ECE 223: Computer Systems Engineering **Fall 2013**
Machine Problem 1 **Due: 11:59 pm, Thursday, August 29**

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*.

Consider the access-point manager for an 802.11 wireless network (or WiFi network). The access-point manager keeps track of all mobile devices that are within range of any of the base stations that are under the administrative control of the manager. The manager may have to track a very large number of mobile devices (as high as a few thousand) and the manager must store some control information that it receives from each base station concerning each mobile device. In this program, we will implement an abstract data type that allows the manager 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.
wifilist.c	– contains the ADT code for our sequential list. The interface functions must be exactly defined as described below.
wifilist.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 mobile WiFi device. The first step of your program should prompt the user for the size of the array, and this value is used in the construction function to build the data structure.

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

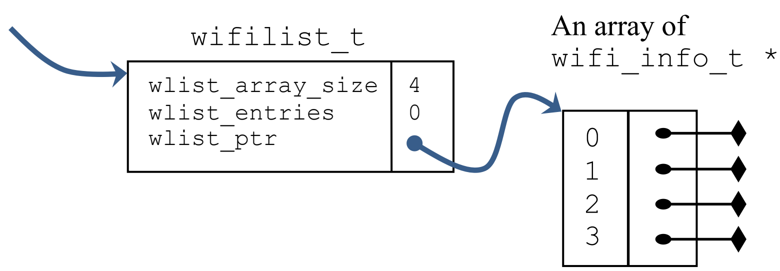
```
struct wifilist_t {
      int wlist_array_size;  // size of array given by user
      int wlist_entries;     // current number of records in WiFi list
      struct wifi_info_t **wlist_ptr;
};
struct wifi_info_t {
      int eth_address;    // mobile's Ethernet address
      int ip_address;     // 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 for WEP, 2 for WPA, 3 for WPA2
      int standard_letter;// a, b, e, g, h, n, or s.
                          //   Convert letter to integer with a=1, b=2, etc.
      float band;         // 2.4 or 5.0 for the ISM frequency bands (in GHz)
      int channel;        // 1-11 for 2.4 GHz and 1-24 for 5 GHz
      float data_rate;    // 1, 2, 5.5, 11 for 2.4 GHz and
                          // 6, 9, 12, 18, 24, 36, 48, 54 for 5 GHz
      int time_received;  // time in seconds that information last updated
};
```

The sequential list ADT must have the following interface:

```
struct wifilist_t *wifilist_construct(int);
```

```
void wifilist_destruct(struct wifilist_t *);
int wifilist_add(struct wifilist_t *, struct wifi_info_t *);
struct wifi_info_t *wifilist_lookup(struct wifilist_t *, int);
struct wifi_info_t *wifilist_access(struct wifilist_t *, int);
struct wifi_info_t *wifilist_remove(struct wifilist_t *, int);
int wifilist_arraysize(struct wifilist_t *);
int wifilist_number_entries(struct wifilist_t *);
```

`wifilist_construct` should return a pointer to a structure that includes an array with initial size equal to the value passed in to the function. Each element in the array is defined as a pointer to a structure of type `wifi_info_t`. Each pointer in the array should be initialized to NULL.

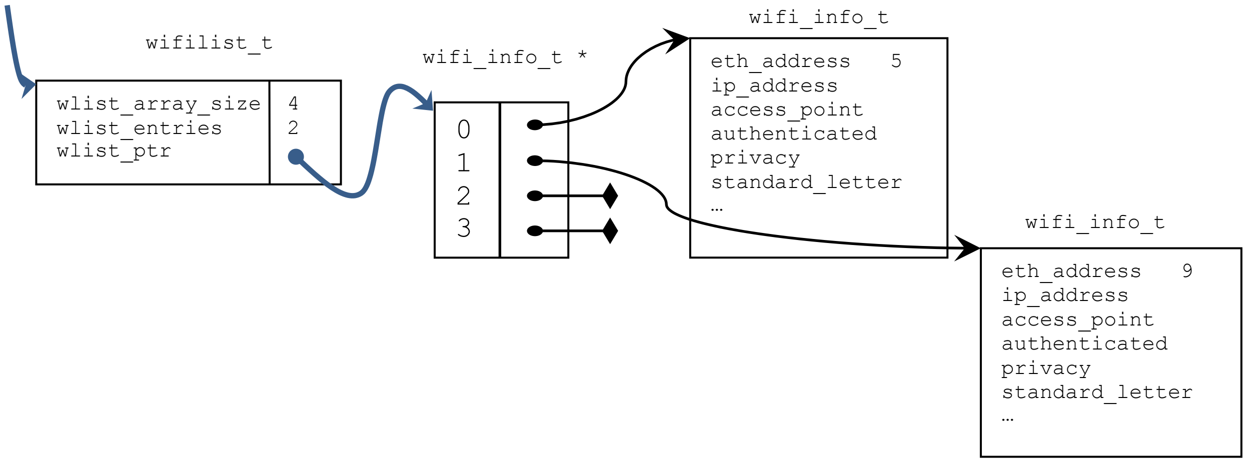


`wifilist_destruct` should free all `wifi_info_t` memory blocks in the list, free the array of type `wifi_info_t *`, and finally free the memory block of type `wifilist_t`.

