ECE 2230: Computer Systems Engineering **Fall 2015**
Machine Problem 2 **Due: 11:59 pm, Thursday, September 10**

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. You 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. You will then re-write the MP1 honeypot application to utilize the new list ADT.

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 packet information, one sorted and the other 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.
`hpot_support.c`	– contains subroutines that support handling of packet 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.
`hpot_support.h`	– The data structures for the specific format of the packet records, and the prototype definitions.
`list.h`	– The data structures and prototype definitions for the list ADT.
`datatypes.h`	– Key definitions of the honeypot packet structure and a packet 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`. 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
 */
typedef struct packet_tag {
    int dest_ip_addr;       /* IP address of destination */
    int src_ip_addr;        /* IP address of source */
    int dest_port_num;      /* port number at destination */
    int src_port_num;       /* port number at source host*/
    int hop_count;          /* number of routers in route */
    int protocol;           /* TCP=1, UDP=2, SSL=3, RTP=4 */
    float threat_score;     /* rating of source host */
    int time_received;     /* time in seconds last updated */
} packet_t;

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

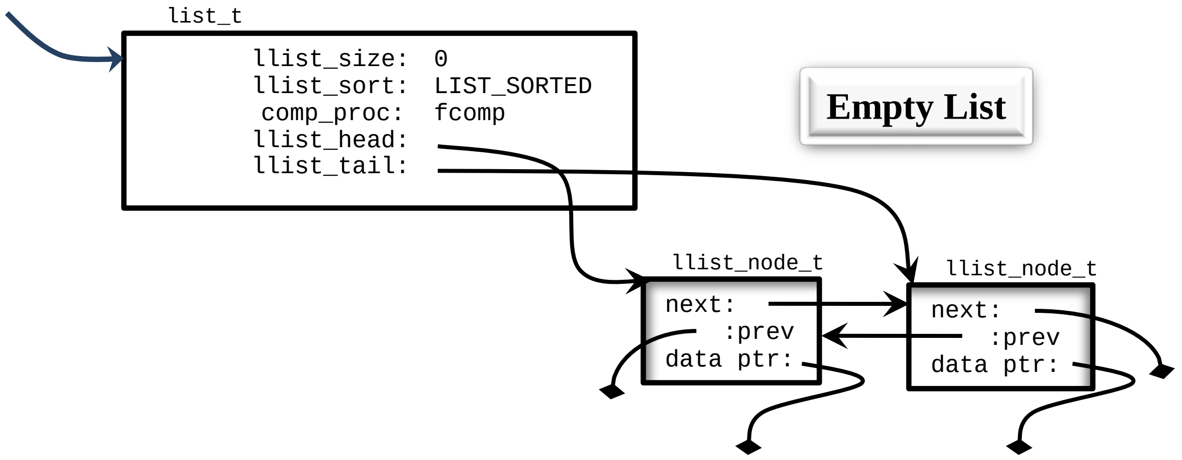
The list ADT must have the following interface, defined in the file `list.h`.

```
/* 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 llist_node_tag {
    /* private members for list.c only */
    data_t *data_ptr;
    struct llist_node_tag *prev;
    struct llist_node_tag *next;
} llist_node_t;

typedef struct list_tag {
    /* private members for list.c only */
    llist_node_t *llist_head;
    llist_node_t *llist_tail;
    int llist_size;
    int llist_sort;
    /* 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 llist_node_t * pIterator;
typedef list_t * pList;

/* public prototype definitions for list.c */

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

/* iterators into positions in the list */
pIterator list_iter_first(pList list_ptr);
pIterator list_iter_tail(pList list_ptr);
pIterator list_iter_next(pIterator idx_ptr);

data_t * list_access(pList list_ptr, pIterator idx_ptr);
pIterator list_elem_find(pList list_ptr, data_t *elem_ptr);
```

