HELLENIC REPUBLIC
National and Kapodistrian University of Athens
EST. 1837

DEPARTMENT OF
INFORMATICS & TELECOMMUNICATIONS

Disease Monitor

Anna-Aikaterini Kavvada - 1115201500050

March 24, 2020

Professor: Alexandros Ntoulas

1 Introduction

The project has been implemented in C language, using the jetbrain's editor Clion and tested both at my local linux and the departments linux machines, using ssh protocol for the connection.
Inside the tar.gz file the project is organised as shown below

- build: the executable, *diseaseMonitor*, after running the makefile can be found here
- header: all the header files associated with the project
- src: all the .c files associated with the aforementioned headers for the project, along with the main function
- makefile

2 Compilation and Running

In order to compile this project, we use a Makefile. Thus, while having an open terminal, type "make" in order to compile the program.Following, in order run the project, type:

./build/diseaseMonitor -p patientsRecordsFile -h1 diseaseHashTableNumOfEntries -h2 CountryHashTableNumOfEntries -b bucketSize

In case the user enters a wrong number of argumemts, the program prints in the stderr stream the error message and exits with exit code 1. If the arguments given are valid, the program procedes to opening, reading the txt with the patients' records and creates all the data structures needed.

3 Data Structures

In this project the following data structures were implemented and used

- Simply linked list: used to store the patient records and later as an auxiliary structure for the implementation of the binary heap tree for the queries */topk_Countries* and */topk_Diseases*.

- Hash Table: using this data structure, rwo hash tables were implemented for the project needs, one for the diseases and one for the countries. Each position of a hashtable has a pointer to its own bucket-list. Every bucket has size bucketSize. The bucketSize can be no less than 68bytes, since that is the minimum size in order to store just one value in each bucket. This scenario is not optimal, since we create really long bucket-lists. According to the needs a bucket is allocated expanding the list. A bucket, depending on the size we have defined from the command line can strore multiple entries. The above implementation is used to avoid the hashtable collisions that might appear. (This hashtable is based on an old implementation I have from an OS project)

- Red-Black Tree: For the balanced binary search tree I decided to implement an RBT. All the methods and structures are based on the algorithm for the RBTs from the book Introduction to Algorithms - 3rd Edition by Cormen, Leiserson, Rivest Stein (CLRS). The tree is sorted by each patients entry date and every node has a link to the node of the list where the patient at hand is stored.

- binary tree heap: a balanced binary tree was implemented for the needs of the **max-heap** with tree. The insertions of the tree are made from the leftest available node of each level. If the level is complete we move to the next level. For this on-the-fly heap, an auxialiary linked list was used, in order to collect all the data we needed to insert and sort with the heap. In order to get the max values and sustain the heap properties, we pop the root of the tree and call the heapify function, which is responsible for this job.

4 Assumptions

The size of all the record fields is 32 bytes, which are dynamically allocated during the creation of each patient. The bucketsize contains apart from the entries of the hashtable, the pointers and all values of the structure bucket. I think this was the most appropriate approach. In case the user enters an inappropriate bucketsize, the system prints the error and lets him enter a new bucketsize in order to continue. The patient's id is stored as a string in the system. That means that an id can be either numeric or alpharethmentic.