ECE 223: Computer Systems Engineering **Fall 2013**
Machine Problem 2 **Due: 11:59 pm, Thursday, September 12**

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 for saving and storing WiFi manager information.

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 WiFi records. One list is sorted and the other list is unsorted. 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.
`wifi_support.c`	– contains subroutines that support handling of WiFi records.
`list.c`	– The two-way linked list ADT. The interface functions must be exactly defined as described below. You can include additional functions but the interface cannot be changed.
`wifi_support.h`	– The data structures for the specific format of the WiFi records, and the prototype definitions.
`list.h`	– The data structures and prototype definitions for the list ADT.
`datatypes.h`	– Key definitions of the WiFi structure and a record comparison procedure needed by the list ADT
`makefile`	– Compiler commands for all code files

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`. We also define a function prototype for the specific function that compares two data structure elements. All of the procedures for the list ADT operate on `data_t`. In future programming assignments we can reuse the list ADT by simply modifying the `datatypes.h` file to change the definition of `data_t` and the comparison procedure prototype, and then recompile.

```
/*
/* 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
 *
 * comp_proc(A, B): function to compare two data_t records
 */

typedef struct wifi_info_tag {
       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
} wifi_info_t;

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

/* the comparison procedure is found in wifi_support.c */
#define comp_proc(x, y) (wifi_compare(x, y))
```

The list ADT must have the following interface, defined in the file `list.h`.

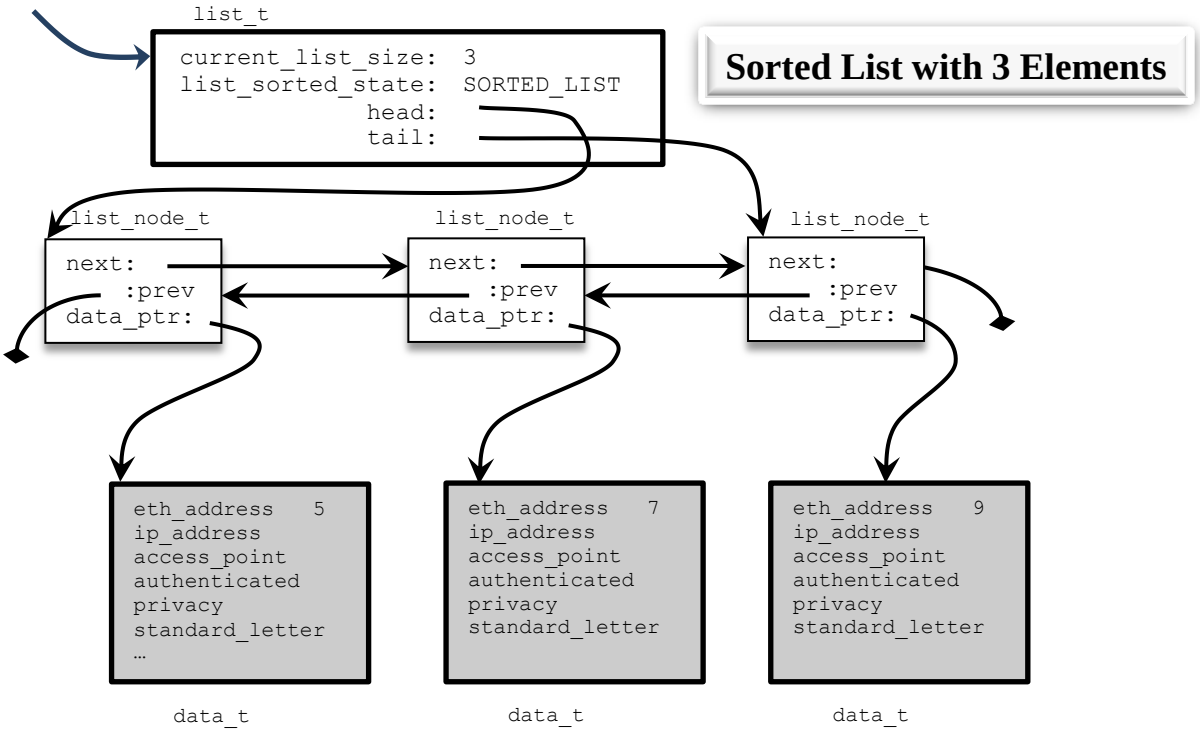
```
/* list.h
 *
 * You should not need to change any of the code this file.  If you do, you
 * must get permission from the instructor.
 */

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;
} list_t;

/* public definition of pointer into linked list */
typedef list_node_t * Iterator;
typedef list_t * List;

/* public prototype definitions for list.c */

/* build and cleanup lists */
```

```
List list_construct(void);
void list_destruct(List list_ptr);

/* iterators into positions in the list */
Iterator list_iter_front(List list_ptr);
Iterator list_iter_back(List list_ptr);
Iterator list_iter_next(Iterator idx_ptr);

data_t * list_access(List list_ptr, Iterator idx_ptr);
Iterator list_elem_find(List list_ptr, data_t *elem_ptr);

void list_insert(List list_ptr, data_t *elem_ptr, Iterator idx_ptr);
void list_insert_sorted(List list_ptr, data_t *elem_ptr);

data_t * list_remove(List list_ptr, Iterator idx_ptr);

int list_size(List list_ptr);
int comp_proc(data_t *, data_t *);
```