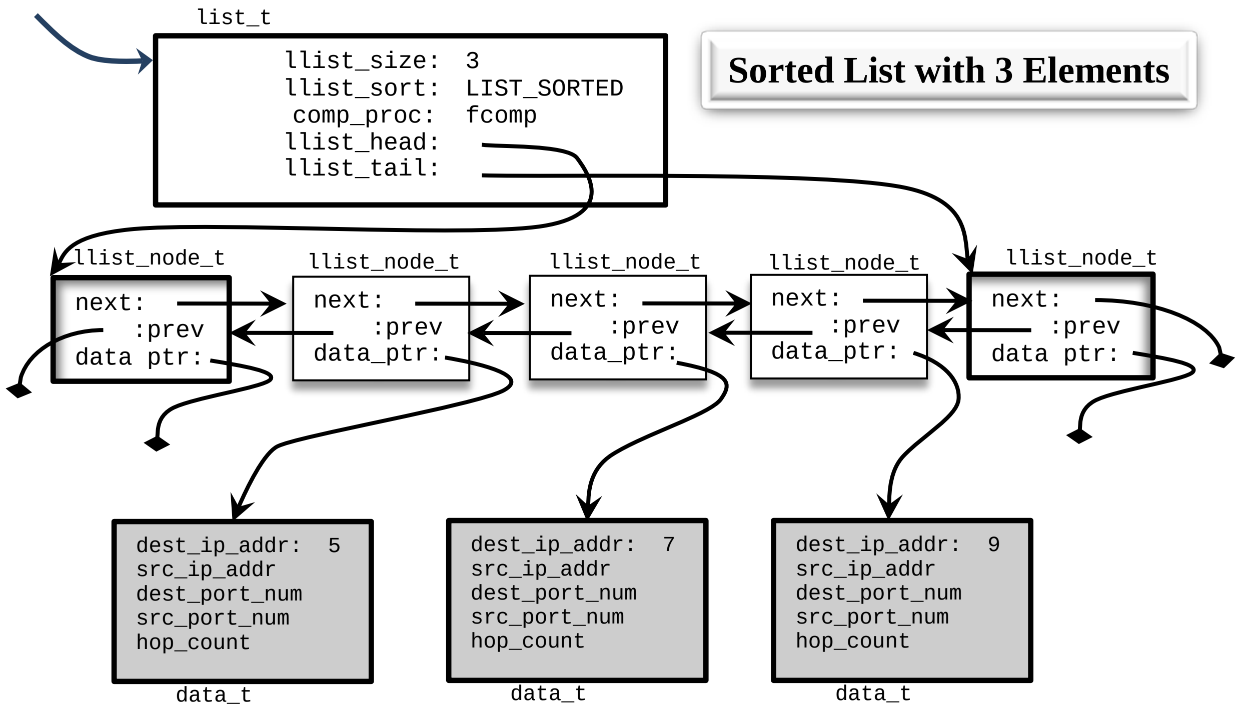
```
void list_insert(pList list_ptr, data_t *elem_ptr, pIterator idx_ptr);
void list_insert_sorted(pList list_ptr, data_t *elem_ptr);

data_t * list_remove(pList list_ptr, pIterator idx_ptr);

int list_size(pList 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.

The definition of the list ADT data structure includes two dummy memory blocks that serve as the head and tail of the list. When the list ADT is constructed, it must have the initial form as show in the figure below. Note that the `data_ptr`'s for the dummy head and dummy tail are always equal to null (these nodes never store a `data_t` memory block).



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 `packet_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 packet records except through the definitions contained in `datatypes.h`. In particular, the letters "hpot_" , and the member names in packet_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 `llist_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`", "`->llist_size`", or any variation on "`->llist_`" are considered private to the list ADT and *must not* be found in any `*.c` file except `list.c`.

The extended functions for packet records

Implement the following procedures in a `hpot_support.c` file. The procedures should be implemented using the list ADT as the mechanism to store the route records. The following header file defines the example prototype definitions. You ***can*** modify the design in `hpot_support.h` (but you cannot modify the design in `list.h`).

```
/* hpot_support.h */

#define MAXLINE 170

/* prototype function definitions */

/* function to compare packet records */
int hpot_compare(const packet_t *rec_a, const packet_t *rec_b);
```

```
/* functions to create and cleanup a Honeypot list */
pList hpot_create(char *);
void hpot_cleanup(pList);

/* Functions to get and print packet information */
void hpot_record_fill(packet_t *rec);   /* collect input from user */
void hpot_record_print(packet_t *rec);  /* print one record */
void hpot_print(pList list_ptr, char *);      /* print list of records */
void hpot_stats(pList sorted, pList unsorted);  // print size of each list

/* functions for sorted list */
void hpot_add(pList , int);
void hpot_list(pList , int);
void hpot_remove(pList , int);
void hpot_scan(pList , int);

/* functions for unsorted list with no duplicates
 * inserts at the tail, removes at the head,  */
void hpot_add_tail(pList , int);
packet_t *hpot_remove_head(pList );
```

Extended user interface for managing packet records

Implement the five user functions from MP1 that control a sorted list. These commands should produce identical results compared to MP1. Unlike with MP1, there is no need to specify the size of the initial list. So you ***must not*** read a command line argument when the MP2 program is run.

INSERT ip_address
LIST ip_address
REMOVE ip_address
SCAN threshold
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. However, there can never be more than one packet from a specific IP address in the list. So, the ADDTAIL command first checks if a packet with the same destination ip_address is already in the list. If so, the packet is removed and freed. In any case, the new packet is appended to the end of the list (the new packet is **not** inserted into the position of the old packet).

ADDTAIL ip_address
RMHEAD
PRINTQ

STATS prints the size of both the sorted and unsorted lists
QUIT ends the program and cleans up both lists.

Your program maintains two lists, one sorted and one unsorted. The first five commands operate on the sorted list. The second three commands operate on the unsorted list.

To facilitate grading, the output for each command must be formatted exactly as provided in the supplemental template programs. The `printf()` commands in `hpot_support.c` must not be changed.

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 to Blackboard.

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