ECE 2230: Computer Systems Engineering **Fall 2021**
Machine Problem 1 **Due: 11:59 pm, Thursday, August 26**

The goal of this machine problem is to familiarize you with the basics of maintaining information with dynamic memory allocation and pointers. This assignment provides a simple example of data abstraction for organizing information in a sorted list. We will develop a simple abstraction with a few key interfaces and a simple underlying implementation using sequential arrays. In a subsequent programming assignment we will expand upon the interfaces and explore alternative implementations. We will refer to this abstract data type (ADT) as a *sequential list*.

Contact tracing is an important tool in managing disease outbreaks. Recently many major cell phone operating systems have implemented digital contact tracing or exposure notification. A key approach in these systems is to use ephemeral IDs. These IDs appear to be random numbers, do not provide personally identifiable information, and are typically changed on the order of every few minutes. A second key idea is to use decentralized reporting and processing. Consider Alice and Bob that each have phones that run these exposure notification protocols, or decentralized contact tracing. When they are close to each other they exchange their ephemeral IDs (there can be multiple different IDs depending on how long they are in proximity). If Alice gets sick, she sends her messages to a centralized source. But because the messages are randomized no information about Alice is released. Bob's phone occasionally collects codes from the centralized list to see if any of the codes on the centralized list are found in his private log. If Bob finds enough messages (meaning he was exposed long enough), his phone will alert him.

For this project, we model one portion of the exposure notification system (ENS). Assume another process gathers codes from Bluetooth or other wireless devices and creates a record of receiving contact information. Our project collects key data from the records and stores the data in a sorted order. Another task for your program is to take a list of ephemeral IDs and scan for the number of matches in the local list. If there are a significant number of matches, print an exposure notification message. An ENS may have to track a very large number of ephemeral IDs (perhaps many thousands) and store critical details about each one. So, we will also have tasks that can remove a specified ID or remove all IDs that are determined to be too old. For this project we will write the ADT for storing the records, and functions to search for exposures and also remove old data.

In this program, we will implement an abstract data type that allows the ENS to store the received information in a **sorted** list.

You are to write a C program that must consist of the following three files:

lab1.c	– contains the main() function, menu code for handling simple input and output used to test our ADT, and any other functions that are not part of the ADT.
enslist.c	– contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.
enslist.h	– contains the interface declarations for the ADT (e.g., constants, structure definitions, and prototypes).

Your program must use an array of pointers to C structures that contain information received for each ephemeral ID. Functions are specified to add, list, and remove IDs, and to search for matches against known codes.

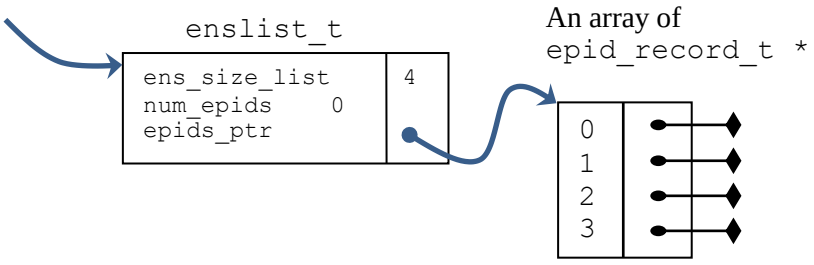
The ephemeral ID and additional information that is stored is represented with a C structure as follows:

```
struct enslist_t {
       struct epid_record_t **epids_ptr;
       int num_epids;     // current number of records in SAS list
       int ens_size_list;  // size of array given by user
};
struct epid_record_t {
    unsigned int ep_id;          // ephemeral ID
    unsigned int rec_num;        // EPID sequential record number
    unsigned int time_received; // time in seconds when record recevied
    int source_type;    // wireless type: 0 bluetooth 1 wifi 2 LTE 3 other
    unsigned int mac_addr;       // MAC address of source device
    int authenticated;  // true or false
    int privacy;         // mode 0 for none, 1 for encrypted
    int channel;         // 1-10
    float rssi;          // received signal strength in dB
    float band;          // received frequency band in GHz
};
```

