Programming Assignment 2

Overloads and Dynamic Memory

Due 4/7/2024 @ 11:59 PM

Redo deadline 4/28/2024 @ 11:59 PM

Assignment Objectives

- Provide practice with function overloading.
- Provide practice with operator overloading.
- Provide practice with friend functions.
- Provide practice with dynamic memory allocation.
- Provide practice with reading technical documentation.

Important Notes

- The assignment has a unit test file named PA2_endToEnd.py. You can run the unit test executable by doing **python3 PA2_endToEnd.py**t, in the terminal.The test file must be in the same directory as the files you want to test. Each test will give you a test name, which you can use to go back and fix your code.
    - If you get a permission denied error, run **chmod u+x <filename>** where <filename> is PA2_tst. (<> angled brackets indicate to r**eplace the entire entry with the requested information, including the brackets**)
    - Test your code on Ubuntu for the graders
    - If you have not coded in your exits at each menu, the unit tests will become stuck and they will either fail or never finish. Make sure you have them done before attempting to run the unit tests.
- 50% of your grade will be for passing the end to end test. The other half will be for program structure, which we will grade using the rubric posted on WebCampus with the assignment. Make sure you are following the styling conventions laid out on the c++ styling conventions page on WebCampus.
- I have provided a database file dbLarge.txt. This is the file that is being used in the test as well.
- Song objects should be encapsulated using a Playlist object, which dynamically allocates space for each new Song. **Do not use arrays to work with Songs in the Playlist.**
- Code that does not compile and run until the main menu, uses libraries not explicitly allowed, or uses an array in the playlist class as opposed to dynamically allocated pointers will receive a 0.

- If you find that you're "losing" information from a pointer to an object, you likely left scope. Make sure that the object has been passed by reference or pointer in every function that object visits, even if you're not making changes to it in that function.
- Please name your executable **jukebox.**

Program

Your program should contain the following files:

- main.cpp
- song.h
- song.cpp
- playlist.h
- playlist.cpp
- helpers.h
- helpers.cpp
- makefile

Please **do not** include anything other than your .h, .cpp, and makefile in the final submission. You are welcome to add additional .h and .cpp files where it makes sense to do so, but the files listed are the bare minimum expectation.

Programming Problem

Write a program that allows users to create a playlist using a song database. Here's example main menu output, though you are welcome to format it in whatever way you see fit:

Welcome to SongStoreSupreme!
1. Load an exis8ng playlist.
2. Create a new playlist.
3. Exit SongStoreSupreme.
Make your selection:

The program should then allow the user to enter a value, 1-3, to view **on the same line as the prompt to view at the end of the display**. If a user makes an invalid entry, the program should repeatedly display the menu until a valid option is chosen. If the user chooses to exit, the program should gracefully end- do not use the exit function. If the user chooses to create a playlist, the user should be prompted for the playlist name on the same line as where the entry occurs, like so (again, formatting is up to you):

Playlist name:

The playlist name should be able to take all character entries until the user hits enter, including spaces. If a playlist with that name already exists, the user should be told and then prompted to enter a new name. Once the user hits enter after a non-exist, all songs in the database should be displayed, like so:

Displaying Available Songs:
1. **'Hey Ya!**
2. **Dreams**
3. **Get Your Freak On**
4. **Strawberry Fields Forever**
5. **What's Goin' On?**
6. **Smells Like Teen Spirit**
7. **A Change is Gonna Come**
8. **Fight the Power**
9. **Respect**
10. **I Want to Hold Your Hand**
11. **Finalize List.**
12. **Exit Program.**

Make a selection:

When you load the songs is up to you, as is format, but please make sure your options are in the same order as the provided file. I have provided a dbSmall.txt containing 10 entries for testing and debugging, which can be hardcoded in as you work on your solution. Your final program should use dbLarge.txt.

Next, the program should allow the user to enter a value 1 - (number of songs +2), where (number of songs +1) is the option to finalize and save the list, and (number of songs +2) is the option to fully exit the program. Users should be allowed to add as many songs from the database as they want, repeatedly prompting them to make a selection until the option to finalize or exit is selected. Songs in the list should appear no more than once. If the user chooses to exit the program after creating the list, they should be prompted to save the list, like so:

Would you like to save (y/n):

If the user selects yes, the playlist should be saved as <playlist>.txt- for example, **Gaston's Jams.txt -** Saved playlists **should not** have angled brackets around them, and the file saved to should match the name of the playlist given. The playlist output should be the same as the databases. That is, it should be:

songTitle-songAr8st

If the user chooses not to save, then no file should be opened or written to, and the program should terminate gracefully.

