Introduction In this work you will implement a program that will accept, process, record and answer questions about vaccinating citizens. Specifically, you will implement a set of structures (bloom filters, linked lists, skip lists) that allow the import and query of large volumes of **citizenRecord type registrations.** Although the exercise data will come from files, eventually all records will be stored in main memory only.

Bloom Filter - Structure

Description A bloom filter (see https://en.wikipedia.org/wiki/Bloom_filter) is a compact structure used to check for the existence of an element in a set. More specifically, using the bloom filter for a given item, we can draw one of the following conclusions. Either that (I) the item **certainly does not belong** to the set, or that (ii) the item is **likely to belong** to the set, but we do not know for sure.

That is, the bloom filter can, in some cases, respond positively that an element belongs to the whole, even when it does not belong (false positive). On the contrary, the negative answer is always valid, that is, when the bloom filter indicates that an element does not belong to the whole, that element certainly does not belong. In this way, the bloom filter makes an overestimation of the elements of the set. The probability of error (in case of a positive answer) increases as data is added to the set.

A bloom filter is represented as a table of bits of *M* positions (initialized with 0), and is accompanied by *K* hash functions. The following are the basic actions it supports and how they are implemented:

1. To insert an element *s* into the set, we apply each of these *K* hash functions to that element. These will return K positions in the table (some of which may coincide). Then, in each of these positions we set the corresponding bit to 1.
2. To check if an element *s* belongs to the set, we apply each of these *K* hash functions to that element. If in each of the *K* positions that return to us there are 1, then the element *s* (probably) belongs to the set. Otherwise (if there is even one 0), the element certainly does not belong to the set.

Example Let M = 16, K = 3, and hash1 (), hash2 (), hash3 () be the three hash functions given to us. The table of bits will initially be: 00 00 00 00 00 00 00 00.

Suppose we enter successively the elements s1, s2, s3, for which we have:

hash1 (s1) mod 16 = 3 hash1 (s2) mod 16 = 10 hash2 (s1) mod 16 = 7 hash3 (s1) hash2 (s2) mod 16 = 11 hash3 (s2) mod 16 = 7

hash1 (s3) mod 16 = 2
hash2 (s3) mod 16 = 4
hash3 (s3) mod 16 = 2

Bit array ==> 00 01 00 01 00 01 00 00 ==> 01 01 00 01 00 11 00 00 ==> 01 11 10 01 00 11 00 00

Let's then test the existence of the elements s4, s5, s3, for which we have: hash1 (s4) mod 16 = 5 hash1 (s5) mod 16 = 2 hash1 (s3) mod 16 = 2 hash2 (s4) mod 16 = 10 hash2 (s3) mod 16 = 4 hash2 (s5) mod 16 = 6 hash3 (s4) mod 16 = 1 hash3 (s5) mod 16 = 11 hash3 (s3) mod 16 = 2

Bit array ==> 01 1̲1 10 0̲1̲ 00 11 00 00 ==> 01 1̲1 10 01 00 1̲1̲ 00 00 ==> 01 1̲1 1̲0 01 00 11 00 00

Answer: no Answer: yes Answer: yes

In the case of s5, we have a false positive since the element seems to belong to the whole, when in fact it does not belong. The other two answers are valid.

Skip List - Structure Description

A skip list is a probabilistic data structure that allows **O (log n)** search complexity as well as **O (log n)** import complexity within a sorted **n** list of items. This way it can get the most out of a sorted table (for search) while maintaining a linked list type structure that allows importing, which is not possible in a table. Quick search is made possible by maintaining a linked list hierarchy, with each successive list omitting fewer items than the previous one. You can find out more about this data structure at https://en.wikipedia.org/wiki/Skip_list. Also many YouTube video tutorials like this https://www.youtube.com/watch?v=UGaOXaXAM5M.

there are also in the

A) The application (75%)

The application will be called **vaccineMonitor** and will be used as follows:

```
./vaccineMonitor -c citizenRecordsFile –b bloomSize
```

where:

- The **bloomSize** parameter specifies the size of the bloom filter in * bytes *. Indicative size of the bloom filter for exercise data will be of the order of 100Kbytes.
- **CitizenRecordsFile** (or some other file name) is a file that contains a number of citizen registrations to edit. Each line in this file describes a citizen's vaccination status for a specific virus. For example if the contents of the file are:

