ECE 2230: Computer Systems Engineering **Fall 2018**
Machine Problem 2 **Due: 11:59 pm, Thursday, September 13**

The goal of this machine problem is to build an implementation of a list ADT which includes a set of fundamental procedures to manipulate the list. We will implement the list using two-way linked lists, and define a general-purpose interface that will allow us to use this list ADT for the remainder of the semester. Utilizing the basic list ADT, we will extend the MP1 model of a CBRS spectrum access system (SAS) for saving and managing secondary user information, including the assigned channel number.

A key element in this assignment is to develop a well-designed abstraction for the list so that we can return to this interface and replace it with either (a) a different application that needs access to a list or (b) change the underlying mechanism for organizing information in a sorted list (e.g., other than a two-way linked list).

You are to write a C program that will maintain the **two** lists of secondary user records. One list is sorted based on the secondary user ID and has a maximum number of records that can be inserted. The other list is unsorted and has no limit on the size of the list. The list ADT provides a general list package that is used for both of the lists. The code **must** consist of the following files:

`lab2.c`	– contains the main() function, menu code for handling simple input and output, and any other functions that are not part of the ADT.
`sas_support.c`	– contains subroutines that support handling of secondary user records.
`list.c`	– The two-way linked list ADT. The interface functions must be defined exactly as described in `list.h`. You are allowed include additional functions but the interface cannot be changed.
`sas_support.h`	– The data structures for the specific format of the secondary user records and the prototype definitions.
`list.h`	– The data structures and prototype definitions for the list ADT.
`datatypes.h`	– Key definitions of the secondary user structure
`makefile`	– Compiler commands for all code files

Part 1: The two-way linked list ADT

Your program must use a two-way linked list to implement the list ADT. A list is an object, and `list_construct()` is used to generate a new list. There can be any number of lists active at one time. For a specific list, there is no limit on the maximum size, and elements can be added to the list until the system runs out of memory. Your program must not have a memory leak, that is, at no time should a block of memory that you allocated be inaccessible.

The list ADT is designed so that it does not depend on the type of data that is stored except through a very limited interface. Namely, to allow the list ADT to be used with a specific instance of a data structure, we define the data structure in a header file called `datatypes.h`. This allows the compiler to map a specific definition of a structure to a type we call `data_t`. All of the procedures for the list ADT operate on `data_t`. In future programming assignments we will reuse the list ADT by simply modifying the `datatypes.h` file and then recompiling.

```
/*
/* datatypes.h
 *
```

```
 * The data type that is stored in the list ADT is defined here.  We define a
 * single mapping that allows the list ADT to be defined in terms of a generic
 * data_t.
 *
 * data_t: The type of data that we want to store in the list
 */

typedef struct secusr_info_tag {
       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
} su_info_t;

/* the list ADT works on packet data of this type */
typedef su_info_t data_t;
```

The list ADT must have the following interface, defined in the file `list.h`. We refer to this as the *public* information.

