hotelFinder : a hashtable based storage and retrieval application

Máté Hekfusz, Li Cheng

7th December 2019

hotelFinder is a program written in C++ which manages hotel data. It reads a csv file of entries and stores them in two hash tables. From there it can quickly retrieve, insert, or delete data. It also contains a few additional functions: *allinCity* which lists all hotels in a given city, and *dump* which stores the entire content of the tables in a new file in alphabetically sorted order. This writeup will give a brief overview of the program's functionalities.

1 File Structure

The program is divided into three cpp files and their header files, as well as a makefile. The hotelFinder.cpp file is the main file which contains hashnode methods, utility functions including a prime-finder and a keymaker, and the main function which takes an input file from terminal and parses it into the hash tables. The user has six commands they can issue, including 'quit', which releases all dynamically acquired memory through destructors before exiting the program. The main functions of the HashMap.cpp file are insert, find, remove, and dump, which will all be elaborated on below. CityMap is a child class derived from HashMap and its methods are stored in the separate CityMap.cpp. It also stores the entries in array, however it uses only the city name as a key instead of hotelname + cityname. It uses separate chaining in addition to open addressing, which means that every hotel in the same city will be stored in the same array position in a linked list. It is used for the allinCity command.

2 Main File

First, the number of entries in the file is determined in order to calculate the size of the hash table. It is calculated by taking the number of entries in the file multiplied by 1.333 (to decrease the load factor), and taking the next larger prime following that number. This method aims to reduce the number of collisions because when the modulus of a prime is taken, it is less likely to result in the same values. The hashing function uses the method we learned in class, which is to sum up the ASCII values of the characters in the key and multiply each by the square of their position in the entry string. Then, the function takes the sum's modulus with the table size we calculated before. Once the hash table size is calculated, the entries are inserted into the hotel table and the city table according to their hash codes.

3 HashMap

3.1 insert

The insert function takes a key (hotel name + city name) and value (the whole entry) as input. It calculates the hashcode using the key and searches for it in the array using the following method:

1. It checks if the array at the position of the hash is a nullpointer. If so, a new hashnode is created and the value inserted.
2. Else, it might be that the node *isAvailable* which means that there had been a value in that place which was 'deleted', or more accurately speaking, marked for deletion. If so, we delete the old node and make a new one in its place with the new value.
3. It checks if the key to be inserted is already there. If so, we do not need to add it again. This simple check works because all subsequent additions of the same entry would have the same hash, would be moved along the same path and thus would inevitably run into the first definition.
4. In all other cases, we increment the hash and repeat steps 1 to 3. If the end of the hash table is reached but there is still space in the array, the hash resets to 0 and checks from there onwards, essentially wrapping around the array. If the hashtable is full, it returns an error message.

3.2 find

The find function is very similar to the insert method. It starts searching the array from the hash position and if it finds an entry with a matching key it returns the value. Otherwise it increases the hash. It does so until it encounters a nullptr, because in that case it the key is not present in the array. To account for deleted nodes (to keep searching when encountering a deleted node) we use a boolean called *isAvailable*. When an element is removed, instead of deleting the node, the *isAvailable* marker is set to true, meaning it is available for insertion. The actual node is only deleted when there is a new entry inserted in its place. Since the node is technically still there when searching, the find function does not erroneously terminate when finding it and runs normally instead.
If the incremented hash reaches the end of the table, it resets to zero and keeps searching. This increases running time slightly as compared to a simple termination on reaching the end of the array, but it is more space efficient since the array can always be completely filled. To avoid an infinite loop in case of a full table, the boolean *wrappedAround* indicates whether the hash has been reset once before, and if so, it means it has iterated through the entire list and the hotel was not found.

3.3 remove

The remove method works in the same way as the find method, but instead of returning the key when it is found, it sets the *isAvailable* marker to true, marking the node for overwriting by a potential insert later on and hiding it from the find and remove functions. Else, it returns an error message that the hotel was not found.

3.4 dump

The dump function iterates over the entire table and stores all the non-nullptr node values in a vector. Then, it applies the STL sort() to the vector, sorting it alphabetically, before finally writing its contents to the specified file.

4 CityMap

CityMap is a derived class from HashMap because of the similarity of their main functions and so that it can rely on the HashMap's hashing function. The main difference is that the city name is used as the key, and entries in the same city are stored in a linked list (seperate chaining). However, the cities themselves are still stored using linear probing, as it is still possible that two cities have the same hash.

4.1 allinCity

The purpose of having the CityMap is to quickly retrieve all hotels in a given city. Since all hotels in a city are stored in a linked list (the STL list) at the same node of the array, the *allinCity* function first finds the node with the matching input key with the same method as HashMap's find(). Then it prints the contents of the linked list at that node.

4.2 Synchronization to HashMap

To ensure coherence between the two hash tables, the add and delete prompts call insert() and remove() for both tables. In the CityMap, this is achieved in two steps: first the node with the same key (city name) is found in the table, then the entry is either added to its linked list or found within it and removed. If a removal makes a list empty, its node is marked as 'available', just like with deletion in the HashMap.
Additionally, the HashMap methods are always run first, and they return information (in form of a boolean) about whether an entry was inserted/deleted successfully. The matching CityMap method is only run if this was the case, ensuring total synchronization.

5 Time Complexity

5.1 Storing the File

The program can be considered very efficient in terms of running time and occupied space due to the use of the hash table data structure. The time for initially parsing a file of size n into the hash table is $O(n)$. This is due to the following operations:
1. Getting the size of the file takes $O(n)$, as it iterates over every entry.
2. Computing the hashcode for all entries takes $O(n)$.
3. Inserting all nodes takes an expected time of $O(n)$, given that the hash code distributes the entries more or less evenly over the array. In the worst case, each entry has the same hashcode, and linear probing insertion would take $O(n^2)$ as each insertion increments the hash as many times as the number of elements already in the table.
Since our hash code leads to a close to random distribution, and no other operation exceeds $O(n)$, this is the overall complexity for storing the entire file into the hash table.

5.2 find, delete, add, dump, allinCity

Deleting, adding, and finding an element all use linear probing, which is $O(1)$ in the average case. CityMap also uses separate chaining, which trades less probing with looking through a list, also averaging $O(1)$ total. They are also very fast when real execution time is measured, generally falling between 0-5 ms even for massive datasets.
Dump is $O(n\ logn)$ due to the time it takes for the sorting of the data set. Iterating over the array and outputting the sorted list into a file take $O(n)$ time each, which means that sorting is the bottleneck.
allinCity takes constant time to find the city node. The time it takes to print the entries is $O(k)$, where k is the number of hotels in that particular city. On an average case, this can also be considered $O(1)$ and can at most be $O(n)$, if every single hotel is in the same city.