If the user chooses to finalize the list instead of exit the program while creating a list, then the playlist should be saved to a txt file with the same name as the playlist, using the same format above. If the user chooses to exit or finalize before selecting any songs to add to the playlist, the program should not create any saved file for the playlist.

Back at the main menu, if the user chooses to load an exiting playlist, then the program should find all of the files in the current directory ending in .txt that are not dbSmall or dbLarge. Code to gather the names of all files in a directory is below (you will need to modify it to work as intended):

```
DIR *directoryPtr = opendir(".");
struct dirent *directoryEntry;
int numPlaylists = 0;
if (directoryPtr) {
   while ((directoryEntry = readdir(directoryPtr)) != NULL) {
      string filename = directoryEntry->d_name;
   }

closedir(directoryPtr); //close all directory
}
```

You will also need to **include dirent.h** in the helpers file to use the code above. After the program gathers all txt file names excluding the databases, the playlist name (not including .txt) should be displayed as available to the user, like so:

The following playlists are currently available:
1. Gaston's Jams
2. Ge
3. Temp
4. Return to Main Menu.
5. Exit Program.

Which playlist would you like to load?

Again, the order matters but format does not. If the user chooses to return to the main menu, then the previous menu should be displayed. If the user chooses to exit, the program should be exited using a return from main- you will likely need to use return values from functions to indicate if the user has chosen to exit. If the user selects a playlist, the playlist title and all songs in it should be displayed, immediately followed by the playlist menu option, shown below:

1. Modify Playlist.
2. Delete Playlist.
3. Return to Main Menu.
4. Exit Program.
What would you like to do with Gaston's Jams?

Then, the program should construct a Playlist object containing all songs in the playlist. Just like with previous menus, if the user chooses to exit, then find a way to indicate that to main so that the program uses a return value of 0 to terminate. If the user chooses to return to the main menu, then the program should return to the original menu shown at the beginning of the program. If the user chooses to delete the playlist, the remove function should be used to permanently delete it from the current directory, and the playlist object created to hold the playlist should be cleaned up. Then the program should prompt the user to either exit the program or return to the main
menu, like so:

Gaston's Jams deleted!
1. Return to Main Menu.
2. Exit Program.
What would you like to do?

If the user chooses to modify the selected playlist, the user should be prompted to remove songs, add songs, return to the previous playlist options menu, or exit, like so:

1. Remove Song(s).
2. Add Song(s).
3. Return to Playlist Op8ons Menu.
4. Exit.
What would you like to do?

If the user chooses to remove songs, then the songs in the playlist should be displayed with the option to return to the playlist options menu and to exit the program, like so:

========Ge========
1. **'Hey Ya!**
2. **Dreams**
3. **Return to Playlist Options Menu.**
4. **Exit Program.**

Select a song:

The user should be repeatedly prompted to select until they choose to either return or exit. If one of the songs is chosen, that song should be removed from the playlist. A user should not be able

to remove a song more than once. If the user selects all songs to be removed, the program should delete the file using the same method used to delete a playlist from earlier, and then it should ask if the user would like to exit the program y/n, like so

Exit program (y/n)?

If they choose not to exit, the program should return to the main menu.

If the user chooses to add songs to the playlist, then all songs in the database should be displayed, as well as an option to return to the previous menu or exit the program, like so:

1. 'Hey Ya!
2. Dreams
3. Get Your Freak On
4. Strawberry Fields Forever
5. What's Goin' On?
6. Smells Like Teen Spirit
7. A Change is Gonna Come
8. Fight the Power
9. Respect
10. I Want to Hold Your Hand
11. Return to Playlist Options Menu.
12. Exit Program.
Select a song to add:

The user should be repeatedly prompted to select a song until they choose to return or exit. Songs should not be added to the playlist more than once, but the user should not be told that it isn't being added. Once a user chooses to add a song, they should be prompted as to whether they should add the song at the end, like so:

Add song to end of playlist(y/n)?

If they choose to add it to the end, the song should be added after the last one in the playlist. Otherwise, the program should display the songs in the playlist and ask which song to insert before, like so:

========Ge========
1. Dreams
Which song would you like to insert before?

