CptS 122 – Data Structures

Programming Assignment 3: Digital Music Manager & Doubly Linked Lists – Part II

Assigned: Wednesday, September 12, 2018
Due: Friday, September 21, 2018 by midnight

I. Learner Objectives:

At the conclusion of this programming assignment, participants should be able to:
- Design and implement a dynamic doubly linked list
- Allocate and de-allocate memory at runtime
- Manipulate links in a dynamic linked list
- Insert items into a dynamic linked list
- Delete items from a dynamic linked list
- Edit items in a dynamic linked list
- Traverse a dynamic linked list
- Design and implement basic test cases

II. Prerequisites:

Before starting this programming assignment, participants should be able to:
- Analyze a basic set of requirements for a problem
- Compose C language programs
- Compile a program using Microsoft Visual Studio 2015
- Create basic test cases for a program
- Apply arrays, strings, and pointers
- Summarize differences between array notation and pointer notation
- Apply pointer arithmetic
- Apply basic string handling library functions
- Define and implement structures in C
- Summarize the operations of a linked list

III. Overview & Requirements:

In this assignment you will complete the Digital Music Manager that you started in PA 2. You must implement the following features:
- (4) insert
- (5) delete
- (7) sort
- (10) shuffle

You will also be required to write 3 test functions.

➢ *What must "insert" do?*
The "insert" command must prompt the user for the details of a new *record*. The prompt must request the artist name, album title, song title, genre, song length, number of times played, and rating. The new record must be *inserted* at the *front* of the list.

➢ *What must "delete" do?*
The "delete" command must prompt the user for a *song title*, and *remove* the matching record from the list. If the song title does *not* exist, then the list remains unchanged.

➢ *What must "sort" do?*
The "sort" command must prompt the user for 4 different methods to sort the *records* in the list. These include:
1. Sort based on artist (A-Z)
2. Sort based on album title (A-Z)
3. Sort based on rating (1-5)

4. Sort based on times played (largest-smallest)

Once a sort method is selected by the user, the sort must be performed on the records in the list. Consider using bubble sort, insertion sort, or selection sort.

➢ *What must "shuffle" do?*

The "shuffle" command must provide a random order in which the songs are played. This command must not modify the links in the list. It must just specify the order in which songs are played, based on the position of the song in the list. For example, let's say we have a list with 5 songs at positions 1 – 5 in the list, shuffle must generate an order 1 – 5 in which the songs are played. An order 2, 5, 3, 1, 4 would require that the second song in the list is played first, the fifth song in the list is played second, the third song in the list is played third, the first song in the list is played fourth, and the fourth song in the list is played fifth. The songs are accessed by traversing the list both forwards and backwards to satisfy the order. Hence, the need for a doubly linked list!

Once again you will find an example `musicPlayList.csv` (here).

➢ *What "test" functions are required?*

You must design and implement 3 test functions. These test functions must not accept any arguments or return any values. They should be self-sufficient. You should provide function declarations for them that are in a separate header file than your utility/application function declarations. Also, the corresponding implementations for them should be placed in a separate source file than your utility/applications function definitions and main (). You must implement one test function for *insert*, *delete*, and *shuffle* features for a total of 3 functions.

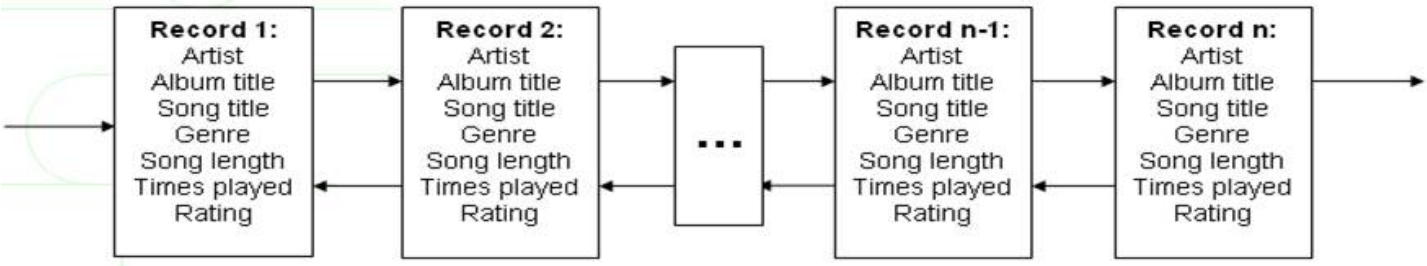
- For the *insert* test function you must provide a test *case* with the following test *point*: artist name = "Perry, Katy", album title = "Witness", song title = "Chained to the Rhythm", genre = "pop", song length = "4:36", times played = -1, rating = 6. List state = initially empty. You must think about what is your expected result? Should you able to insert a song with -1 times played? Should you able to add a song with rating 6? Is the head pointer of the list updated?

