Iva Tomevska
Magdalena Gunkova

A short write-up about the algorithmic/design choices

Our program starts with prompting the user to enter the initial function ‘init’. If He enters another function the program tells that the user needs to start with the ‘init’ function. This is because that function initiates the main-memory structure that we use- a Hash Map, that will accommodate the incoming data from the text file (which is also taken from the command line). If init is executed for the second time around once your program has commenced operation, it will have no effect, and the program tells that to the user. Inserting into a Hash Map has O(n) worst time complexity because if too many elements were hashed into the same key: looking inside this key may take O(n) time. However, it is said to be O(1) average and amortized case because it is very rare that many items will be hashed to the same key [if you chose a good hash function and you don't have too big load balance]. The has function that we use takes the name and the surname as a key: for every character of the key we take the ASCII value and we multiply it by the index of the character in the key, we add it together and when we finally have the sum of the characters we multiply it by the size of the key. This method makes every key unique because if we don’t multiply the characters with the indexes we will have a lot more collisions.

After this the user has 7 options:

1. To add a record to the phonebook: this method takes the whole record and adds it in the Hash Map if it doesn’t exist. It adds it based on the key: it calls the function *insert()* which separates the name and the surname from the rest of the record; it converts them into a key and then it adds the whole record to the Hash Map. If there aren’t many collisions the time complexity is constant.
2. To find a record: this method finds the element having a key (name and surname) and display the entire record. It gets the key and it goes in the hash map: if it finds the key it prints the record, and if it doesn’t it tells the user that the record does not exist. If there aren’t many collisions the time complexity is constant. This method also displays the time that the program will use for searching.
3. To delete a record: the method works the same as the previous two methods, just when it finds the key it replaces it’s value with the word “deleted”. If no such element exists, print out a diagnostic message to the standard error. If there aren’t many collisions the time complexity is constant.
4. To read into the Hash Map the content of the file f. This function adds the content of the file to the existing Hash Map and it is used by the init method also to initially load the first file into the map.

5. To dump the content of the entire Hash Map into the file f and sort the content of the file f according to the last name of the customers in alpha-numerically increasing order. This method creates a list which will contain the records. We decided to use a list so we can sort the data faster. It loops through the whole Hsh Map (O(n) time complexity, n being the number of elements in the map) and it adds everything that has a value (that is not null or deleted) to the list. Then, we swap first and last name (O(1) time complexity) because we want to use the last name to sort the list. Then, we sort the list using a predefined sorting function for lists from the STD library. This function runs O(nlogn)- and it uses intro sort, which is a combination of quick sort and heap sort, and then we stream it into a new file.
6. Retrieve and store in a file all customers residing in a specific city. This method loops through the entire Hash Map and takes the city from every value of every key and it displays the whole record when it finds the matching city. Time complexity O(n), n being the number of records in the map.
7. Terminate the program with graceful release of all dynamically acquired memory by entering ‘quit’ in the command line.