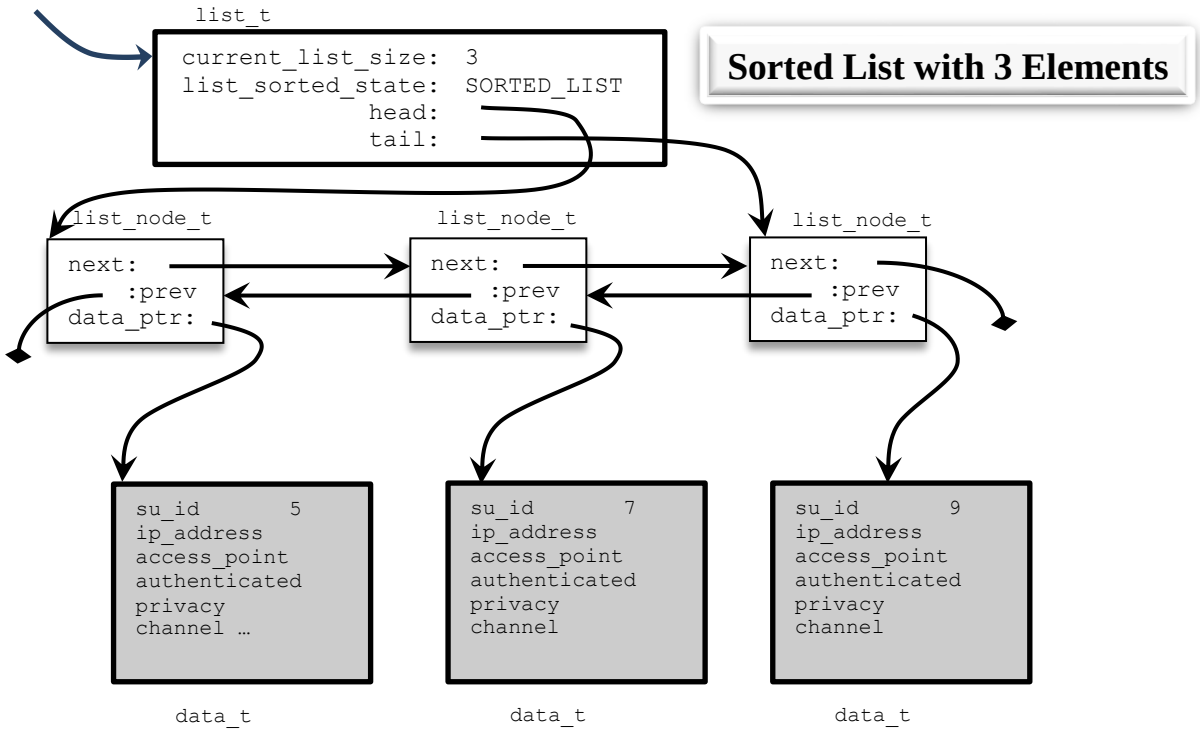
```
/* list.h
 *
 * Public functions for two-way linked list
 *
 * Do not change any of the code in this file.  If you do, you
 * must get permission from the instructor.
 */

typedef struct list_node_tag {
    // private members for list.c only
    data_t *data_ptr;
    struct list_node_tag *prev;
    struct list_node_tag *next;
} list_node_t;

typedef struct list_tag {
    // private members for list.c only
    list_node_t *head;
    list_node_t *tail;
    int current_list_size;
    int list_sorted_state;
    // Private method for list.c only
    int (*comp_proc)(const data_t *, const data_t *);
} list_t;

/* public definition of pointer into linked list */
typedef list_node_t * IteratorPtr;
typedef list_t * ListPtr;

/* public prototype definitions for list.c */

/* build and cleanup lists */
ListPtr list_construct(int (*fcomp)(const data_t *, const data_t *));
void list_destruct(ListPtr list_ptr);

/* iterators into positions in the list */
IteratorPtr list_iter_front(ListPtr list_ptr);
```

```
IteratorPtr list_iter_back(ListPtr list_ptr);
IteratorPtr list_iter_next(IteratorPtr idx_ptr);

data_t * list_access(ListPtr list_ptr, IteratorPtr idx_ptr);
IteratorPtr list_elem_find(ListPtr list_ptr, data_t *elem_ptr);

void list_insert(ListPtr list_ptr, data_t *elem_ptr, IteratorPtr idx_ptr);
void list_insert_sorted(ListPtr list_ptr, data_t *elem_ptr);

data_t * list_remove(ListPtr list_ptr, IteratorPtr idx_ptr);

int list_size(ListPtr list_ptr);
```

The details of the procedures for the list ADT are defined in the comment blocks of the `list.c` template file. You can define additional *private* procedures to support the list ADT, but you must clearly document them as extensions.

When the list ADT is constructed, it must have the initial form as show in the figure below.

The list ADT supports two variations on a list. A list can be maintained in either sorted or unsorted order, depending on the procedures that are utilized. Multiple lists can be concurrently maintained. Here is an example of the list ADT data structure when three `su_info_t`'s have been added to the sorted list.



It is critical that the `list.c` procedures be designed to not depend on the details of our secondary user records except through the definitions contained in `datatypes.h`. In particular, the letters "`sas_`", and the member names in su_info_t *must not* be found in `list.c`. It is also critical that the internal details of the data structures in the `list.c` file are kept private and are not accessed outside of the `list.c` file. The members of the structures `list_t` and `list_node_t` are private and their details *must not* be found in code outside of the `list.c` file. In particular, the letters "`->data_ptr`", "`->next`", "`->head`", "`->current_list_size`", etc. are considered private to the list ADT and *must not* be found in any `*.c` file except `list.c`.