- For the *delete* test function you must provide a test *case* with the following test *point*: song title to delete = "Chained to the Rhythm". List state = artist name = "Perry, Katy", album title = "Witness", song title = "Chained to the Rhythm", genre = "pop", song length = "4:36", times played = 3, rating = 5 (the only song in the list). You must think about what is your expected result? Has the head pointer been updated? Is it NULL? Did the memory get released?

- For the *shuffle* test function you must provide a test *case* with the following test *point*: play order = 3, 1, 2. List state = you provide 3 songs. Does the shuffle play in the correct order?

IV. Logical Block Diagram

Once again, the logical block diagram for your doubly linked list should look like the following:



As you can see from the illustration a doubly linked list has a pointer to the next node and the previous node in the list. The first node's previous node pointer is always NULL and the last node's next pointer is always NULL. When you insert and delete nodes from a doubly linked list, you must always carefully link the previous and next pointers.

BONUS:

Modify your doubly linked list implementation(s) for your DMM so that last node in the list points to the first node, and the first node points to the last node. Hence, there is no longer a first or last node. This list is now called "circular". Overall, it is called a circular doubly linked list. Any one of the nodes may by the current node!

V. Submitting Assignments:

1. Using the OSBLE+ MS VS plugin, please submit your solution. Please visit https://github.com/WSU-HELPLAB/OSBLE/wiki/Submitting-an-Assignment for more information about submitting using OSBLE+.
2. Your project must contain at least two header files (a .h file), three C source files (which must be .c files), and a local copy of the .csv file. One of the header files is required to contain the declarations for your test functions and one of the .c files must contain the implementations for those test functions.
3. Your project must build properly. The most points an assignment can receive if it does not build properly is 65 out of 100.

VI. Grading Guidelines:

This assignment is worth 100 points. Your assignment will be evaluated based on a successful compilation and adherence to the program requirements. We will grade according to the following criteria:

- 5 pts – Appropriate top-down design, style, and commenting according to class standards
- 17 pts – Correct "insert" command implementation
  1. (7 pts - 1pt/attribute) For prompting and getting the details of a *new* record from the user
  2. (10 pts) For correctly inserting the record at the *front* of the list
- 24 pts – For correct "delete" command implementation
  1. (3 pts) For prompting and getting the *song title* from the user
  2. (5 pts) For *searching* for specific record *matching* the song title
  3. (16 pts) For *removing* the matching record from the list, and reconnecting the list correctly
- 29 pts – Correct "sort" command implementation
  1. (3 pts) For prompting and getting the *sort method* from the user
  2. (7 pts) For sorting based on artist (A-Z)
  3. (7 pts) For sorting based on album title (A-Z)
  4. (6 pts) For sorting based on rating (1-5)
  5. (6 pts) For sorting based on times played (largest-smallest)
- 15 pts – Correct "shuffle" command implementation
  1. (5 pts) For generating the random order based on the number of songs in the list
  2. (10 pts) For moving through the list (forwards and backwards) and playing the songs in the order generated
- 10 pts – Robust *test* functions – 3 required
  1. (4 pts) For a test function that challenges the bounds of your *insert* feature.
  2. (3 pts) For a test function that challenges the bounds of your *delete* feature.
  3. (3 pts) For a test function that challenges the bounds of your *shuffle* feature.

- BONUS: Up to 10 pts for correct circular implementation