889 John Papadopoulos Greece 52 COVID-19 YES 27-12-2020
889 John Papadopoulos Greece 52 ÿ1ÿ1 ÿÿ
776 Maria Tortellini Italy 36 SARS-1 NO
125 Jon Dupont USA 76 H1N1 YES 30-10-2020

means that we have four registrations describing three citizens in three different countries (Greece, Italy, USA). **YES** indicates that the citizen has been vaccinated (on the following date) while **NO** does the opposite. Specifically, an entry is an ASCII line of text consisting of the following elements in this order: - **citizenID:** a string (it can have a single digit) that defines each such

sign up.
- **firstName:** a string consisting of letters without spaces. - **lastName:** a string consisting of letters without spaces.
- **country:** a string consisting of letters without spaces.
- **age:** a positive (> 0), integer <= 120.
- **virusName:** a string consisting of letters, numbers, and possibly a hyphen “-” but without spaces.
- **YES** or NO: indicates whether the citizen has been vaccinated against the virus.

- **dateVaccinated:** date the citizen was vaccinated. If the previous field is NO , not
  there is a **dateVaccinated** field in the record.

To get started, your application should open the **citizenRecordsFile** file to read the lines one by one and initialize and store the data structures it will use when
executing queries. You should check that the information in the files is valid. For example, if in the **citizenRecordsFile** file there are two inconsistent entries with the same **citizenID,** you should ignore the second entry. Also, if you find a NO entry that has a vaccination date, the entry should be discarded. Ignore the incorrect entries when reading the file by printing the message:

ERROR IN RECORD theRecord

where **theRecord** is the problematic record.

When the application finishes editing the **citizenRecordsFile** file, it will wait for user input from the keyboard. The user will be able to give the following commands (arguments in []
are optional):

Vacc / **vaccineStatusBloom citizenID virusName**

The application will check the bloom filter associated with **virusName** and will print a message on whether the citizen with **citizenID** ID number has been vaccinated against **virusName.** (See below in detail the structures you should keep).

Output format:

NOT VACCINATED OR
MAYBE

• / **vaccineStatus citizenID virusName**

The application will check the skip list associated with **virusName** and print a message as to whether the citizen with **citizenID** ID number has been vaccinated against **virusName.**

Output format:

NOT VACCINATED OR
VACCINATED ON 27-12-2020

Vacc / **vaccineStatus citizenID** The
application will check all skip lists (one for each virus), locate all citizen records with **citizenID ID number,** and print for each virus if vaccinated and the date of vaccination.

Output format: one line for each virus. Example:

```
COVID-19 YES 27-12-2020
SARS-1 NO
H1N1 YES 11-11-2020
```

• / **populationStatus [country] virusName date1 date2** If no **country** argument is
given, the application will print for **virusName** the number of citizens in each country who have been vaccinated within **[date1 ... date2]** and the percentage of the population of the country has been vaccinated. If a **country** argument is given, the application will print the **virusName disease,** the number of citizens who have been vaccinated and the percentage of the population of the country that has been vaccinated within **[date1 ... date2].** If there is a definition for **date1** there must also be a definition for **date2,** otherwise the **ERROR** error message will be printed to the user.

Output format: one line for each country. Example where no **country** argument is given:

```
GREECE 523415 5.02%
USA 358000000 10.8%
ISRAEL 3289103 38.0%
```

• / **popStatusByAge [country] virusName date1 date2** If no **country** argument is given, the application will print for **virusName** the number of vaccinations per age group in each country and the percentage of the age group vaccinated during the period **[date1 .. .date2].** If a **country** argument is given, the application will print the **virusName disease,** the number of vaccinations per age group and the percentage of the age group population vaccinated within **[date1 ... date2]** in the **country.** If there is a definition for **date1** there must also be a definition for **date2,** otherwise the **ERROR** error message will be printed to the user.