The program should repeatedly prompt the user to add a song until they choose to return or exit.

Note that all user entries should be written so that boundaries are checked. For example, if in the previous display the user entered 2, **which song would you like to insert before? w**ould be displayed again until the user chose a song to insert before that existed. That means that every

user entry should be surrounded by a loop that checks for user input and confirms that it's within the bounds of the menu or prompt.

==========

Program Requirements

- All menus should use the same numbering for options.
- Unless otherwise indicated in the function description, listed methods should be public.
- In addition to the function overloads in the playlist class listed below, the helpers should have at least one other function overload.
- All pointers should be dynamically allocated using the **new** keyword.
- Classes using pointers should follow the Rule of Three to ensure pointers do not lose scope.
- If you don't understand how the pointers work as opposed to an array, please see Figure I below. If you need a visual representation of the addition process, please see Figure II below. If you need a visual representation of the removal process, please see Figure III below.
- Your program should have two classes: Song and Playlist; Additional description is below.
    - *Song* consists of a **title** and **artist,** and a **pointer to the next song.**
        - Songs should not be stored in arrays. They should only be stored in the playlist object. Failure to follow this rule will result in a 0 on the assignment, since we're practicing dynamic memory allocation.
        - The pointer should be dynamically allocated and point to the next entry. If there isn't a song after it in the playlist, then the next song should be **nullptr**
        - The default constructor should set the next song pointer to be a nullptr, since no other songs will have been linked to it at creation.
    - *Playlist* consists of the **number of songs in the playlist, playlist name,** a **pointer to the first song in the playlist**, and **a pointer to the last song in the playlist.** See the diagrams at the end for a visual representation.
        - The playlist should not use any arrays to store the songs. Arrays **are** allowed for strings or integer values if you would like in helpers.cpp
        - Every song has a pointer to the next song in the playlist. You can use the firstSong pointer to iterate through all of the songs in the playlist to do the work that you need to do. To do so, copy the first song pointer to a temporary pointer that you can update over time, and create a previous pointer that is null at the start. Then, construct a while loop that stops either at the end of the list (nullptr), or once an item is found (usually an index). Within the while loop, if the nullptr hasn't been found or the index

hasn't been reached, set the previous pointer to be equal to the current pointer, and set current to be current's next song.

- The last song in the playlist should always have the nextSong be **nullptr**. This allows you to check if you've iterated through the entirety of the playlist.
- The default constructor should set the first and last song properties to be nullptr, since there aren't any items in the playlist.

==========

In addition to basic constructors, getters, and setters, you must include the following functions in the *Song* class, they must match the specifications given, and your parameters should be provided in the order listed:

Method: insertion operator overload
Return Type: ostream&
Parameters: outputStream(ostream&), song(const Song&)
Pre-Condition: Songs have been loaded from file and/or from user input
Post-Condition: output stream is overloaded so that the insertion operator (<<) can be used directly on a song, like so:

```
cout << someSong;
```

Description: This should be a friend function, so place the definition in helpers.cpp and don't forget the friend keyword when declaring it in Song. This method should use the passed output stream to output the title and an endline, then return the output stream. **Anytime you want to output the song title, you must directly cout the song, as above.**
Methods Called: None

==========

In addition to basic constructors, getters, and setters, you must include the following functions in the *Playlist* class, they must match the specifications given, and your parameters should be provided in the order listed:

Method: addNewSong
Return Type: void
Parameters: song (const Song)
Pre-Condition: User has selected a song to add from the playlist containing database songs, and the song has passed to this function.
Post-Condition: The song is added to the **end** of the playlist.
Description: This function should be **private.** It should first construct a temporary pointer using the new keyword. Then, it should check if the song has already been added to the playlist (use another function to do this!). If it hasn't, then the newly created temporary pointer should use the passed song to set the title and artist, and it should set the next song to be a nullptr (since it will be at the end). If the first song pointer is null, then this is the first song added to the playlist, so you will set the first song equal to the pointer for the last song, which should then be set equal to the new temporary pointer. Otherwise, modify the lastSong pointer so that the next song is the new temporary pointer, and then update lastSong so that it is now the temporary pointer. Don't forget to increment the number of songs loaded to the playlist!
Methods Called: Your choice, but there should be a call to a method that checks the playlist for the song that you're trying to add.

