Hotel Finder Application using HashTable and BST - C++

13.12.2023

-

Haad Mehboob
New York University
Saadiyat Island, Abu Dhabi
hm2957@nyu.edu

Overview

This is a basic implementation of Hash Table and Binary Search Tree working in conjunction to produce an efficient data structure that supports a large set of hotel records. Although the methods employed. This document provides a brief overview of the code's implementation schemes, goals, and key features. (Tested on version 11.3.0)

Goals

1. Efficient Search.
2. Scalability
3. Flexibility
4. Readability
5. Simple User Interface

Specifications

1. Hash Table (‘hashtable.hpp’, ‘hashtable.cpp’)

   The hash table is used to map keys, which are generated using a simple cyclic hash function from the City name and Hotel name. An array of lists mechanism is used to handle chained collisions. The bucket contains Entry objects, which are <key, value> pairs where the key is HotelName, CityName and the value is the Hotel object pointer itself.

   The major functions include insert, find, findAll erase, and dump. Specifically, the findAll method makes efficient use of the binary search tree to efficiently locate hotels based on criteria.

2. Binary Search Tree (‘bst.hpp’, ‘bst.cpp’)

   This binary search implementation is special in the sense that it is double-templated. The node itself is an object that refers to the CityName, based on which lexical order is generated for tree structure. An individual node is a connector to a vector of type Hotel which generates efficiency when looking at records for a single given city. The find method in the BST also allows a search for a node based on the city name and stars criteria, making use of helper print functions and recursive traversal.

3. Hotel Class (‘hotel.hpp’, ‘hotel.cpp’)

   This represents the simplest pack of this program, which is a package of a hotel object. This bundles together properties like name, city, stars, price, country, and address. It provides methods for getting details and printing helpers.

4. Main Application (‘main.cpp’)

   This implements the main driver program, which loads a CSV file at the stars and builds a hash table along with a binary search tree. This code also provides a CL interface for operatives and supports the commands displayed at the start screen.

Complexity Analysis

I. Hash Table:

- Insert('insert') - O(1) amortized, O(n) worst case if many collisions occur
- Find('find') - O(1) amortized, O(n) worst case
- Erase('erase') - O(1) amortized, O(n) worst case for linear bucket search
- Dump('dump') - O(n), where n is the number of entries in record

II. Binary Search Tree:

- Insert('insert') - O(log n) amortized, O(n) worst case if tree is unbalanced
- Find('find') - O(log n) amortized, O(n) worst case for unbalanced tree
- Erase('remove') - O(log n) amortized, O(n) worst case for unbalanced tree
- Height('height') - O(log n) amortized, O(n) worst case

Potential Improvements

1. Balancing the Binary Search Tree

   A self-balancing BST implementation such as AVL or Red-Black would create better performance in worst-case scenarios and ensure logarithmic nature.

2. Better Hash Function

   An improved hash function could result in a lower collision rate in the hash table hence creating opportunities for faster access time.

3. Robust User Interface

   A better user experience can be created by making the program more interactive through graphical representation (GUI).