ (balanced)	$O(H^2)$ (unbalanced)
Overall Suggestion	**O(M)** (balanced)	**O($M + H$)** (balanced)	**O($M \times H^2$)** (unbalanced)
	O(1) (complete word)	**O(M)** (non-existent prefix)	**O(M)** (non-existent prefix)

3.2 Espace complexity

3.2.1 Suggest Words

Function/Process	**Best Case**	**Average Case**	**Worst Case**
suggestWords()	O(1)	O(1)	O(1)
printSuggestWords()	O(1)	O(1)	O(1)
Overall Suggestion Space Complexity	**O(1)**	**O(1)**	**O(1)**

3.2.2 Complet Words

Function	**Best Case**	**Average Case**	**Worst Case**
completWord()	O(1)	O(1)	O(1)
printWordComplet()	O(1)	O(H) (balanced)	$O(H^2)$ (unbalanced)
Overall Suggestion Space Complexity	**O(1)**	**O(H)** (balanced)	**O(H^2)** (unbalanced)

Note :
M *represents the length of the actual prefix entered by the user*
C *represents the number of child nodes in the current node of the Trie*
K *represents the number of characters in the path from the root to the node representing the actual prefix or the complet word.*
H *refers to the maximum word length stored in Trie*

4 Feedback

This project proved to be quite interesting. Selecting the appropriate data structure posed a challenge initially, prompting the need for some research. However, the implementation phase was relatively swift ; I dedicated two days to coding, an additional day to address debugging issues that arose in a small portion of the program, and another day to analyze the code complexity and write the report