Method: addNewSong
Return Type: void
Parameters: song (const Song), index (int)
Pre-Condition: User has selected a song to add from the playlist containing database songs, as well as a song to insert before. The song and index are then passed to the function.
Post-Condition: The song is added **before** the song at the selected index.
Description: This function should be **public.** The description given below is *one* way of inserting before the selected song- if you have another way that makes more sense to you, use it. First, create a temporary pointer to the first song in the playlist (currentSong), and a temporary pointer to the previous song (which should be nullptr since we're at the beginning). Construct a third pointer that is a dynamically allocated song (temp). Check if the song is already in the playlist. If it isn't, set the title and artist of the temp song. Then, use a while loop that iterates through the songs in the playlist while the number of songs iterated through is less than the user entered index. Within the loop, if the number of songs iterated through is not equal to the index -1, set the previous node to be the current node, and set the current node to be current's next song- this process will allow you to iterate through the playlist one song at a time. Otherwise, if

the number of songs iterated through is equal to the index - 1, it's time to add the new song. There are 4 different cases for adding at that point. If the index is equal to the number of songs in the playlist, then you can just add it to the end with a call to the other private addNewSong method. If their are no entries in the playlist, then set the first song to be equal to the last song, and set them both equal to the temporary pointer that you created with the new song info in it. If the index entered by the user is 1, then you are inserting at the front of the playlist, so set the temporary pointer's next song to be the current song, update the first song to be the temporary pointer, and increment the number of songs loaded. If the song is being inserted somewhere other than the front, back, or within an empty list, set the next song of the temporary pointer to be the current song, set the previous song's next song to be temp, and increment the number of loaded songs. You will need a counter initialized to 0 to track how many songs have been visited in the playlist while iterating with the while loop.

Methods Called: addSong(Song), a method to check if the song is already in the playlist.

Method:addition operator overload
Return Type: None
Parameters: newSong(Song)
Pre-Condition: User has selected song to add, and has chosen to add it at the end.
Post-Condition: Song is added to the end of the playlist.
Description: You were probably thinking "Sara, why on earth would I want to make the addNewSong function private". So that we can use the addition operator overload in a way that makes sense! Since addNewSong(song) is private, the only job this function has is to call addNewSong and pass it the song parameter. The invocation of this method **must** be done any time a song is added to the end of a play list. It should look like this:

```
somePlaylist + someSong;
```

Methods Called: addNewSong(song)

Method:deleteSong
Return Type: void
Parameters: None
Pre-Condition: User has selected a song to delete from the playlist.
Post-Condition: Song is removed from Playlist.
Description: This function should be **private. and no loop is needed.** If the first song is also the last song, then removing the song at the front will empty the list, so set first and last song to be nullptrs. Otherwise, set the firstSong to be the next song after first song. Decrement the number of songs in the playlist.

Methods Called: Your choice.

Method: deleteSong
Return Type: void
Parameters: index (int)
Pre-Condition: User has selected a song to delete from the playlist.
Post-Condition: The song is added to the **end** of the playlist.
Description: This function should be **private.** The description given below is *one* way of inserting before the selected song- if you have another way that makes more sense to you, use it. If there is only one song in the playlist, then you are technically deleting from both the front and the end of the playlist, so you can call deleteSong() directly. Otherwise, create a pointer to the first song in the playlist (current) and a pointer to the previous song (which is null at the start). Just as you did with the addSong method, construct a while loop that iterated until the number of songs iterated through is equal to the song selected for deletion. Within that loop, as long as the number of songs iterated through is not equal to the index of the song selected for deletion minus 1, set previous to be the current song, and set current to be current's next song (just like you did in add). This will iterate through every single song in the playlist without the use of an array. Otherwise, if the number of songs iterated through is equal to the index of the song minus 1, there are two possible cases. If the index of the song selected is one, then call deleteSong() directly. Otherwise, set the previous song pointer's next song to be current's next song, delete the current song, and set the current pointer to the previous' next song. Then decrement the number of songs in the playlist.
Methods Called: Your choice, but you should use a call to the other version of deleteSong where appropriate.

