Problem 11 - Phonebook

Object Oriented Programmin
Group CEN2.2A,
IInd $year$, 1st $semester, Computers$

Frăsineanu Claudia Andreea

1 Introduction

Design and implement a module for managing operations performed on a phone book. Each user is identified by its name, address and phone number. The module must allow to solve the following problems:

1. Adding a new user,
2. Deleting a user,
3. Searching a user by name and address (the string resulted by concatenating the two sub-strings is supposed to be unique) and printing its phone number,
4. Loading the phone book from a file and saving the phone book on a file.

2 Algorithm

The aim of a phone book is to store the data corresponding to a user, and to facilitate searching for such data. As a result, all information is sorted alphabetically.

Due to this fact, when implementing a phone book one must take into consideration that there are 2 important functions with witch one must operate: search and insertion. Because we work with ordered lists, an efficient algorithm that comes to ones mind easily is the classic *binary search* algorithm, which will undergo a few changes to accommodate the problem's requirements.

To begin with, the variable chose to represent such data can be seen as a matrix, having the dimension $M_{n\times 3}$, where:

i n represents the number of people in the list,

ii column 1 represents the name

iii column 2 holds the address,

iv column 3 contains the phone number.

2.1 Adding a user

Since we start from the assumption that the list is ordered, eacht time want to insert an element, we will just need to search for the position where it should be inserted such that the list remains ordered. Hence, our adding algorithm will turn into a searching algorithm, which will return the position where we have to insert our element. Then, it only remains to use the *.insert()* function, already implemented in Python. The algorithm will have the simple form:

```
ADDUSER($phonebook, newName, newAddress, newPhone$)
1  $position = searchInsert(phonebook, newName, newAddress, newPhone)$
2  insert (newName, newAddress, newPhone) in phonebook
```

In *searchInsert* we take as parameters the *name* and *address* and search through the list using the iterative version of *binary search*. Since the algorithm compares only one value with another at a time, we have to concatenate the *name* and *address* into a single string. This is done both for the given variables, and for each new iterator.

```
SEARCHINSERT(phonebook, newName, newAddress)
1  n ← size of phonebook
2  lowerBound ← 0
3  uppertBound ← n-1
4  concatenate newName and newAddress into one string
5  while we still have subarrays
6      middle ← lowerBound + uppertBound
7      concatenate iterator string's name and address
8      if givenString = midString
9          update lowerBound
10     else if givenString <midString
11         ignore right half
12         retain position
13     else if givenString >midString
14             ignore left half
15             retain position
16 return position
```

2.2 Deleting user

In order to delete an user, we must first check if it is present in our list. To do this, we use a function similar to *searchInsert*, called *searchUser*. Again we will use binary searh, and will return the position (which will be a positive integer ≥ 0) if we find the element, or -1 if there is no occurence of the element in the current phone book.

```
SEARCHUSER(phonebook, newName, newAddress)
1  n ← size of phonebook
2  lowerBound ← 0
3  uppertBound ← n-1
4  concatenate newName and newAddress into one string
5  while we still have subarrays
6      middle ← lowerBound + uppertBound
7      concatenate iterator string's name and address
8      if givenString = midString
9          update lowerBound
10     else if givenString <midString
11         ignore right half
12     else if givenString >midString
13             ignore left half
14 if no element was found
15     return -1
```

In the function *deleteUser* we will call the above function and perform the operations in accorance with the result:

```
DELETEUSER(phonebook, newName, newAddress)
1  position ← searchUser(phonebook, newName, newAddress)
2  if position = −1
3      return position
4  delete element at position
5  return 1
```

2.3 Loading and Uploading a file

When we handle input and output file, we read (or print) it line by line. Every line contains the *name*, *address* and *phone* number (in that particular order), and each of these is separated by a comma and a blank space.

3 Experimental data

In order to facilitate the testing of the algorithms, all necessary data is already stored and ready to be accessed. A text file, called *input.txt*, is used to load the first instances. Then, for each operation there are several test cases embedded within each function to ensure that all possible situations are verified.
Generally, there are four cases:

i the user is placed at the beginning of the list;

ii the user is placed in the middle of the list;

iii the user is placed at the end of the list;

iv there are two users with the same name, but different addresses.

4 High level application design

4.1 Modular structure

The is composed of only two Python files, one containing the functions needed to perform the operations, and another one where these functions are called and tested. The modular structure is illustrated in the below figure (1).

4.2 Project hierarchy

Within the archive, there are two Python modules, holding the functions used to solve the given problem. In addition, there is a text file containing a generated input, which can be loaded and used for testing, and another text file where we can upload the phonebook, at any given time.

