CSCI-1200 Data Structures — Spring 2017
Homework 5 — RadioDS Song Grouping

RadioDS is a new radio station that is trying to get their library organized. The station has made a deal with a music company, and is able to use a large collection of songs, called a `Library`. However, in order to keep the licenses, **RadioDS** has been told that they cannot play any one song too frequently. Instead of keeping an exact count, the station has decided it's good enough if they break up the library into different `Song Groups`. A given song can be in 0 or 1 `Song Groups` at at any time. If a staff member decides they want a song that's currently in Song Group 1 to be used in Song Group 2 instead, they must first remove the song from Song Group 1.

To stand out from the many competing stations, *RadioDS* has two speciality radio programs. One is called an "Artist Marathon", where all the songs played will have the same artist. The other program is called a "Rewind List", where every song played is from an earlier year than the song before it. Examples of both programs are placed later in this handout.

Before beginning this assignment, make to **read the entire handout**. There are several classes involved in the operation of *RadioDS*, and it is important to understand how they interact in order to succeed on this assignment. In addition, many of the things described in this handout are already implemented and provided, so if you are not careful you may be doing more work than you should!

Classes and Data Organization

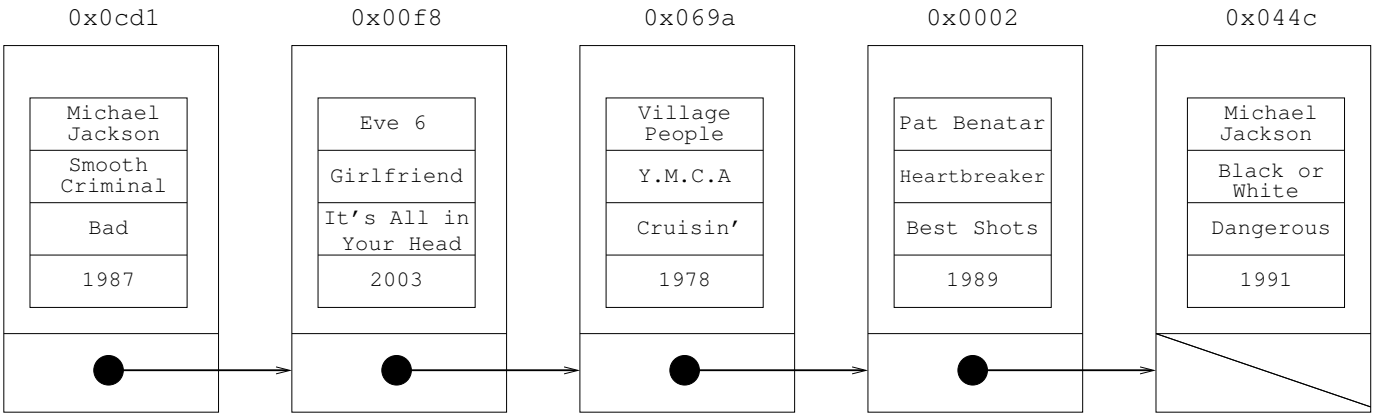
There are three classes used in this assignment.

`Song` is a simple class that contains the metadata (information) about a song. There are four distinct attributes used to identify songs: the *title* of the song, the *artist*, the *year* the song came out, and the *album* the song was on. The artist is the band or person(s) who made the song. An album is typically a CD or vinyl record, but it may represent a different collection. We will not focus on more advanced metadata, but if you are curious about how complicated describing a song can get in the real world, you may find this link to be an interesting read: https://musicbrainz.org/doc/Terminology

Each `Song` also has a boolean flag to indicate whether it has been used in a Song Group, and a pointer to a `SongGroupNode`, described later in this section. If there is no `SongGroupNode` associated with this `Song` then the pointer should be set to `NULL`. It will be important to keep these variables updated throughout your code.