Method:subtraction operator overload
Return Type: None
Parameters: index (int)
Pre-Condition: User has selected song to remove
Post-Condition: Song at the supplied index is removed.
Description: You were probably thinking "Sara, why on earth would I want to make the deleteSong functions private". So that we can use the subtraction operator overload in a way that makes sense! Since deleteSong(int) is private, the only job this function has is to call deleteSong(index). The invocation of this method **must** be done any time a song is removed at a specific index from a playlist. It should look like this:

somePlaylist - userSelectedIndex;

Methods Called: deleteSong(int)

Method:getSongAtIndex
Return Type: Song*
Parameters: index (int)
Pre-Condition: User has selected a song using the selection numbers displayed
Post-Condition: The selected song's pointer is returned
Description: Create a temporary pointer (current) that is the same as the first song pointer. Use a while loop to iterate through the songs in the playlist, which can be done by updating the temporary pointer (current) to be equal to its next song at the end of the while loop. Before that, check if the number of songs visited is equal to the user entered index. If it is, return the current node. If the index is not ever found using the while loop, return a nullptr.
Methods Called: Your choice.

Method: insertion operator overload
Return Type: ostream&
Parameters: outputStream(ostream&), playlist(const Playlist&)
Pre-Condition: Playlist has been filled with songs, either through playlist creation or user file selection.
Post-Condition: output stream is overloaded so that the insertion operator (<<) can be used directly on a playlist, like so: **cout << somePlaylist;**
Description: This should be a friend function, so place the definition in helpers.cpp and don't forget the friend keyword when declaring it in Playlist. This method should use the passed output stream to output a selection number followed by each song directly using the Song insertion operator overload. You will want to create a pointer duplicate of the first song in the playlist, and use it to iterate through all songs in a while loop (see playlist description of iteration above). You'll know you've reached the end of the list when the duplicate pointer is a nullptr. This operator overload must be used **any time the program outputs a playlist with selection values.**
Methods Called: Song insertion operator overload.

==========

You must include the following *helper* functions, they must match the specifications given, and your parameters should be provided in the order listed

Method: readData
Return Type: void
Parameters: inputStream(ifstream&), playlist(Playlist&)
Pre-Condition: input stream opened, playlist initialized as pointer in main
Post-Condition: opened file stream reads the file data into playlist
Description:This function uses inputStream to iterate through the entire file, reading each song's title and artist, and storing them in a new Song object.The function adds the each new song to the playlist using Playlist's addition operator overload.
Methods Called: Playlist's addition operator overload

Method: createNewPlaylist
Return Type: bool
Parameters: allDBSongs(Playlist&), newPlaylist(Playlist&)
Pre-Condition: A playlist containing all songs from the database is passed, as well as an empty playlist.
Post-Condition: newPlaylist is filled with user selections, and whether the user chooses to exit is returned.
Description: Displays all available songs, an option to return to main menu or exit, and a prompt to make a song selection. The user is repeatedly prompted until they choose to return or exit, and then the playlist is saved by playlist name with the .txt extension in the current (.) directory.
Methods Called: Your choice.

Method: deleteSongFromPlaylist
Return Type: bool
Parameters: playlist(Playlist&)
Pre-Condition: playlist is initialized and stores Songs.
Post-Condition: playlist is updated with songs removed and txt file is updated to reflect the changes.
Description: Displays all of the songs in the playlist, an option to return to main menu, and an option to exit the program. Prompts user repeatedly to enter a song to remove until either all songs are removed, or the user chooses to exit or return. Songs selected for removal are deleted using the subtraction operator overload in the Playlist class. Then whether the program should fully exit is returned from the function.

If the playlist is empty because all songs are removed, the playlist should be removed using the remove function linked above (you will need to use .c_str() on the filename string to get it to work), and the user should be prompted if they want to return to main menu. The program should then return the boolean representing whether the program should fully exit.

Methods Called: Your choice, subtraction operator overload

Method: addSongToPlaylist
Return Type: bool
Parameters: playlist(Playlist&), allSongsInDb(Playlist&)
Pre-Condition: playlist is initialized, allSongsInDB has loaded database songs.
Post-Condition:playlist is filled with selected songs from the database
Description: displays all songs in the database, then prompts user to return to menu, exit program, or select a song. Once a song is selected, prompts user if they would like to add to the end of the playlist. If yes, then the Playlist addition operator overload is used to add it to the end. Otherwise, the Playlist add method prompts for which song to add before, and uses the add method to add it there. Repeats song prompting until the user chooses to return or exit. Then the function should return the appropriate boolean value to indicate exit to main.
Methods Called: Your choice, additionOperatorOverload

