ECE 2230: Computer Systems Engineering **Fall 2018**
Machine Problem 1 **Due: 11:59 pm, Thursday, August 30**

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*.

The Federal Communications Commission (FCC) has recently approved a new wireless system called Citizens Broadband Radio Service (CBRS)[1]. Radio frequency (RF) spectrum is in high demand and nearly all practical frequencies have already been assigned to various systems. However, some of these old systems do not make efficient utilization of their frequency allocation; we refer to users of these old systems as *incumbent users*. Recent research has shown that it is possible to share the spectrum with legacy users by allowing *secondary users* to utilize the same frequencies if no legacy users are currently active in a local region. The CBRS system is defining standards on how to support *spectrum sharing* among incumbent and secondary users. One critical piece of the CBRS system is the spectrum access system (SAS) that monitors for the presence of incumbent users. If no incumbent users are present in a local region then the SAS allows secondary users to access some of the frequency channels normally assigned to the incumbent users. However, if an incumbent user is detected on a channel, then the SAS must notify all secondary users that they must vacate this channel. The SAS can reassign a secondary user to a different frequency channel that is not being utilized by an incumbent user. Or, the SAS may require the secondary user to shut down if there are no free channels available. Therefore, it is essential that the SAS register all secondary users so that it is possible to inform secondary users when they need to change their operating frequency.

The SAS may have to track a very large number of secondary users (perhaps many thousands) and store critical details about each one. In this program, we will implement an abstract data type that allows the SAS 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.
saslist.c	– contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.
saslist.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 secondary user.

The secondary user information that is stored is represented with a C structure as follows:

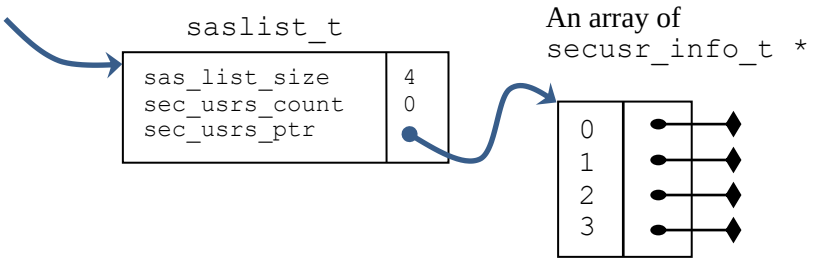
[1] S. Bhattarai, J.-M. Park, B. Gao, K. Bian, and W. Lehr, "An Overview of Dynamic Spectrum Sharing: Ongoing Initiatives, Challenges, and a Roadmap for Future Research," IEEE Trans. Cogn. Commun. Netw., vol. 2, no. 2, pp. 110–128, 2016.

```
struct saslist_t {
       int sas_list_size;  // size of array given by user
       int sec_usrs_count;     // current number of records in SAS list
       struct secusr_info_t **sec_usrs_ptr;
};
struct secusr_info_t {
       int su_id;          // secondary user's ID number
       int ip_address;     // mobile's IP address
       int access_point;   // IP address of access point that is connected to mobile
       int authenticated;  // true or false
       int privacy;        // mode 0 for none, 1 encrypted
       float band;         // 3.5 for the CBRS frequency bands (in GHz)
       int channel;        // 1-10
       float data_rate;    // in bits per second
       int time_received;  // time in seconds that information last updated
};
```

The sequential list ADT must have the following interface:

```
struct saslist_t *saslist_construct(int);
void saslist_destruct(struct saslist_t *);
int saslist_add(struct saslist_t *, struct secusr_info_t *);
int saslist_lookup(struct saslist_t *, int);
struct secusr_info_t *saslist_access(struct saslist_t *, int);
struct secusr_info_t *saslist_remove(struct saslist_t *, int);
int saslist_arraysize(struct saslist_t *);
int saslist_number_entries(struct saslist_t *);
```

`saslist_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 `secusr_info_t`. Each pointer in the array should be initialized to NULL.



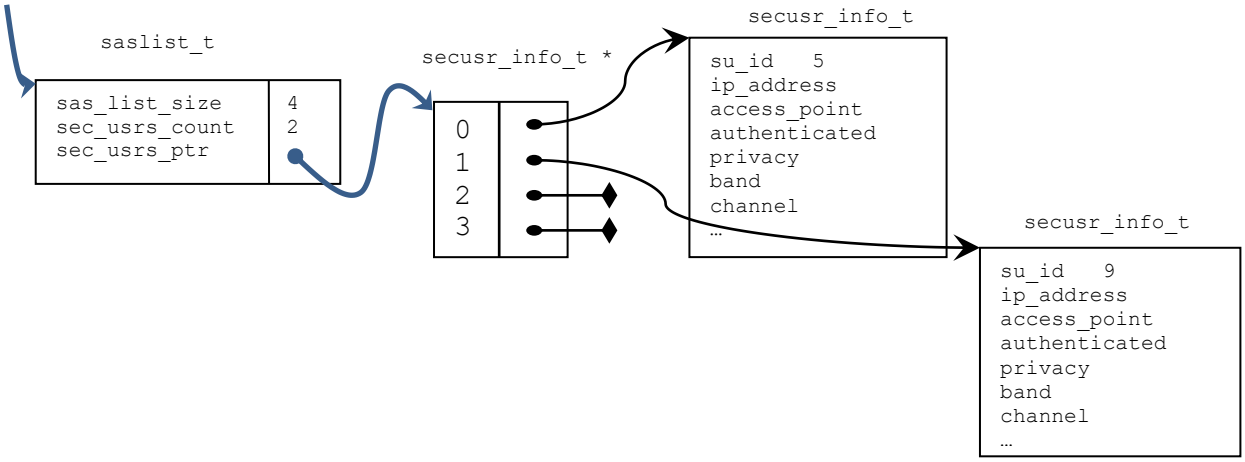
`saslist_destruct` should free all `secusr_info_t` memory blocks in the list, free the array of type `secusr_info_t *`, and finally free the memory block of type `saslist_t`.