Output format: Example where no **country** argument is given:

```
GREECE
0-20 0 0%
20-40 18795 0.36%
40-60 64650 1.24%
60+ 439970 8.44%

ISRAEL
0-20 0 15%
20-40 18795 23%
40-60 64650 32.24%
60+ 4399070 90.44%
```

• / **insertCitizenRecord citizenID firstName lastName country age virusName YES / NO [date]**

The application will insert a new record with its data in the bloom filter and in the appropriate skip list related to the **virusName** virus. Only **YES** is accompanied by a **date.** If the citizen with **citizenID** ID has already been vaccinated against the **virusName** virus the application returns:

```
ERROR: CITIZEN 889 ALREADY VACCINATED ON 27-12-2020
```

Vacc / **vaccinateNow citizenID firstName lastName country age virusName** The application checks if the citizen with **citizenID** ID number has already been vaccinated against the **virusName** virus and if so, returns:

```
ERROR: CITIZEN 889 ALREADY VACCINATED ON 27-12-2020.
```

If not vaccinated, the application inserts in the bloom filter and in the appropriate skip list related to the **virusName** virus the entry: **citizenID firstName lastName country age virusName YES todays_date** where **todays_date** is the current date.

List / list- nonVaccinated **-Persons virusName** The application will access the appropriate virusName-related skip list **and** print all citizens who have not been vaccinated against **virusName.** Specifically, it will print **citizenID, firstName, lastName, country** and **age.**

Output format: one line for each citizen. Example:

```
125 Jon Dupont USA 76
889 John Papadopoulos GREECE 52
```

Exit / **exit**

Exit the application. Make sure you release all the free memory correctly.

Data structures

You can use C or C ++ to implement the application. However, you cannot use the Standard Template Library (STL). All data structures should be implemented by you. Make sure you only free up as much memory as needed, e.g. the following tactic is not recommended:

int countries [512]; // store up to 512 countries, but really we do not know how many

Also make sure that the memory is released correctly during the execution of your program and during the exit.

To complete the exercise you will need, among other things, to implement the following data structures.

1. A bloom filter per virus to be used by the application for a quick check if **not** a citizen is vaccinated.
2. A **vaccinated_persons** skip list per virus for the application to locate information about citizens who have been vaccinated. Skip lists will be sorted based on **citizenID.**
3. A **not_vaccinated_persons** skip list per virus for the application to locate information about unvaccinated citizens. Please note that only NO entries will be added to the **not_vaccinated_persons** skip list of a virus. In the sample input of the four entries above, **889 John Papadopoulos** will NOT be added to the **not_vaccinated_persons** skip list of **SARS-1.** Only if the application then reads a record that explicitly writes **NO** for **889 John Papadopoulos** for **SARS-1** will it be added to the **SARS not_vaccinated_persons** skip list

   1.
4. Your goal in this exercise is to have as *little data duplication as possible.* For example, the information provided by each record should only be stored once in memory, and any data structure that needs access to it will be accessed via pointers. (see Figure 1 for a possible layout proposal for some data structures).
5. When the application receives an **insertCitizenRecord** or **vaccinateNow** request, if the citizen is registered in the system as unvaccinated, the appropriate skip lists should be updated (eg, removed from **not_vaccinated_persons** skip list, added to **vaccinated_persons** skip list).

Make sure that for each sub-problem you need to solve when implementing the exercise, you use the most efficient algorithm or data structure. Any design decisions and choices you make during implementation should be described in README, in the deliverables.