Method: deletePlaylist
Return Type: bool
Parameters: playlist(Playlist)
Pre-Condition: playlist exists on disk and has been loaded
Post-Condition: playlist is removed from disk
Description: uses the remove method (discussed above) to remove the playlist from disk. Prompts the user to return to main, or exit, and then returns appropriate boolean.
Methods Called: Your choice

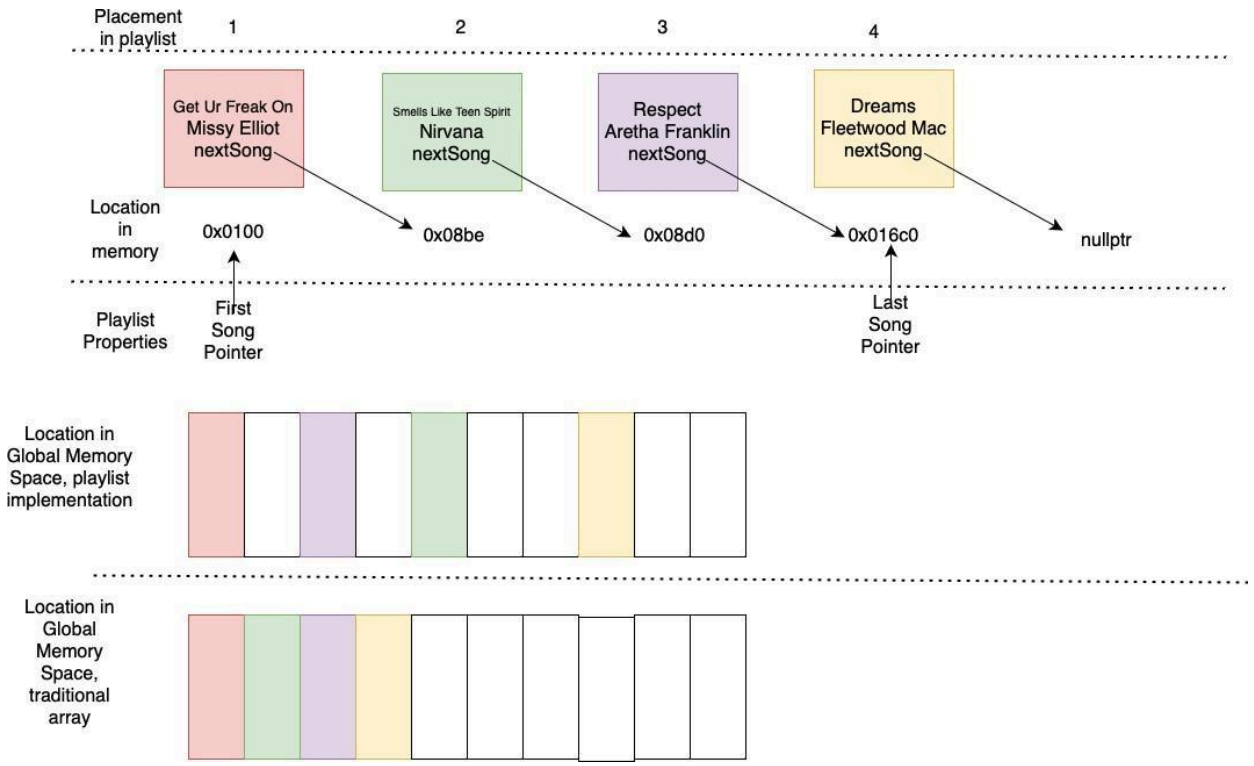

Figure I. A comparison of where our playlist class stores entries in the global memory space, vs how a normal array would store them.

As you can see, our implementation is vastly different from a traditional array. Arrays store items at contiguous memory address spaces in the global memory space, which is why we can iterate through an array using brackets ([0] for example). Instead of allocating all of our songs to an array, we will allocate each song a new memory space assigned with the **new** keyword. Then, we link every item in the class using a series of pointers, which allows us to allocate non-contiguously.