`wifilist_add` should take a `wifi_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 `eth_address` and the list is sequential with no empty gaps between entries in the list. That is, the record with the lowest `eth_address` should be found at index position 0, the next lowest `eth_address` at index position 1, etc. If an entry already exists for a record with the same `eth_address`, assume that this is an update of the WiFi 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 `wifilist_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 `wifilist_add` are

return value	`wifilist_add` **Action**
0	replaced record in list
1	inserted new record into list
-1	List is full, recorded not inserted but returned

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



`wifilist_lookup` should find the `wifi_info_t` memory block in the list with the specified `eth_address` and return a pointer to the record within the list. If the `eth_address` is not found, then return NULL.

`wifilist_access` should return a pointer to the `wifi_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 WiFi record is found at the index position, the function returns NULL.

`wifilist_remove` should search for a WiFi in the list with the specified `eth_address` and if found remove it from the list and return the memory block. The resulting list should still be sequential with **no** gaps between entries in the list. If no matching WiFi record is found, the function returns NULL.

`wifilist_arraysize` should return the current size of the array, i.e., `wlist_array_size`.
`wifilist_number_entries` should return the current number of WiFi records stored in the list, i.e., `wlist_entries`.

In the `lab1.c` file you will write a program to test the sequential list of WiFi information. For purposing of testing our ADT, the program should read from the command line one of these commands:

```
ADD eth-addr
FIND eth-addr
DEL eth-addr
STATS
PRINT
QUIT
```

The first word of each command must be formatted exactly as shown. You program should be able to handle a line that contains any number of strings. If the line does not exactly match the format of one of the above commands, the line must be simply printed (precede the printing of the line with the character # to indicate the line was not evaluated as a command). The `ADD` command should create a dynamic memory block for the `wifi_info_t` structure using `malloc()` and then prompt 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 `wifilist_add` function, print the output message given in the table below, where *x* is the `eth-addr`. The `FIND` command must print the information for the WiFi record for which the `eth_address` matches the `eth-addr` if it is found. The `DEL` command searches for the matching record and, if found, removes the record from the list and frees the memory. 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.

To facilitate grading the output for each command must be formatted exactly as follows

Command	Output
ADD *x*	Inserted: *x*
	Updated: *x*
	Rejected: *x*
DEL *x*	Removed: *x*
	Did not remove: *x*
FIND *x*	Did not find: *x*
	eth addr: *x* (other information in record printed on one line) Time: *t*
STATS	Number records: *y*, Array size: *z*
PRINT	List empty
	y records Record 1 eth addr: *x* (other information in record printed on one line) Time: *t* Record 2 eth addr: *x* (other information in record printed on one line) Time: *t* (continue to print details of each record)
QUIT	Goodbye

Notes

1. The eight wifilist_* function prototypes must be listed in wifilist.h and the corresponding functions **must** be found in the wifilist.c file. Code in lab1.c can call a function defined in wifilist.c **only** if its prototype is listed in wifilist.h. You can include additional functions in lab1.c (such as a function to support gathering WiFi information or printing the list). You can also add other "private" functions to wifilist.c, however, these private functions can only be called from within other functions in wifilist.c. The prototypes for your private functions cannot be listed in wifilist.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 wifilist_t (i.e., wlist_array_size, wlist_entries, or wlist_ptr), instead code in lab1.c **must** use the sequential list functions wifilist_* as defined in wifilist.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 wifilist.c. The only information that lab1.c has about the WiFi list data structure is found in wifilist.h (and any "private" functions you add to wifilist.c are not available to lab1.c). The fact that wifilist.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 `wifilist.c` does need to read one member of the `wifi_info_t` structure (i.e., `eth_address`). If we decide to store different types of records, we have to re-write the part of `wifilist.c` that uses `eth_address`. 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.

2. Use 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. For example:

```
#include <stdio.h>
#define MAXLINE 256

char line[MAXLINE];
char command[MAXLINE];
char restofline[MAXLINE];
int num_items;
int sock_id;

while (fgets(line, MAXLINE, stdin) != NULL) {
       num_items = sscanf(line, "%s %d %s", command, &sock_id, restofline);
       if (num_items == 1 && strcmp(command, "QUIT") == 0) {
             /* found exit */
       } else if (num_items == 2 && strcmp(command, "LOOK") == 0) {
             /* more tests on command */
       } else { /* did not match any other test */
             printf("# %s", line);
       }
}
```

You do not need to check for errors in the information the user is prompted to input for the `wifi_info_t` record. However, you must extensively test your code that it can handle any possible combinations of ADD, FIND, DEL, STATS, and PRINT. For example, you code must handle a request to delete, print, or look in an empty list.

3. Recall that you compile your code using:

```
gcc -Wall -g lab1.c wifilist.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 > 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 they are using the latest version of gcc or make a final check on a machine running Ubuntu.)

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, a test script, and a test log must be turned in by email to assign@assign.ece.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. The email program you use to send your code needs to use your Clemson address as the “from email address” (and not your google address). WebMail works well. 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.

6. Turn in a paper copy of all code files at the start of the first class meeting following your submission to the assign server. Print in landscape mode, two columns, with a small monospaced font size to save paper.

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.