Hotel Finder Program

Asif Tauhid

Introduction:
The Hotel Finder program is designed to efficiently manage a big and diverse database of hotels. There are six main commands that can be performed in this Hotel Finder program to help smoothen the experience of managing the hotel dataset. They are-

- insert <Hotel Details>
- find <HotelName,CityName>
- findAll <CityName>
- findAll <CityName,Stars>
- delete <HotelName,CityName>
- dump <file>

Here, insert ads a new hotel record to the dataset, find helps to find a specific hotel in a specific city, findAll <CityName> returns all the hotels in a specific city, findAll <CityName,Stars> finds all the hotels with a specific stars in a specific city, and dump creates a file with all the hotel current data.
This document will be outlining the design choices that has been made and why they help achieving modularity, efficiency, and scalability in implementing the program. The primary two data structures used here are Hash Table and Binary Search Tree (BST), which made the program significantly efficient.

Hash Table Implementation:
The Hash Table servers as the main data structure here for storing hotel records. It's because, it provides fast insertion and deletion of data, crucial for the program's efficiency, especially as the number of different cities and hotels can scale up to the thousands. For this program,

Design Choices for Hash Table:

- Hash Function: For an efficient hash function, I used cycle shift hash code (more specifically 5-bit cycle shift) to convert the key into integers and MAD (Multiply, Add, and Divide) method (multiplied by two prime numbers- 7 and 11) to do compression. All these were to get the less possible collisions. For our data, I got a collision rate of 36.239% while running the program.
- Collision Handle: For an efficient collision handle function, I used the separate chaining. This ensures efficient resolution of collisions and provide the flexibility of scaling the dataset.
- Dynamic Sizing: The initial size of the Hash Table is chosen based on expected input size. However, dynamic resizing ensures adaptability of the program, making it size efficient.

Binary Search Tree (BST) Implementation:
Here the BST complements the Hash Table, specifically for efficiently finding all the hotels based on their city names and number of starts they hold. It makes the program efficient by providing logarithmic time complexity for finding records.

Design Choices for BST:

- City-Based Structure: For the BST structure, it is built in the basis of the cities of the hotels. Thus, it made it efficient for searching and finding hotels based on cities rather than their names.
- Modularity: I designed the BST to be modular here, allowing it to easily integrate with the Hash Table. Each node of the BST represents a city, where for each city, all the hotels are stored in a vector. Thus, the finding function was way clearer.

Efficiency of Main Commands:

Because of the integration of the HashTable and BST, efficiency of the main commands were really good. Here's the worst-case asymptotic running time complexities of each of the commands-

- **Insert:** Here the insert function is using both for hash table and BST together.
    - For Hash Table: Even though the average case takes O(1), the worst case can take O(n). In the worst case, when there are many collisions and the hash table needs to resolve them using separate chaining method, the complexity is proportional to the number of collisions(n). Because it has to find if the same hotel exists or not to update the value.
    - For BST: Here the time complexity depends on the height of the tree. In average case the height is O(log n) and in the worst case the height will give a complexity of O(n)
    - Together: The average time complexity they give together is O(1) + O(log n) = O(log n) and for worst case, it's O(n) + O(n) = O(n)

- **Find:** The average case takes O(1) and worst case can take O(n). In the worst case, if all the keys get the same hash values, resulting in the same index, the time complexity becomes O(n) as it requires linear search in the chain.
- **FindAll (by City):** The average case takes O(long n) and the worst case can take O(n). The worst case is when all the records (hotels) are under the same city.
- **FindAll (by City and Stars):** Here, the average time complexity is O(long n) and the worst time complexity is O(n). Similar to the previous one, when all the records are under one specific city.
- **Delete:** O(1) average-case time complexity for an empty bucket with hash table erase function, whereas O(n) is the worst case. For BST, O(log n) and O(n) is the average and worst-case complexity of removing a node. Together the average time complexity they give is O(1) + O(log n) = O(log n) and for the worst case, it's O(n) + O(n) = O(n).
- **Dump:** The time complexity here is O(n). In the worst case, it iterates through the entire hash table and writes a worst n number of hotels from the hash table to the file.

However, the design choices made in the Hotel Finder program overall increases efficiency, modularity, and scalability for many cases. Because if the same program was implemented with other data structures like Array based list or linked list, most of the function would have taken an average runtime of O(n). Overall, the combined implementation of a Hash Table and BST here in this hotel finding program allows for a fast and well-organized data storage, retrieval, and manipulation.