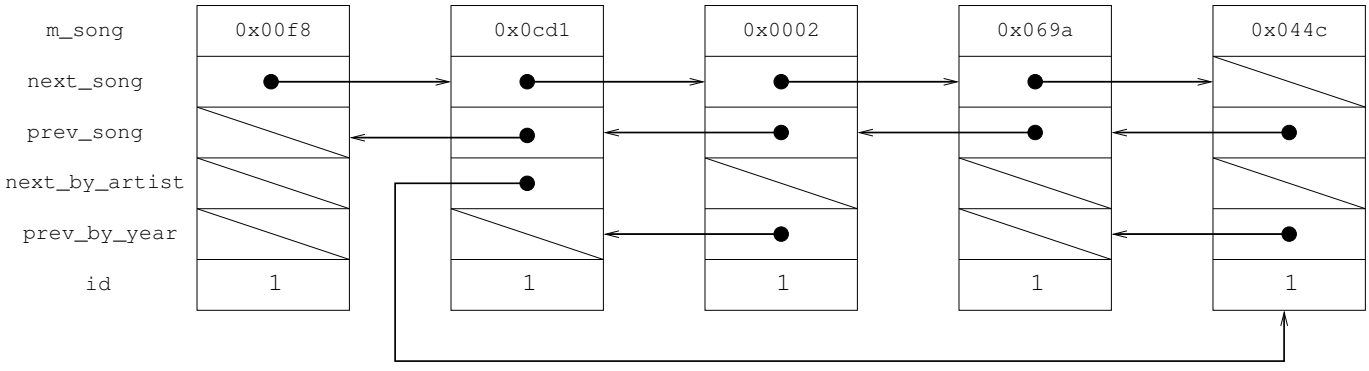
To provide a singly-linked list, we have included the templated `Node<T>`, similar to what has been seen in class. Each `Node` holds a `value` and has a pointer `ptr` that points to the next node. There is no special "head" node; the list is either empty or it contains one or more `Node` objects linked together through forward pointers.

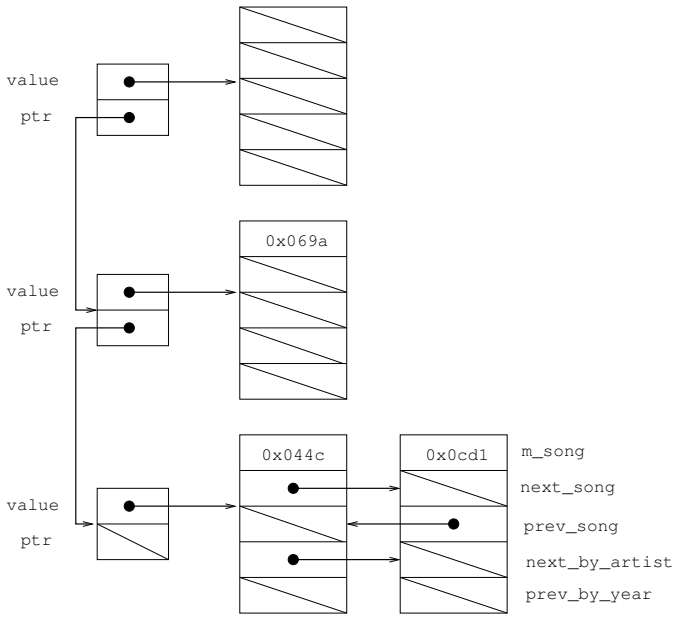
The `Library` is represented by a singly-linked list of type `Node<Song>*`. Your implementation should never copy `Song` objects out of the `Library`. Instead we will use pointers in other classes to decrease how often we have to search through the `Library`. Below is an example of a `Library` with simplified `Song` objects (we do not include the `bool` or `SongGroupNode*`. We include memory addresses of the `Node<Song>*` to help illustrate the other structures.



The other concept we have to represent is `Song Groups`. Before we can do so, we need to introduce a complicated linked list node type called `SongGroupNode`. Two of the pointers in each node should look familiar - namely *prev_song_ptr* and *next_song_ptr*. These two pointers allow us to traverse a `Song Group` like any other doubly-linked list, so maintaining them should be familiar if you have been following along with lecture and lab. Neither pointer has anything to do with the order in the `Library`, they instead reflect the order that `Song` objects were added to the `Song Group`. The *next_by_artist_ptr* should point to the next `SongGroupNode` in the `Song Group` with the same artist. Similarly, the *prev_by_year_ptr* should point to a previous `SongGroupNode` in the `Song Group` which has an earlier year than the current `SongGroupNode`.