Hint: Begin this assignment by completing all of Part 1 only. All work for Part 1 is contained in `list.c`. Test your `list.c` module using `driver.c`. Only when Part 1 is completed and tested should you move on to Part 2.

Part 2: The extended functions for CBRS SAS system

Our CBRS systems from MP1 is extended to now use two lists. Both lists must use the new two-way linked list ADT. The SAS accepts new secondary users and first assigns them to a waiting queue. A secondary user held in the waiting queue is not assigned to a frequency channel. The queue is maintained in first-in-first-out (FIFO) order and there is no limit on the size of the queue. For a FIFO queue, a new record is added at the tail and removed from the head. The SAS also maintains a second list that holds the secondary users that are assigned to a channel. This list has a maximum size and is maintained in sorted order based on the secondary user ID number (this is the same rules for ordering and limiting the list as for MP1). The following commands can be given to the SAS.

CREATE list-size
ADDSU
DELSU addr
ASSIGNSU channel-num
CLEARCH channel-num
CHANGECH old-channel-num new-channel-num
LISTCH channel-num
STATS
PRINTASSIGNED
PRINTWAITING
QUIT

CREATE: constructs the two list objects: the assigned sorted list and the waiting queue. As for MP1, the `list-size` is the maximum number of secondary user records that can be stored in the assigned sorted list. If the list objects already exist when this command is called, then the old objects must first be destructed and new empty objects created in their place.

ADDSU: a memory block to hold the secondary user information is created. If the secondary user ID is found in the assigned list, then two cases are possible. If the channel ID is the same, then simply update all the other information for the secondary user. If the channel ID is different, then remove the secondary user from the assigned list and place it at the tail of the waiting queue (and update all the secondary information). If the secondary user is not in the assigned list, then determine if the secondary user is in the waiting queue, and if so update the secondary user information. If the secondary user is not found in either the assigned list or waiting queue, then add the secondary user to the tail of the waiting queue.

DELSU: if a secondary user with ID `addr` is found in either the assigned list or waiting queue, remove the user and free the memory block.

ASSIGNSU: First check if the waiting queue is empty, and if so report that no users are waiting and otherwise take no action for this command. If there is at least one secondary user waiting, then check if the assigned list is full. If so, then report that the assignment could not be completed. Finally, if neither problem is discovered, then remove the secondary user at the head of the waiting queue and insert into the assigned list (in sorted order). Set the channel for the secondary user to `channel-num`.

CLEARCH: Remove all secondary users in the assigned list with a matching `channel-num` to the waiting queue. Do not change the `channel-num`. The search in the assigned list must be by increasing order of the secondary user ID. A secondary user removed from the assigned list is inserted into the tail of the waiting queue.

CHANGECH: Change the channel number of any secondary user in the assigned list that is assigned to the `old-channel-num` and updated to the `new-channel-num`. Do not consider any secondary users in the waiting queue.

LISTCH: print the record for all secondary users that are assigned to the frequency channel given by `channel-num` and that are found in the assigned list.

PRINTASSIGNED or **PRINTWAITING**: print each record in the respective list.

STATS: print the number of records in the assigned list and the waiting queue.

Implement the procedures to complete these commands in the `sas_support.c` file. The procedures must be implemented using the list ADT as the mechanism to store the secondary user records. You can modify the design of the `sas_support` code and header files to support your design. However, you cannot add any new `printf` commands. Only `printfs` provided in the `sas_support.c` file can be used. (Note: while you can change the `sas_support.h` header file, you CANNOT change the `list.h` or `datatypes.h` files).

Submit all the files listed on page 1 and a test script to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE223-1,#2. The 223-1 identifies you as a member of Section 1 (the only section this semester). The #2 identifies this as the second 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 listing all the attachments, your submission was not successful. You must include your files as attachments, and your email must be in text only format. 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.

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.

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student, and see the course syllabus for additional policies.