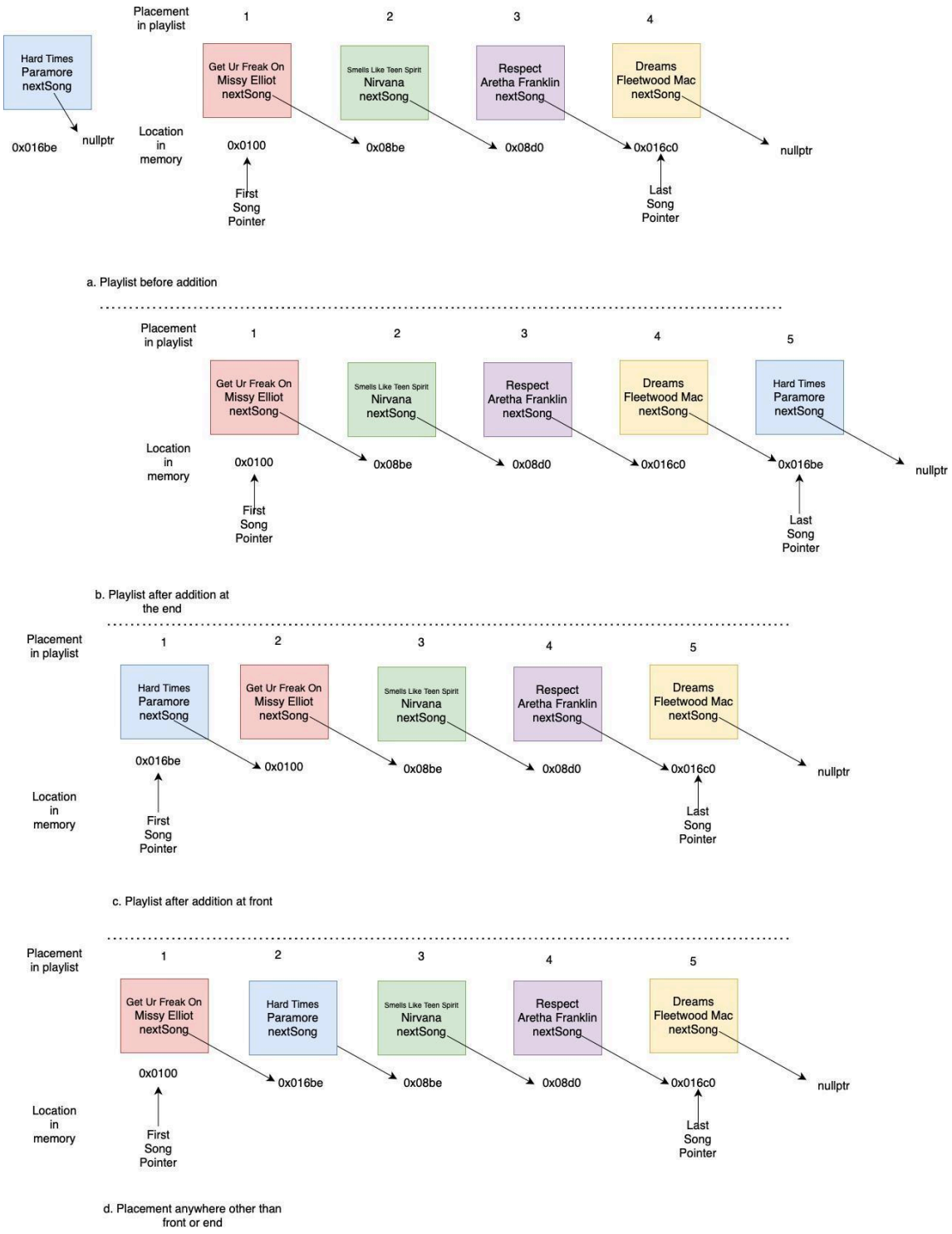

Figure II. A comparison of different places to insert in our playlist.

Before any Song is added to the playlist, the first song is Get Ur Freak On, and the last song is Dreams (a). If we insert a song into the playlist at the end of the playlist (b) then the last song pointer property in the Playlist class must be updated to point at the newest song's memory

address, and the previous last song (Dreams) must set its next song to be the new song that we've inserted. If we insert a song into the playlist at the beginning of the playlist (c) then the first song pointer property in the Playlist class must be updated to point at the newest song's memory address, and the newest song's next song pointer must updated to point to the song that was previously the first song in the playlist. If we insert a song anywhere else in the playlist(d), then the song before must update it's next song pointer to be the new song's pointer, and the new song's pointer must be updated so that its next song pointer points to the previous song's previous next song pointer.

