ECE 2230: Computer Systems Engineering
Machine Problem 2

Spring 2020

Due: 11:59 pm, Wednesday, Februrary 17

1 Introduction

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. We will then re-write the MP1 atom application to utilizing our 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).

2 Problem Statement

You are to write a C program that will maintain the **two** lists of atom 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.

atom_support.c – contains subroutines that support handling of atom records. The interface functions must be exactly defined as described below. You can include additional functions, but the interface **cannot** be changed.

list.c – The two-way linked list ADT.

atom_support.h – The data structures for the specific format of the atom records, and the prototype definitions. list.h – The data structures and prototype definitions for the list ADT.

datatypes.h – Key definitions of the atom structure and a procedure needed by the list ADT

makefile – Compiler commands for all code files

2.1 The Two-way 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 called 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 and 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
*
*/

#define NDIM 2

typedef struct atom_info {
unsigned int atom_id;       // Unique atom id
unsigned int atomic_num;    // Atomic number for atom
float mass;                 // Atom's mass

float position[NDIM];       // Atom position
float momenta[NDIM];        // Atom momenta vector
float force[NDIM];          // Forces
float potential_energy;     // Potential energy per atom
} atom_t;

/* the list ADT works on atom data of this type */
typedef atom_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
*
* You should not need to change this file.  If you do please ask.  */

/* public constants used as parameters by most functions */

#define LISTPOS_HEAD -1010
#define LISTPOS_TAIL -1011

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 *);
void (*data_clean)(data_t *);
} list_t;

/* public prototype definitions for list.c */
data_t * list_access(list_t *list_ptr, int pos_index);
list_t * list_construct(int (*fcomp)(const data_t *, const data_t *), void (*dataclean)(data_t *));
data_t * list_elem_find(list_t *list_ptr, data_t *elem_ptr, int *pos_index);
void     list_destruct(list_t *list_ptr);
void     list_insert(list_t *list_ptr, data_t *elem_ptr, int pos_index);
void     list_insert_sorted(list_t *list_ptr, data_t *elem_ptr);
data_t * list_remove(list_t *list_ptr, int pos_index);
void     list_reverse(list_t *list_ptr);
int      list_size(list_t *list_ptr);
int      list_order(list_t *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. Look for "*TODO*" or "*fix*" in comments to identify where you should add code, modify return values, and variable assignments.

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.

It is critical that the `list.c` procedures be designed to not depend on the details of our atom records except through the definitions contained in `datatypes.h`. In particular, the letters "`atom_`" , and the member names in `atom_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`. You can check for some of these errors by using "`make design`"

2.2 The extended functions for atom records

Implement procedures to take atom record information and store the information in the our List ADT. The code to manage the details of the atom records should be placed in the `atom_support.c` file. The procedures should be implemented using the list ADT as the mechanism to store the atom records. The `atom_support.h` header file defines the prototype definitions.

The template files `lab2.c` and `atom_support.c` provide the framework for input and output and testing the list of atom 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 by looking for "*TODO*" or "*fix*" in comments to identify where you should add code, modify return values, and variable assignments. 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 and nothing else. The three example input and output files show the exact output format that is required. The format for MP2 is similar to MP1 but not exactly the same. Do not submit your code if your program cannot match to the given output files. If your code does not match the output, you must contact the instructor or TA and fix your code.

2.3 Extended user interface for managing atom records

Implement the user functions from MP1 that control a sorted list with exactly the same rules as for MP1. These commands should produce identical results compared to MP1. We also extend these command options to REVERSE the list of atoms.

INSERT
FIND potential_energy
REMOVE potential_energy
UPDATE
MIGRATE
REVERSE
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.

ADDTAIL
RMHEAD
PRINTHEAD
PRINTQ

STATS prints information about each list. QUIT ends the program and cleans up both lists.

Your program maintains two lists, one sorted and one unsorted. The MP1 commands operate on the sorted list that has a maximum size. The second four commands operate on the unsorted list that does not have a maximum size.

3 Submission

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

Compress all code, test cases, test case document defining what each test cases tests into a ZIP file, and submit the ZIP file to Canvas by the deadline. Your last submission is the one that will be graded.

See the ECE 2230 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.