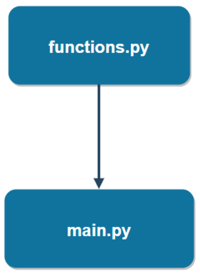

Figure 1: Modular structure

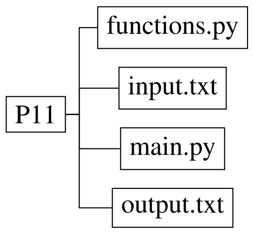

Figure 2: Project hierarchy

4.3 Input data

As we have stated before, in section 2, there will be several lines, each containing three elements: name, address, phone number. Thus, we can see the input as a matrix, with n lines and three columns. Each line and column has the starting index equal to zero.

$$Input = \{\ \text{x} \in M_{n\times 3} |\ x_{k,0} = \text{name},\ x_{k,1} = \text{address},\ x_{k,2} = \text{phone number},\ \forall k = \overline{0, n-1}\}$$

4.4 Output data

Due to the fact that the the input will not be altered in any way, other than adding and removing some elements, the output will be represented using the same rule. It should be noted that we cannot assume that the dimension of the matrix is the same.

$$Output = \{\ \text{x} \in M_{m\times 3} |\ x_{k,0} = \text{name},\ x_{k,1} = \text{address},\ x_{k,2} = \text{phone number},\ \forall k = \overline{0, m-1}\}$$

5 Functions and variables

The entirety of the input will be stored within a "list of lists", or a 2D matrix, called *phonebook*.

5.1 functions.py

I *printPhonebook(phonebook)*: this is an utility function I added to the project, which will print the phone book in the terminal. At the beginning of every line there will also be printed the index (starting from 0) of each person. The function has no return value.

II *loadFile(phonebook)*: here we open the text file *input.txt* in order to read the data and store it in the variable *phonebook*. The variables are set apart through the separator ", ". At the end the file is closed. We will return no value.

III *uploadFile(phonebook)*: in this function we open the text file *output.txt*, where we will store the current version of the *phonebook*. Just as before, between the variables placed on the same line, there will be a separator, ", ". After all the values are uploaded, the file is closed. Again, the function will not return anything.

IV *searchUser(phonebook, name, address)*: this function uses *binary search algorithm* in order to efficiently find a user, by *name* and *address*. It does so by concatenating the two strings and repeatedly dividing the list into two halves. At each iteration, the values of the current iterator name and address are concatenated and compared to the original values. Depending on the outcome, we will either continue to search in the left half, or the right one. At the end we will either return the *position* of a user, if it is found in the list, or *-1* otherwise. Further details can be found here 2.2

V *searchInsert(phonebook, name, address, phone)*: this is an utility function used in order to insert a new user. Because adding a user at the end and sorting the new list would prove highly inefficient, we can keep the list sorted just by adding the user at the right place. Hence, this function is similar to the previous one, except for two things:

- it searches a user by all of its values: *name*, *address* and *phone number*
- if a user is not present in the current list, it will return the position where is should be inserted.

For further explanations of the algorithm, check section 2.1

VI *addUser(phonebook, newName, newAddress, newPhone)*: here we have a variable *position*, holding the value returned by the function *searchInsert()*. Then, using the *.insert()* method, we will insert an element at *position*, and shifting the other elements to the right. The algorithm is explained in more detailed here 2.1

VII *deleteUser(phonebook, name, address)*: it is similar to the *addUser()* function. We have a variable *position*, which this time will hold the value returned by the function *searchUser()*. If it is -1, we return it to signal the user is not present in the list. Otherwise, we delete the element at *position* and return 1 to signal we have successfully deleted it. In depth explanation is found in section 2.2

5.2 main.py

Within this module, the user can chose among seven options. For this, seven additional functions have been created, each of them performing a specific task. Depending on the user input, one of them is chosen.

I *one()*: upload data from a text file

II *two()*: adding a user to the current list. Four persons are inserted, covering the cases stated in section 3.

III *three()*: search a user by name and address. The same cases (3) are covered.

IV *four()*: delete a user from the list. Here three users are chosen: one for a basic case, one when two users have the same name, and another one when a user is not present in the list.

V *five()*: upload the function to a text file

VI *six()*: print the current phone book in the terminal

VII *0*: stops the program

References

[1] https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm, *Binary search*, (accessed at November 19th 2020)

[2] https://jaxenter.com/implement-switch-case-statement-python-138315.html, *Python switch*, (accessed at November 20th 2020)

[3] https://docs.python.org/3/tutorial/datastructures.html, *Data structures*, (accessed at November 20th 2020).