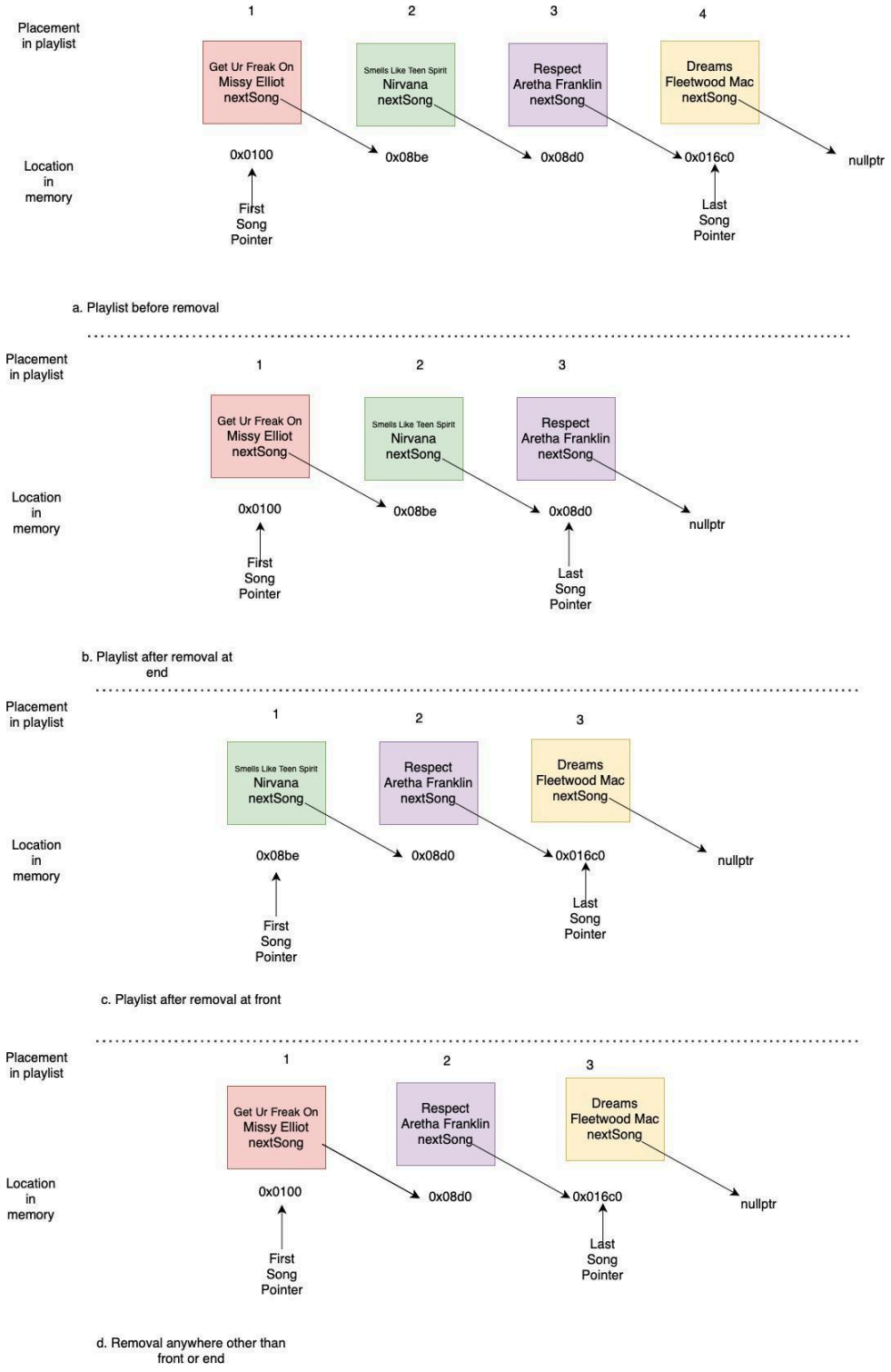

Figure III. A comparison of different places to remove from our playlist.

Before any Song is removed from the playlist, the first song is Get Ur Freak On, and the last song is Dreams. If we remove a song from the end of the playlist (b), then the song that was

previously the last in the list must free its memory space with the **delete** keyword. Then the song before the song that is being deleted must adjust its next song pointer to be a nullptr, and the last song pointer property must be updated so that it is now pointing to the song that was second to last prior to removal. If we remove a song from the front of the list (c), then the first song property of Playlist must be updated so that it points to the first song's next song address. Then we must free the old first song's memory with the delete keyword. If we remove a song from anywhere else in the Playlist, the song before the removed song must be adjusted to point at the removed song's next song, freeing the memory space of the removed item afterward.