CS-UH 1050
May 9,2019
Professor Lefteris Kirousis and Instructor Khalid Mengal

Project 3 Documentation

Junior Garcia(jfg388) and Khalid El Essawi(kee289)

For the final Data Structures Project, we were assigned to create a *hotelfinder* application that stores,retrieves, and deletes records of hotels. In order to do so, we implemented the HashTable data structure to ensure optimal efficiency and execution of queries. Using this data structure, we are able to manage hotel databases of 1,000, 10,000, and 100,000 hotel entries and execute commands in optimal time. In the following paragraphs, we will further explain the design choices of our program as well as the expected worst-case complexity of our algorithms.

Main Design Choices:

- Create two hashmaps that store the file's information simultaneously

One of the major design hurdles that our program had to tackle was to employ the *AllinCity* functionality. This function takes in the city name as the key and retrieves from the hashtable all the entries that have the city as the key. Since our original hashmap, named *myHashmap*, stored the hotel's information using the a combination of hotel name and city name as the key, we decided to create an additional hash table that stores the hotel entries using the city name as the key and name it *myCityHash.* This decision entailed a modification of our HashMap Class so it differentiates between nodes that are for either *myCityHash* or *myHashmap* and performs the corresponding version of the Hashmap functions for each case.

- Conditional switch statements in main program

To ease the program's execution for the user, we decided to use the conditional switch statements so that the user is able to choice one of the 6 choices our program allows.

Classes Implemented

- HashNode: The HashNode class is used to make objects of the hotel entries found in the csv file. By doing this, we tokenize the important information found on each hotel entry (hotel name, city name, key, and value) and we make it a private member of our class.
    - Private Members
        - Hotelname, cityname, key,and value: In order to differentiate between a *myCityHash* node and a *myHashmap* node, we place the following

condition on all methods so that the correct version of the method runs whenever it is called: key==city. If the aforementioned is true, then the node is part of *myCityHash* and if not then it is a *myHashMap* node.

- Public Methods
  - A constructor to instantiate a Hashnode object.
  - The remaining methods of the class are just getter functions to allow access of the private data members of the class to other parts of our program.
    - Worst-case complexity of getter functions: O(1).

- Hashmap
  - Private Members
    - **Buckets:** A doubly-linked list of type Hahsnode that stores pointers to buckets. This allows for Chain Hashing, which leads to the dealing of collisions(when two keys produce the same hashcode) in a more efficient manner.
    - **Size:** Current Size of HashMap
    - **Capacity:** Total Capacity of HashMap. A big initial capacity must be given so as to reduce the number of possible collisions.
  - Public Methods
    - **HashMap:** constructor to instantiate a Hashmap object.
    - **hashCode:** produces of hashcode (position in the array) for the given key. In order to do reduce collisions, we chose a hashcode that adds the ascii code of each letter times a counter to the power 10 and then gets the remainder of the total sum. This non-trivial method reduces the number of collisions as there is less of a possibility for the same hashcode to appear for different keys.
    - **insert:**The function compares the key of the hash node and the key to differentiate between a *myCityHash* node and a *myHashmap* node. In both cases, the function gets the bucket's index from the hash code and then adds the node at the end of each bucket. The *myHashmap* nodes, however, check whether any node in the bucket has the same key as itself to overwrite its value with the given node. The *myCityHash* nodes do the same but the the overwriting occurs whenever the hotel name of the inspected node equals the hotel name of the inserted node.
      - Average complexity : O(1)
      - Worst case complexity: O(n). This complexity occurs in the rare case that all nodes reside within one bucket.
    - **Search:**The function compares the key of the hash node and the key to differentiate between a *myCityHash* node and a *myHashmap node*.In both

cases, the function gets the bucket's index from the hash code and then iterates through all the nodes in the bucket to find the target node. In the *myCityHash* case, every single node with the key will be printed out to execute the AllinCity functionality.

- Average complexity : O(1)
- Worst case complexity: O(n) in the rare case that all nodes reside within one bucket.

- **remove:** The function utilizes the same logic as the insert function but deletes a node instead of inserting it.
  - Average complexity : O(1)
  - Worst case complexity: O(n) in the rare case that all nodes reside within one bucket.
- **getSize:** returns the size of the hash map
  - Worst-case complexity: O(1)
- **getBucket(): returns the list of buckets**
  - Worst-case complexity: O(1)
- **sortNodes():** This function creates and returns a vector of the nodes in alphabetical order based on the hotelname. It uses the built-in sort function from the algorithm library and sorts them in nlogn time.
  - Worst-case complexity: O(nlogn)
- **~HashMap():** destructor to destroy an instance of the class Hashmap.

Auxiliary Functions

- **Mycomp:** Auxiliary function that compares two strings and returns 1 if the first one is higher or not. Used in the **sortNodes()** function.
- **Sort:** Sorts a vector of strings in alphabetical order and returns it. Used in the **sortNodes()** function.
  - Worst-case complexity: O(nlogn)
- **Dump**
  - Takes in the sorted nodes and places them in a csv file in alphabetical order.