Before illustrating what a `Song Group` looks like, there are two more members in the `SongGroupNode` that need to be described. To tell `Song Groups` apart, each `Song Group` is assigned a unique identification number (ID), which is stored in the *id* member of every `SongGroupNode`. The final member is *m_song*, which points to a `Node<Song>*` - in other words it points to the `Library Node` that contains the `Song` represented in the corresponding `SongGroupNode`. Below is a picture of a `SongGroup`, with addresses from the `Library` illustrated above instead of replicating all the `Song` members:





There is one final part to the organization of data, and that is how to keep track of `Song Groups`. Since we need to know the beginning of each `Song Group`, we maintain a list of "group heads" in a singly-linked list of `Node<SongGroupNode*>*` objects. Each `Node` points to the first `SongGroupNode` in a `Song Group`. This introduces an interesting quirk to how we represent `Song Groups` though - there will always be at least one `SongGroupNode` for a given group ID. When the group is "empty", the first `SongGroupNode` will simply set m_song to `NULL`. An example of the group head list, along with an illustration of an empty, one song, and two song `Song Group` is shown to the left.

Execution and Input Commands

The program will read in a library file, and then read in a series of commands. Since the syntax of input commands is less important (the code to read in commands is already written for you), we will only describe them briefly.

Execution looks something like:

```
./radioDS.out TinyLibrary.txt input.txt out_library.txt out_songgroup.txt out_print.txt
```

The first argument is a file that contains the `Library` data. The second argument is a file contains a list of commands, which you can assume are always formatted correctly. The third file is where output for library-related commands will be saved. The fourth file is similarly a file for output from `Song Group` commands, and finally the fifth file holds output from "print" commands. The commands are described below, and grouped by which file their output goes to.

Library Commands

add_to_library adds a new `Song` to the `Library`

next_in_library attempts to find a `Song` and print the `Song` that was added to the `Library` immediately after it

song_exists prints out whether or not a `Song` is in the `Library`.

Song Group Commands

make_group creates a `Song Group` with a particular ID if that group doesn't already exist.

group_exists prints out whether or not a `Song Group` exists.

song_in_group prints out whether or not a particular `Song` is in a specific `Song Group`.

add_to_group attempts to add a `Song` to a specific `Song Group`.

remove_from_group attempts to remove a `Song` from the `Song Group` it's in.

remove_group removes a `Song Group` from the list of groups, and marks any `Song` that was in it as unused.

Print Commands

print_library prints every `Song` in the `Library` in the order it was added.

print_group_sequential prints every `Song` in a `Song Group` in the order that the songs were added.

print_group_rewind prints a `Rewind List` starting from (and including) a particular `Song`.

print_group_marathon prints an `Artist Marathon` starting from (and including) a particular `Song`.

print_longest_group_rewind finds the longest `Rewind List` in a given `Song Group`, and prints out the `Rewind List` along with its length.

Your Task

Unlike previous homework assignments, the parsing is already complete, and so is the logic to handle the commands. The parsing and input handling logic is in *main.cpp*, and this file should not be changed. Similarly, none of the typedefs, class definitions, or prototypes in *SongLibrary.h* should be changed. We still recommend you read through both files to understand what's happening. *main.cpp* also handles all the `Library` functions, and you may find the code that is already written helpful in figuring out how to manage singly-linked lists.

You will need to complete the implementation of *student_functions.cpp* file. A few functions at the top of the file should not be changed. You should only change the implementation of functions that are empty. Each of the functions you should implement has a comment block above it which has some information about the arguments being passed in, the expected behavior of the function, and the return values. Between the description in the write-up, the code in *main.cpp*, and the comments in *student_functions.cpp*, you should be able to figure out what each function should do.

You are not allowed to use STL `vector` or STL `list` objects. You may not write any additional classes or structs. You should make sure the program does not run with any memory leaks or memory errors when compiled with your implementation.

While you are encouraged to write your own test input, we will not be looking at your inputs. You should submit only a completed *README.txt* and your completed *student_functions.cpp*.

Helpful Hints

Linked lists can get messy quickly, so we advise looking at small examples and drawing pictures when your code is behaving strangely. Update your picture as you follow your code line-by-line and see if something in the drawing doesn't end up the way you expected it to. Drawing pictures will be helpful in understanding the various data structures and may help you as you plan out your implementation.

A technique that will be of particular use is getting the address of a variable on the stack using &. For example, consider the following code:

```
int x = 5;
int* y = &x;
Node<int*> n;
n.value = y; //Now n.value is 5
*(n.value) = 10; //Now x is 10
```

Some sample files have been provided to give you an idea of how the output looks and start you off with some simple test inputs. These are not thorough, but they do test a few things that you may not have thought about. The library *library_small.txt* is used for *input_basic.txt*, *input_basicsong.txt*, and *input_moregroups.txt* while *input_radio.txt* is designed to work with *library_medium.txt*