Hotel Management System Design Write-up

Introduction

In developing our hotel management system, I recognized the paramount importance of efficient data structures. This report encapsulates the thought process and decisions underlying the design of the Hash Table within our system. Each choice was made to optimize system performance.

Data Structures

HashTable

1.Hash Table Design Choices:

a. Capacity and Load Factor

When determining the initial capacity of the hash table, I aimed to strike a balance between memory efficiency and operational speed. The concept of a load factor became instrumental in ensuring that our hash table maintains an optimal density of data without compromising on performance.

b. Hash Function

The selection of the hash function, inspired by the DJB2 algorithm, stemmed from a desire to distribute keys evenly across the hash table. I wanted to mitigate collisions and create a more predictable and uniform retrieval pattern.

2. Asymptotic Running Time Analysis

a. find & findAll Operation:
Average Case: O(1)
Worst Case: O(n)

For the find operation, my design philosophy centered around achieving an average time complexity of O(1) due to the direct access nature of hashing. The hash function and linked list traversal were carefully considered to minimize the impact of collisions, resulting in efficient search operations.

b. insert Operation:
   Average Case: O(1)
   Worst Case: O(n)

In the insertion process, I opted for an O(1) time complexity on average. By implementing chaining for collision resolution, I aimed to ensure minimal disruption to performance. Additionally, the integration of Binary Search Tree (BST) insertion for city-related data was a deliberate choice to enhance the overall efficiency of the system.

c. erase Operation
   Average Case: O(1)
   Worst Case: O(n)

Efficiency in the erase operation was a key consideration. I strived for an average time complexity of O(1) while acknowledging that the worst-case scenario, involving traversing a linked list due to collisions, could lead to O(n) complexity. Collisions were addressed with a strategy that balanced simplicity and efficiency.

3. Collision Handling

My decision to adopt open addressing with chaining for collision resolution was driven by a desire for a practical yet effective strategy. The collision count was introduced as a metric to track and manage collisions, providing insights into system behavior.

Integration with Binary Search Tree (BST)

Integrating a Binary Search Tree was a deliberate choice to enhance specific functionalities, such as finding hotels in a specific city. This approach complements the hash table by providing a structured and efficient search mechanism for certain types of queries.

1. insert Operation

The insertion operation in the BST aims for an average time complexity of O(log n) for balanced trees. The insert method recursively navigates through the tree, maintaining the binary search property. However, in the worst case, when the tree is highly unbalanced, the time complexity can degrade to O(n), where n is the number of nodes.

2. remove Operation

The removal of a node in the BST is designed to maintain the binary search tree properties. The remove method has an average time complexity of O(log n) for balanced trees. However, similar to insertion, in the worst case of an unbalanced tree, the time complexity may reach O(n).

3. find Operation

The find method performs a binary search within the BST, resulting in an average time complexity of O(log n) for balanced trees. The worst-case scenario occurs when the tree is highly unbalanced, leading to a time complexity of O(n).

4. Tree Height

The height method calculates the height of a tree/subtree, providing insights into the tree's balance. In the worst case, the height calculation has a time complexity of O(n), where n is the number of nodes.

5. Asymptotic Running Time Estimates

insert: O(log n) on average, O(n) in the worst case.
remove: O(log n) on average, O(n) in the worst case.
find: O(log n) on average, O(n) in the worst case.

Conclusion:

In summary, our hotel management system utilizes a Binary Search Tree (BST) for ordered data and a Hashtable for fast average-case operations. The BST offers efficient retrieval with potential worst-case scenarios due to tree imbalance (O(n)) which I planned to get over by implementing an AVL tree but was not sure if we are allowed to do so. The Hashtable provides constant time complexity on average (O(1)) but may degrade in worst-case scenarios (O(n)) due to resizing. Balancing efforts for the BST and vigilant load factor management for the Hashtable would make our program more optimized and efficient.