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 `wifi_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 WiFi records except through the definitions contained in `datatypes.h`. In particular, the letters "wifi_" , and the member names in wifi_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`.

The extended functions for WiFi records

Implement the following procedures in a `wifi_support.c` file. The procedures should be implemented using the list ADT as the mechanism to store the WiFi records. The following header file defines the example prototype definitions. You can modify the design.

```
/* wifi_support.h */

/* A template for wifi_support.h
 *
 * You may change this file if needed for your design */

#define MAXLINE 170

/* prototype function definitions */

/* function to compare WiFi records */
```

```
int wifi_compare(wifi_info_t *rec_a, wifi_info_t *rec_b);

/* functions to create and cleanup a WiFi list */
List wifi_create();
void wifi_cleanup(List);

/* Functions to get and print WiFi record information */
void wifi_record_fill(wifi_info_t *rec);   /* collect input from user */
void wifi_record_print(wifi_info_t *rec);  /* print one record */
void wifi_print(List list_ptr, char *);      /* print list of records */

/* functions for sorted list with a maximum size */
void wifi_add(List , int id, int size);
void wifi_lookup(List , int);
void wifi_remove(List , int);

/* functions for unsorted FIFO list that has no limit of the size of the list,
 * inserts at the tail, removes at the head,  */
void wifi_add_tail(List , int);
void wifi_remove_head(List );
void wifi_drop_max(List, int);
```

Extended user interface for managing packet records

Implement the five user functions from MP1 that control a sorted list that has a maximum size. These commands should produce identical results compared to MP1. As with MP1, the maximum size of the list is specified as a command line argument when the MP2 program is run. As in MP1, the list cannot contain any WiFi records with duplicate Ethernet addresses.

ADD eth_address
FIND eth_address
DEL eth_address
STATS
PRINT

In addition add the following four user functions for a *second* list that is *unsorted* and does *not* have a limit on the number of elements in the list. When adding a WiFi record, do not look for a duplicate entry, but simply add the new record to the end of the list. The DELMAX command finds the WiFi record with the largest Ethernet address and removes all WiFi records that match this Ethernet address.

ADDTAIL eth_address
DELHEAD
DELMAX
PRINTQ

QUIT ends the program and cleans up both lists.

Your program maintains two WiFi lists, one sorted and one unsorted. The first five commands operate on the sorted list that has a maximum size and does not allow WiFi records with duplicate Ethernet addresses. The second four commands operate on the unsorted list that does not have a maximum size and does not check for duplicates.

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*, queue size: *z*
PRINT	List empty
	List with *y* records 1: eth addr: *x* (other information in record printed on one line) 2: eth addr: *x* (other information in record printed on one line) (continue to print details of each record)
QUIT	Goodbye
ADDTAIL *x*	Added to tail: *x*
DELHEAD	Removed from head: *x* Did not remove: queue empty
DELMAX	No deletions, queue empty Removed *y* copies of *x*
PRINTQ	Queue empty Queue with *y* records (if the queue is not empty use identical format as for PRINT)

See the ECE 223 Programming Guide for additional requirements that apply to all programming assignments. 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,#2. Work must be completed by each individual student, and see the course syllabus for additional policies.