CAPSTONE PROJECT

Code Snippet Manager

Project ID: 12

Team Name: BEAT 100%

Team Members:

1) Krish Makhwana (202301103)
2) Dhruv Sanjaykumar Patel (202301024)
3) Dev Trivedi (202301150)
4) Neeraj Vania (202301060)

Github Repository Link :

https://github.com/Krish-Makwana-1205/DS-project-Beat-100

Introduction

This report provides a detailed exploration of the data structures employed within a Code Snippet Manager application leveraging a Graphical User Interface (GUI). It dissects the reasoning behind these selections, investigates their time and space complexities, and offers insights into the data structures deemed unsuitable for this project.

1. Project Overview

The project centered around the development of a Code Snippet Manager equipped with a user-friendly GUI. This application aims to streamline the organization, retrieval, and management of code snippets, fostering developer productivity.

2. Employed Data Structures: Hash Table

- **Definition:** A data structure that excels at rapid insertion, retrieval, and deletion of key-value pairs. It leverages a hash function to map unique keys (often code snippet identifiers) to their corresponding values (the actual code snippets).
- **Collision Resolution Technique:** Separate Chaining is employed to address collisions that arise when multiple keys hash to the same index within the table. In separate chaining, each index in the hash table acts as the head of a linked list, where elements that collide are stored.
- **Time and Space Complexity:** Under the assumption of uniform hashing (where each key has an equal probability of mapping to any slot in the table), the average-case time complexity for search, insert, and delete operations is O(1). This signifies that on average, these operations can be executed in constant

time. However, the space complexity scales linearly with the number of elements stored (O(n)) within the hash table.

- **Justification:** Hash tables were chosen due to their exceptional average-case performance in searching, inserting, and deleting code snippets. Since each code snippet likely possesses a unique identifier, hash tables provide efficient retrieval based on these identifiers, significantly accelerating the process of locating specific snippets within the manager.

3. Data Structures Not Considered

3.1 Stacks and Queues

- **Unsuitable Ordering Mechanisms:** Stacks and Queues enforce specific ordering disciplines:
    - Stacks adhere to a Last-In-First-Out (LIFO) principle, where the most recently added element is retrieved first.
    - Queues follow a First-In-First-Out (FIFO) approach, where the element that was added first is retrieved first.
- **Mismatch with Random Access Needs:** A Code Snippet Manager necessitates random access. Developers need the ability to retrieve any snippet at any given time, irrespective of when it was added. Stacks and Queues, with their predefined ordering, are not well-suited for this requirement.
- **Retrieval Challenges:** In a Stack, you can only access the element at the top. Retrieving a specific snippet within the stack would necessitate potentially

removing (and potentially re-adding) multiple snippets that were added later. This becomes highly inefficient, especially for large collections.

- **Limited Functionality in Queues:** Similarly, queues only allow access to the element at the front. While you could iterate through the queue to find a specific snippet, this approach is linear in time complexity (O(n)), making it slow for large numbers of snippets.

3.2 Arrays

- **Predefined Size Limitation:** Arrays offer efficient random access (O(1) on average) – you can directly jump to any element using its index. However, arrays have a significant drawback in the context of a Code Snippet Manager: their size must be predefined.
- **Inflexible for Dynamic Collections:** As a developer's codebase grows and shrinks, the Code Snippet Manager should gracefully accommodate these changes. Arrays cannot dynamically expand or contract in size. If the predefined size is insufficient, you'd need to create a new, larger array and copy all the existing snippets, which can be cumbersome and inefficient.
- **Resizing Bottleneck:** Resizing an array to accommodate new snippets often involves creating a new array, copying the existing elements, and then deleting the old array. This can be a time-consuming operation, especially for large collections of snippets.
- **Wastage with Unfilled Space:** Conversely, if the predefined size is too large, the array might have a significant amount of unused space. This is inefficient memory utilization, as the application is essentially reserving memory that isn't actively being used.

- **Unsuitable for Deletions:** Deleting elements from the middle of arrays can be complex and inefficient. It often involves shifting elements down the array to fill the gap, potentially impacting performance for frequent modifications.

3.3 Singly Linked Lists

While singly linked lists offer efficient insertion at the head (O(1)), their overall performance for random access and frequent modifications throughout the list can be less desirable compared to hash tables. Here's a breakdown of the reasons why singly linked lists wasn't chosen:

- **Limited Random Access:** Traversing a singly linked list to locate a specific snippet requires starting from the head and iterating through each node until the target is found. This operation can become time-consuming, especially for large collections of snippets. Imagine a list containing hundreds or even thousands of snippets. Finding a specific one using a linear search through a singly linked list would be inefficient.
- **Costly Insertions/Deletions in the Middle:** Inserting or deleting a node in the middle of a singly linked list necessitates manipulating pointers in multiple nodes, potentially impacting performance compared to hash tables where updates are localized to the specific key-value pair. Modifying elements in the middle requires navigating the list to find the target node, adjusting pointers in the preceding and potentially following nodes, which can be cumbersome.
- **Unfavourable for Frequent Lookups and Modifications:** Since code snippet managers involve frequent retrieval and modification of snippets in no order, the linear search and pointer manipulation associated with singly linked lists become significant performance bottlenecks. Developers expect quick access to their snippets, and hash tables excel in this regard.

4. Additional Considerations

This report offers in-depth analyses of the time and space complexities associated with each implemented function within the chosen data structures. Furthermore, it explores the data structures that can be effectively utilized for separate chaining within hash tables. Lastly, the report delves into the potential drawbacks of using arrays and singly linked lists in this particular scenario.

5. Conclusion

The strategic selection of hash tables, coupled with a thorough understanding of their time and space complexities, has demonstrably contributed to the development of an efficient and adaptable Code Snippet Manager. This application empowers developers to organize, retrieve, and manage their code snippets effectively, fostering a more productive development environment.

6. Reference

- https://www.geeksforgeeks.org/hash-table-data-structure/
- https://www.geeksforgeeks.org/inserting-elements-in-an-array-array-operations/
- https://www.geeksforgeeks.org/time-complexities-of-different-data-structures/