The sequential list ADT must have the following interface:

```
struct enslist_t *enslist_construct(int);
void enslist_destruct(struct enslist_t *);
int enslist_add(struct enslist_t *, struct epid_record_t *);
struct epid_record_t *enslist_lookup(struct enslist_t *, int);
struct epid_record_t *enslist_access(struct enslist_t *, int);
struct epid_record_t *enslist_remove_epid(struct enslist_t *, int);
int enslist_remove_old(struct enslist_t *, int);
int enslist_remove_smallest(struct enslist_t *);
int enslist_list_size(struct enslist_t *);
int enslist_id_count (struct enslist_t *);
```

`enslist_construct` should return a pointer to the header block for the data structure. The data structure includes an array of pointers where the size of the array is equal to the value passed in to the function. (The size is given at runtime with the CREATE command. See below.) Each element in the array is defined as a pointer to a structure of type `epid_record_t`. Each pointer in the array should be initialized to NULL.



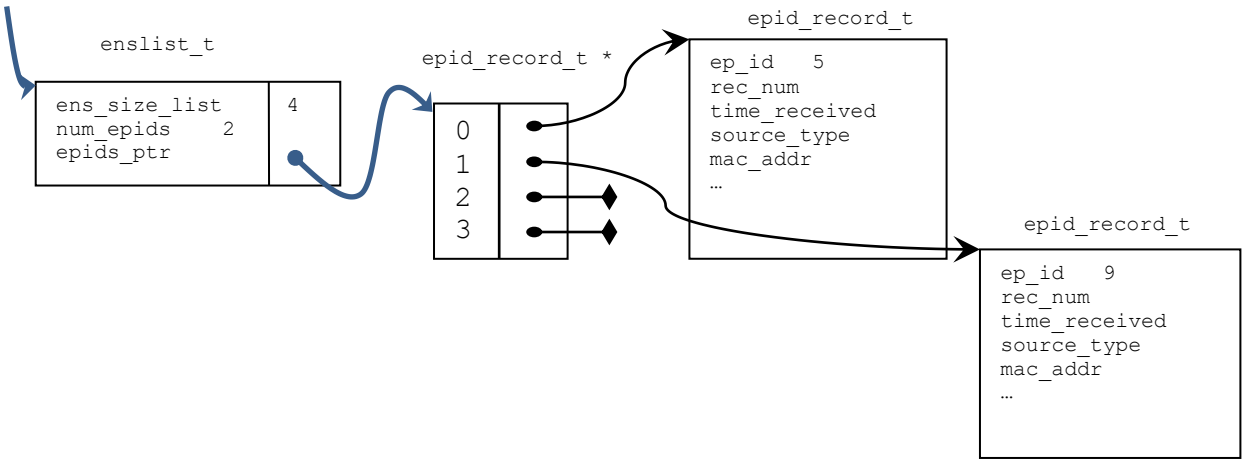
`enslist_destruct` should free all `epid_record_t` memory blocks in the list, free the array of type `epid_record_t *`, and finally free the memory block of type `enslist_t`.

`enslist_add` should take a `epid_record_t` memory block (that is already populated with input information) and insert it into the list such that the list is ordered using the `ep_id` and the list is sequential with no empty gaps between entries in the list. That is, the record with the lowest `ep_id` should be found at index position 0, the next lowest `ep_id` at index position 1, etc. If an entry already exists for a record

with the same ep_id, assume that this is an update of the record. The old information should be removed from the list, and the new information should be added and the return value is 2. If a new item is added to the list then return 1. If the list is full, and the record is not a duplicate, then remove the record in the list with the smallest rec_num and then add the new record in its correct sorted position and return 3. If this function is called but the list does not yet exist, then simply return with a value of 0. It is the responsibility of the program calling enslist_add to react to this error condition (e.g., if the memory block was not inserted into the list it must be freed). In summary, the return values for enslist_add are

return value	enslist_add **Action**
1	inserted new record into list
2	replaced record in list
3	List is full. Removed smallest rec_num and added new record
0	Error, list does not exist!

