Task 2 – A more complex in-memory database

Extend the program / database you already developed in the previous task to support (a) registering students for courses, and (b) quickly searching for students by name. The desired functionality is described below. The program you submit must be saved in a single file named project2.c

Record of student enrollment in courses

A student's enrollments in the courses they desire must be recorded using a simply linked list, with each item in the list having the course code as an unsigned short. The student's course list must be sorted in ascending course code order. The head of the course list must be stored in the struct already used to store the rest of the student information (the existing structure must be extended appropriately).

Additional functions for course management The

management of the student's course list must be supported by corresponding (additional) functions (the exact description of the commands that the program must be able to read from its input, and the responses that the program must print at its output, given in detail in a separate section):

reg: Enrolling a student in a course, adding a corresponding new item to the course list. If there is no student with the given AEM or the student is already enrolled in the given course, an error is returned.

unreg: Delete a student from a course, by removing the corresponding item from the course list. If there is no student with the given AEM or the student is not enrolled in the given course, an error is returned.

isreg: Checking the student's course list to see if the student is enrolled in the course. If there is no student with the given AEM, an error is returned.

list-courses: Print the courses the student is enrolled in, in ascending order of course code. If there is no student with the given AEM, an error is returned.

For simplicity, we assume that a student can enroll in any course he wishes, and drop out of any course he wishes, at any time, without restriction.

Additional student search function A

separate student search function (see below), based on name (not AEM) should be added and implemented:

find-by-name: Search for a student by name. If the student is found, his details are printed (if there are many students with the same name, all the details are printed), otherwise a message is printed that it was not found.

Implementation requirements

- Each of the above functions must be implemented through a separate function, which will be called from the program main depending on the commands the program accepts.
- The functions reg, unreg, isreg must use the already existing student search function based on AEM that you implemented in the previous task.
- The use of global/static variables and the use of goto are prohibited.

Quickly search for students by name

To search for students by name faster, instead of implementing a (linear) search within the dynamic index table, use a hash table. As a hash function, use

the :

```
unsigned long hash(char *str) {

     unsigned long hash = 5381; int c;


     while ((c = *str++)) hash =
          ((hash << 5) + hash) + c;

     return hash;
}
```

Entries mapped to the same container should be stored in a doubly linked list with sentinel and sorted in ascending alphabetical order and, for nodes with the same name, in ascending AEM order. The list nodes must point directly to the corresponding student records, not to dynamic index table positions. Empty containers must be initialized appropriately, based on the conventions for a terminal doubly linked list (the empty list contains the terminal node).

The program takes as the third argument from the command line the initial size of the hash table, which is also the minimum allowed size of this table.

The re-hashing policy for dynamic re-hashing of the table is based on the load factor of the table. If after inserting a new record the degree of completeness is greater than or equal to 4, then the table is doubled. If after deleting a record the degree of completeness is 1, then the array is deduplicated, but never allowed to be smaller than the minimum size.

Changes to existing functions

The existing student registry management functions add, rmv, mod, find, print, clear and quit should be expanded so that the array is properly managed hashing alongside the original/base index table on records.

There is no need to check compatibility between the number of debtors courses and the number of courses in which the student is enrolled.

The rmv function should delete, with a corresponding freeing of memory, all the student data including their course list. The clear mode must delete with a corresponding release of memory all student registrations and index table and reset the hash table to its original size. THE quit function should delete all database data, completely freeing everything of dynamically allocated memory of program data structures.

The format of the above commands that the program accepts from its input, and the format of the messages printed by the program on its output and on its error output, remain as they are (see previous work), with the exception of the functions print, find (see below).

Also, a separate user command must be added to search for student based the name (see below).

Input/Output Formatting

Below is defined the format of the additional commands that the program reads from the input as well as the format of the messages that the program prints on its output.

Mandate	Entrance	output stdout (success)	output stdout (failure)
reg	g <aem> <course nr>	G-OK <aem> <course nr>	G-NOK <aem> or G-NOK <course nr>
unreg	u <aem> <course nr>	U-OK <aem> <course nr>	U-NOK <aem> or U-NOK <course nr>
isreg	i <aem> <course nr>	YES or NO	I-NOK <aem>
list-courses	l <aem>	* see below	L-NOK <aem>
find-by-name	n <name>	** see below	N-NOK <name>

If a command consists of several parts (accompanied by one or more parameters), these are separated by one or more ' (space) characters. Each command terminates with the '\n' character.

Each output message is preceded and followed by a '\n' character. If the message consists of several parts, they are separated by a single character ' (space).

* The list-courses command prints L-OK <name> and starting on the next line, one course record <course nr> per line.

** The find-by-name command prints the message N-OK <name> and starting from next line, for each student with this name, his AEM is printed, character ' (space), the number of courses he owes and character '\n'.

The print function should be changed so that it prints the contents of the hash table to the output of the program as follows:

It initially prints '\n' and '#' characters.

On the next line it prints: <size> <elements> <load factor> <largest bucket> where <size> is the array size, <elements> the total number of records, <load factor> the elements/size value to two decimals digits, and <largest bucket> the size of the largest bucket.

Then, starting from the next line, for each table position, the contents of the corresponding container are printed on one line, starting with <pos>, blank, <len>, where <pos> is the number of the position and <len> the size of the container, and then (if the container is not empty), for each student record a blank is printed and '['<aem> <name> <courses>']' (<courses> is the number of courses owed) . After the contents of the container are finished, two '\n' characters are printed.

Attention: The program must follow exactly the above specifications, otherwise the automated control of the program operation will fail.

Performance study

Study (again) the performance of your implementation, this time recording the number of steps and execution time of the find-by-name command, for student names that exist and student names that do not exist, when the database is 1,000, 10,000, and 100,000 records.

For each measurement, you will find on the course site a text file that in the first phase contains the add commands (to fill the base), and in the second phase the find-by-name commands (to search). You simply redirect the program's input to the file, and the error output (where the steps and seek time are printed) to a file where all the measurements will be stored. When each run is finished, extract from the file the values printed by the program, and process them, e.g., with the help of Excel, to extract the basic statistics (min, max, average, median).

Finally, you must make graph plots with the above results, which you will present/comment on in the examination of the work. Also, compare the results with what you recorded for the previous implementation for the case where the AEM search is performed on a fully sorted table and present your conclusions.

Code development stages

Stage 0:

Define the struct for the simple list of courses and implement separate functions for the basic operations of adding, removing, finding a course in the list, printing the contents of the list as directed by the utterance, and freeing all dynamically allocated list memory (in case where the student is removed from the register).

Double check the correctness of your code before proceeding.

Add to the student struct of the first assignment a field for the head of the student's course list.

Implement reg, unreg, isreg, list-courses and make any necessary changes to the original add, rmv, clear functions to properly manage the course list

Stage 1:

Change the student struct of the first assignment so that it can be used as a node type in a doubly linked list. Implement separate functions for the basic operations of adding, removing, finding a student in the list, printing the contents of the list as directed by the utterance, and freeing all dynamically allocated list memory. Tip: Think first about how you will use the hash table so that you can design the functions for the lists in a way that makes it easier for you.

Stage 2:

Construct the hash table and implement the basic operations of adding, finding, and removing entries in it. Make any necessary changes to the original add, rmv, clear functions to properly manage the hash table.

Stage 3:

Implement the hash function and the find-by-name function that will be called from within main when the user gives the corresponding command. Modify the print function accordingly.

Stage 4:

If you have time and enthusiasm for more, contact the teachers.