Operating System - Assignment 1
Doubly Linked List in C

1 Objective

The objective of this assignment is to practice/review programming in C, including the use of C **structs** and **pointers**, by implementing a doubly linked list which will be used in later programming assignments.

2 Details

Write a C program that will allow the user to enter MP3 data of songs into a doubly linked list, remove MP3 entries by the artist's name, and display the contents of the list.

You should have a single `.h` header file that declares the MP3 struct. Your `.c` file(s) should `#include` the `.h` header file and contain all the actual C code. In this C struct you should have:

- The name of the artist stored in a char*
- The title of the song stored in a char*
- The run time in seconds stored as an int
- A "next" pointer to another one of these structs (so you can make a linked list)
- A "prev" pointer to another one of these structs (so you can make it a doubly linked list)

Once you have your struct, write a program to use it that will prompt the user to (1) add an MP3 to the list, (2) delete MP3(s) from the list, (3) print the list from the beginning to the end, (4) print the list from the end to the beginning, and (5) exit the program. When adding an MP3, your program prompts for each piece of information, and adds the MP3 to the end/tail of the list. When deleting, your program prompts for the name of the artist and deletes all the entries with that artist. When printing, start at one end of the list, traverse and print each MP3 until reaching the other end. When exiting the program, free all dynamically allocated memory (i.e., the exsiting doubly linked list) to leave no memory leak. You may use fgets() to input character strings (for artist name and song title) as shown in this example. Read the man page of fgets() for details.

Your input and output should make sense to the user.

Remember to always allocate memory for all your pointers to point at valid memory with malloc() or calloc(). Avoid dereferencing NULL pointers which will result in segmentation faults unless you happen to get lucky! Also, check out C tutorial on call-by-value and call-by-reference: `https://www.codingunit.com/c-tutorial-call-by-value-or-call-by-reference`.

3 Makefile and submission instructions

You should create a Makefile so that when TA types "make mp3", an executable named mp3 is created.

You should tar up your source code and the Makefile, and submit the resulting tar file so that your program can be tested by TA. To do this, do the following.

- In the current working directory, create a (sub-)directory named proj_1 to store all your code and the Makefile.
- tar cvf YourLastName(s)_proj_1.tar proj_1

To verify that your files are in the tar file, take a look at the table of contents of the tar file like:

- tar tvf YourLastName(s)_proj_1.tar

Submit (upload) your tar file into Canvas. TAs will run your code under Valgrind to check for any memory leak.

4 Sample code

- Sample code: PA_1_sample_code.tar
- Another simple linked list code: `https://www.learn-c.org/en/Linked_lists`
- Youtube: `https://www.youtube.com/watch?v=JdQeNxWCguQ`

5 Notes

Your MP3 C struct in Project 1 should look something like this.

```
typedef struct mp3 {
  char       *name;
  struct mp3 *next;
} mp3_t;
```

And, it should NEVER be something like the following.

```c
typedef struct mp3 {
  char name[128];
  struct mp3 *next;
} mp3_t;
```

Can you tell the difference?

Try the following code on cisc361.acad to learn how fgets() works, how to dynamically allocate memory for each MP3 record, and how to dynamically allocate memory for the "name" field within each MP3 record.

Run the following C program on cisc361.cis.udel.edu under valgrind to see if there is any memory leak. (There shouldn't be any.)

```c
    // struct.c (in ~/361/PA_1)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFERSIZE 128

typedef struct mp3 {
  char       *name;
  struct mp3 *next;
} mp3_t;

int main (int argc, char *argv[])
{
  char  buffer[BUFFERSIZE];
  mp3_t *mp3;
  int   len;

  printf("Enter a name: ");
  if (fgets(buffer, BUFFERSIZE , stdin) != NULL)
  {
    len = (int) strlen(buffer);
    printf("length [%d] of string %s", len, buffer); // notice the
       ↪ length!
    buffer[len - 1] = '\0';                  // why minus 1 ???
    mp3 = (mp3_t *) malloc(sizeof(mp3_t));
    mp3->name = (char *) malloc(len);
    strcpy(mp3->name, buffer);
    printf("Name is [%s]...\n", mp3->name);
  }
  free(mp3->name);   // line free 1
  free(mp3);         // line free 2
  return 0;
}
-------------
```

Now, try to comment out line free 1 and/or line free 2 to see if there are memory leaks under valgrind. (There should be now.)

Also, can the order of line free 1 and line free 2 be switched???

Here is an excerpt from the man page of fgets().

"fgets(char *s, int size, FILE *stream) reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer."

Try to comprehend it to understand why we need the "minus 1" in buffer[len - 1] = '\0';