For example, the figure below shows the state of the list after ep_id 9 and then ep_id 5 were added to the list.



enslist_lookup should find the epid_record_t memory block in the list with the specified ep_id and return a pointer to the record within the list. If the ep_id is not found, then return NULL.

enslist_access should return a pointer to the epid_record_t memory block that is found in the list index position specified as a parameter. If the index is out of bounds or if no record is found at the index position, the function returns NULL.

enslist_remove_epid should remove and return the record with the given EPID and shift all the records at higher index values to one smaller value. The resulting list should still be sequential and sorted with **no** gaps between entries in the list. If the EPID is not found in the list, then return NULL.

enslist_remove_old should remove all records with a time_received value smaller than the specified value. After a record is removed, the gap in the list must be closed by shifting all the records at higher index values to one smaller value. The resulting list should still be sequential and sorted with **no** gaps between entries in the list. The return value should be the number of records that are removed.

`enslist_remove_smallest` should remove the record with the smallest `rec_num` value found in the list. After a record is removed, the gap in the list must be closed by shifting all the records at higher index values to one smaller value. The resulting list should still be sequential and sorted with **no** gaps between entries in the list. The return value should be EPID of the record that was removed or -1 if the list was empty so no record could be removed.

`enslist_list_size` should return the size of the array, i.e., `ens_size_list`. If this function is called before the list has been created, then the array does not yet exist, and return a value equal to zero.

`enslist_id_count` should return the current number of EPID records stored in the list, i.e., `num_epids`. If the list has not yet been created, return zero.

The template for the file `lab1.c` provides the framework for input and output and testing the sequential list of EPID information. The code reads from standard input the commands listed below. The template contains the only prints to standard output that you are permitted to use. You will expand the template code in `lab1.c` to call functions found in **`enslist.c`**. Based on the return information you will call the appropriate print statements. It is critical that you do not change the format of either the input or output as our grading depends on your program reading this exact input pattern and producing exactly the output defined in `lab1.c` and nothing else. These are the input commands:

`CREATE` `list-size`
`ADDID` `epid`
`QUERY` `epid`
`CLEAROLD` `time`
`TRIM`
`MATCH` `threshold`
`STATS`
`PRINT`
`QUIT`

The **`CREATE`** command constructs a new list with the given size. If a list already exists when this command is called, then the old list must first be destructed and then a new empty list is created in its place. The **`ADDID`** command creates a dynamic memory block for the `epid_record_t` structure using `calloc()` and then prompts for each field of the record, one field on each line and in the order listed in the structure. The `lab1.c` file provides the function to collect the input. Based on the return value of the `enslist_add` function, print the corresponding output message. The **`QUERY`** command searches for the matching record and, if found, prints the record. The **`CLEAROLD`** command must remove each record from the list that has a `time_received` value smaller than the `time` value. Note there can be multiple records with the same time since the granularity of the time received field is coarse at one second. The **`TRIM`** command must remove the record with the smallest `rec_num` value. The **`MATCH`** command is followed by a list of EPIDs. The list ends with a -1. If the corresponding EPID is found in the list it must be printed and removed. After all EPIDs have been checked, print a warning message if the number of matches is greater than or equal to the `threshold`. The **`STATS`** command prints the number of records in the list and the size of the array. The **`PRINT`** command prints each record in the list in sorted order by EPID. Finally, the **`QUIT`** command frees all the dynamic memory and ends the program. If commands other than **`CREATE`** are given before the first **`CREATE`** command, make sure your code does not crash, but instead just reports the same failure message as if the command was unsuccessful.

To facilitate grading the output for each command must be formatted exactly as specified by the **`printf()`** commands in the `lab1.c` template file. You cannot modify the format or text in the print commands, and you cannot add any new prints (in the version you submit for grading).

Notes

1. The ten `enslist_*` function prototypes must be listed in `enslist.h` and the corresponding functions **must** be found in the `enslist.c` file. Code in `lab1.c` can call a function defined in `enslist.c` **only** if its prototype is listed in `enslist.h`. You can include additional functions in `lab1.c` (e.g., functions to support gathering an ephemeral ID record or printing a record are already provided). You can also add other "private" functions to `enslist.c`, however, these private functions can only be called from within other functions in `enslist.c`. The prototypes for your private functions **cannot** be listed in `enslist.h`. Code in `lab1.c` **cannot** call any of your private functions. Code in `lab1.c` is **not** permitted to access **any** of the members in `struct enslist_t` (i.e., `ens_size_list`, `num_epids`, or `epids_ptr`), instead code in `lab1.c` **must** use the sequential list functions `enslist_*` as defined in `enslist.h` as to **only** way to access details of the list.