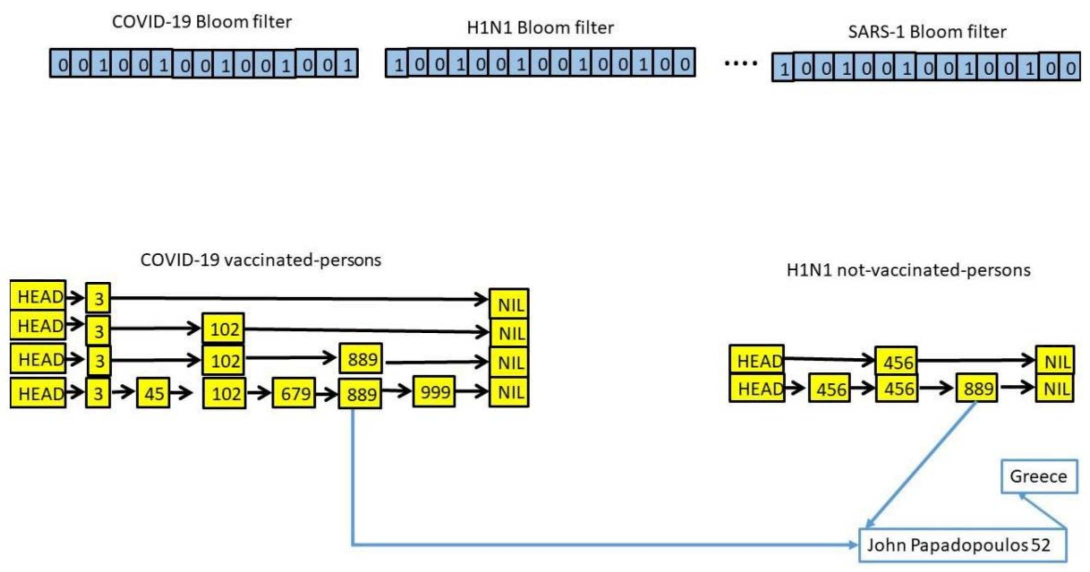

Figure 1: Example of Some Structures for vaccineMonitor application

B) The script testFile.sh (25%)

You will write a bash script that creates a test input file that you will use to debug your program. Of course during the development of your program you can use small and small files to debug. The script testFile.sh works as follows:

./testFile.sh virusesFile countriesFile numLines duplicatesAllowed

- virusesFile: a file with virus names (one per line)
- **countriesFile:** a file with country names (one per line)
- **numLines:** the number of lines in the file to be created - **duplicatesAllowed:**
if it is 0, then **citizenIDs** must be unique, otherwise
duplicate citizenIDs are allowed

The script does the
following: 1. Checks for input numbers 2.
Reads **virusesFile** and **countriesFile** files
3. Creates a file called **inputFile** and places **numLines** rows following the format of the input file described above. For **citizenID**
you can create random numbers with random length from 1 to 4 digits For the name and surname you can create random strings with random length from 3 to 12 characters. For country names, the script will be randomly selected by one of the **countriesFile.** For age it will randomly select an integer in the spectrum [1,120]. For virusNames, it will randomly select the script from one of the **virusesFile. 4.** If the **duplicatesAllowed** flag is enabled, it should also create some (random) records with duplicate **citizenIDs.**

Simplifying info / hints:
a) You can assume that all months have 30 days. b) You may find the $ RANDOM Bash function useful. (See, e.g., http://tldp.org/LDP/abs/html/randomvar.html)

Deliverable

- A brief and comprehensive explanation of the choices you have made in designing your program. 1-2 pages of ASCII text are enough. Include the explanation and instructions for compiling and running your program in a README file along with the code you will submit.

- The code you will submit should be **yours . It is forbidden to use code that does not has been written by you (this includes code from the Internet!)**
- All your work (source code, Makefile and README) in a tar.gz file named OnomaEponymoProject1.tar.gz. Be careful to submit only code, Makefile, README and not binaries.

- It would be good to have a .tar backup of your exercise just as it was submitted to an easily accessible machine (department server, private github repository, private cloud). - The correct submission of a correct tar.gz that contains the code of your exercise and whatever files are needed is your sole responsibility. **Licenses tar / tar.gz or tar / tar.gz that are wrong and are not extracted are not graded.**

Procedural -
For additional announcements, follow the course forum at piazza.com. The full address is https://piazza.com/uoa.gr/spring2021/k24/. Attendance at the Piazza Forum is a must.

- Your program should be written in C (or C ++). If you use C ++ you can not use the ready-made structures of the Standard Template Library (STL). In any case, your program should run on the Linux workstations of the Department. - Your program should compile the executable **(vaccineMonitor)** and have the same

names for the parameters **(-c, -b)** exactly as described in the pronunciation.
- Your password should consist of at least two (and preferably more) different files. Using separate compilation is imperative and your password should have a Makefile.

- Make sure you follow good software engineering practices when implementing the exercise. The organization, readability and existence of comments in the code are part of your rating.

- you must definitely register in eclass and give your name and your Unix user-id. This way we can know that you intend to submit this exercise and take the appropriate steps for the final submission of the exercise.

Other important remarks - The tasks are individual. - Anyone who submits / presents code that has not been written by him / her is reset to lesson.
- Although you are expected to discuss with friends and colleagues how you will try to solve the problem, copying code (of any kind) is not allowed. Anyone involved in copying code simply gets zero in the lesson. This applies to those involved regardless of who gave / received etc. We emphasize that you should take the appropriate measures so that your password is protected and not stored somewhere

is accessed by another user (eg the excuse "I put it on a github repo and probably got it from there" is not acceptable.)

- Planning exercises can be given with a delay of up to 3 days and with a penalty of 5% for each day of delay. Except for these 3 days, no exercises can be submitted.