`saslist_add` should take a `secusr_info_t` memory block (that is already populated with input information) and insert it into the list such that the list is ordered using the `su_id` and the list is sequential with no empty gaps between entries in the list. That is, the record with the lowest `su_id` should be found at index position 0, the next lowest `su_id` at index position 1, etc. If an entry already exists for a record with the same `su_id`, assume that this is an update of the secondary user information. The old information should be removed from the list, and the new information should be added and the return value is 0. 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 do not modify the list and return -1. It is the responsibility of the program calling `saslist_add` to react to the error condition (e.g., if the memory block was not inserted into the list it must be freed). In summary, the return values for `saslist_add` are

return value	`saslist_add` **Action**
0	replaced record in list
1	inserted new record into list
-1	List is full or has not been created, record not inserted but returned

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



`saslist_lookup` should find the `secusr_info_t` memory block in the list with the specified `su_id` and return the index position of the record within the list. If the `su_id` is not found, then return -1.

`saslist_access` should return a pointer to the `secusr_info_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 secondary user record is found at the index position, the function returns NULL.

`saslist_remove` should remove and return the record at the given index position and shift all the records at higher index values to one smaller value. The resulting list should still be sequential with **no** gaps between entries in the list. If the index position is invalid, then return NULL.

`saslist_arraysize` should return the size of the array, i.e., `sas_list_size`. 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.

`saslist_number_entries` should return the current number of secondary user records stored in the list. 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 secondary user 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 **`saslist.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
ADDSU
DELSU addr
CLEARCH channel-num
CHANGECH old-channel-num new-channel-num
LISTCH channel-num
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 **ADDSU** command creates a dynamic memory block for the `secusr_info_t` structure using `malloc()` and then prompts for each field of the record, one field on each line and in the order listed in the structure. Based on the return value of the `saslist_add` function, print the corresponding output message. The **DELSU** command searches for the matching record and, if found, removes the record from the list and frees the memory. The **CLEARCH** command must remove each secondary user from the list that is assigned to the frequency channel given by `channel-num`. The **CHANGECH** command must change the channel for all secondary users that are currently using channel `old-channel-num` to the new channel `new-channel-num`. The **LISTCH** command must print the record for all secondary users that are assigned to the frequency channel given by `channel-num`. The **STATS** command prints the number of records in the list and the size of the array. The **PRINT** command prints each record if there are one or more records in the list. 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.

Notes

1. The eight `saslist_*` function prototypes must be listed in `saslist.h` and the corresponding functions **must** be found in the `saslist.c` file. Code in `lab1.c` can call a function defined in `saslist.c` **only** if its prototype is listed in `saslist.h`. You can include additional functions in `lab1.c` (e.g., functions to support gathering secondary user information or printing the list are already provided). You can also add other "private" functions to `saslist.c`, however, these private functions can only be called from within other functions in `saslist.c`. The prototypes for your private functions **cannot** be listed in `saslist.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 saslist_t` (i.e., `sas_list_size`, `sec_usrs_count`, or `sec_usrs_ptr`), instead code in `lab1.c` **must** use the sequential list functions `saslist_*` as defined in `saslist.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 `saslist.c`. The only information that `lab1.c` has about the SAS list data structure is found in `saslist.h` (and any "private" functions you add to `saslist.c` are not available to `lab1.c`). The fact that `saslist.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 `saslist.c` does need to read one member of the `secusr_info_t` structure (i.e., `su_id`). If we decide to store different types of records, we have to re-write the part of `saslist.c` that uses `su_id`. 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 `saslist_t` or index into the `secusr_info_t *` array directly in `lab1.c`. (For example, the characters "`sec_usrs_`" must not be found anywhere in `lab1.c`. Your submission will not be accepted if the characters `sec_usrs_` 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 `secusr_info_t` record. However, you must extensively test your code that it can handle any possible combinations of CREATE, ADDSU, DELSU, CLEARCH, CHANGECH, LISTCH, 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 saslist.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
```

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

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 and a test script that you develop 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. If you don't get a confirmation 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 223 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.