Note we are using the principle of *information hiding*: code in `lab1.c` does not "see" any of the details of the data structure used in `enslist.c`. The only information that `lab1.c` has about the ENS list data structure is found in `enslist.h` (and any "private" functions you add to `enslist.c` are not available to `lab1.c`). The fact that `enslist.c` uses an array of pointers is unimportant to `lab1.c`, and if we redesign the data structure no changes are required in `lab1.c` (including PRINT). However, notice that `enslist.c` does need to read three members of the `epid_record_t` structure (i.e., `ep_id`, `rec_num`, and `time_received`). If we decide to store different types of records, we have to re-write the part of `enslist.c` that use these values. In future machine problems we will study designs that allow us to hide the details of the records from the data structure, so we can reuse the data structure for any type of record.

You are not permitted to access any of the structure members in `enslist_t` or index into the `epid_record_t *` array directly in `lab1.c`. (For example, the characters "`epids`" must not be found anywhere in `lab1.c`. Your submission will not be accepted if the characters `epids_ptr` appear in `lab1.c`.)

2. The file `lab1.c` uses the C functions `fgets()` and `sscanf()` to read input data. Do **not** use `scanf()` for any input because it can lead to buffer overflow problems and causes problems with the end-of-line character. See the `lab1.c` template.

You do not need to check for errors in the information the user is prompted to input for the `epid_record_t` record. However, you must extensively test your code that it can handle any possible combinations of CREATE, ADDID, QUERY, CLEAROLD, TRIM, MATCH, STATS, and PRINT. For example, you code must handle a request to delete, print, or look in an empty list or even before a list has been created.

3. Recall that you compile your code using:

```
gcc -Wall -g lab1.c enslist.c -o lab1
```

Your code must be able to pipe my example test scripts as input using <. Collect output in a file using > For example, to run do

```
./lab1 < testinput.txt > testoutput
```

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. (OS X users must verify that there are no warnings on a machine running Ubuntu.)

An example `testinput.txt` and `expectedoutput.txt` files are provided. When you run your code on the `testinput.txt` file, your output must be identical to the file `expectedoutput.txt`. You can verify this using

```
meld testoutput.txt expectedoutput.txt
```

If you don't have access to a graphical display you can use `diff` in place of `meld`.

The tests in `testinput.txt` are incomplete! You must develop more thorough tests (e.g., attempt to delete from an empty list, or insert into a list that is already full). Part of your submission it to write your own version of `testinput.txt` that documents are more complete sets of tests.

To receive credit for this assignment your code must compile and at a minimum evaluate the provided `testinput.txt` and identically match the `expectedoutput.txt`. Code that does not compile or crashes before completing the example input will not be accepted or graded. If your code is not a perfect match you must contact the instructor or TA and fix your code. Code that correctly handles the example input but otherwise fails many other cases will be scored at 50% or less.

4. Be sure that your program does not have any memory leaks. That is, all dynamically allocated memory must be freed before the program ends.

We will test for memory leaks with `valgrind`. You execute `valgrind` using

```
valgrind --leak-check=yes ./lab1 < testinput
```

The last line of output from `valgrind` must be:

```
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: x from y)
```

You can ignore the values *x* and *y* because suppressed errors are not important and are hidden from you. In addition the summary of the memory heap must show

```
All heap blocks were freed -- no leaks are possible
```

5. All code files (`lab1.c`, `enslist.c`, `enslist.h`) and your own version of a testinput.txt file must be submitted to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#1. The 223-1 identifies you as a member of Section 1 (the only section this semester). The #1 identifies this as the first assignment. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes that does not report any errors. If you don't get a confirmation email or there are errors listed in the email, your submission was not successful. You must include all your files as an attachment. The email must be formatted as text only (no html features). You can make more than one submission but we will only grade the final submission. A re-submission must include all files. You cannot add files to an old submission.

See the ECE 2230 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.