Assignment 2: MySongPlayer

Josh W. Brook

November 21, 2021

1 Data Structures

1.1 MySong

Firstly, I implemented a new struct to store a song, which can hold both a song *tune* and an int *played*. The *song* struct is defined in the playlist.h interface, and contains an int *id*, a char* *title*, and a float *duration*. Since we are working with interfaces, I cannot alter the orginal song structure definition, but I need to add a new integer value (0 or 1), to record whether a song has been played yet. As such, my playlist stores an array of *mysong* rather than the original *song*.

1.2 Playlist

I defined another struct to store the playlist of songs, as well as some other values to keep track of the playlist functions. I chose to implement the playlist as a queue-style dynamic array, as items within it can be efficiently added and accessed. More specifically, these operations can be performed in constant time O(1). This is far better than using a linked list, which has O(n) complexity for adding and searching. The one downside of using a dynamic array is that it must be infrequently copied, doubling in size, to make space for more items. Copying runs in linear time, O(n), but since we assume that the playlist should be very static, this is not a big issue. The structure contains a dynamic array of *mysong*, Q, as well as two integers to store the positions of the *first* and *last* elements. Since we are not removing from the playlist, *first* is generally unused. The playlist struct also contains integers to store the *size* of the allocated array, the position of the *next* song to be played, and the *mode* that the playlist should play in. This *mode* can be adjusted with the function `skipAllPlayedSongs` and dictates whether the paylist will automatically skip previously played songs.

2 Implementation of Functions

2.1 PlaySong

The function **playSong** takes a playlist as input and returns the id of the next song to be played. If `p->mode == 0`, the function sets the *played* variable attached to the specific *mysong* to 1, showing that the song has been played. It then runs `findnext(p)`, which either increments `p->next` or sets it back to 0 if the end of the playlist is reached. I decided to find the next song at this stage since the playlist should be very static, meaning that most songs will be added at the start of the program. This means that if the last song is played and then another is added, the playlist will still play from the beginning. If `p->mode == 1`, and `p->next` hasn't been *played*, **playSong** runs as before, outputting the id of the next song to be played. If `p->next` has already been played, the function loops through the rest of the playlist, returning the id of the next unplayed song or -1 if all the songs have been played. When the playlist is in mode 0, this function should have constant complexity O(1), as the array can be accessed directly. When in mode 1, the worst case complexity is O(n), as the function may have to search through the entire array linearly before it can say there are no songs left to be played. Unfortunately, this can't be improved upon wthout drastic structure changes, as the playlist isn't sorted so binary search cannot be implemented.

2.2 PlayFrom

The function **playFrom** takes the playlist p and an integer i, representing the position to play from. If `i-1 <= p->last`, the function sets `p->next` to *i - 1*, meaning *i - 1* will be played next. I check *i - 1* rather than *i* as the playlist indexes from 0, but the input integer counts from 1, as a human normally would. Since the playlist is very static and I defined the variable *next* within the playlist struct, playing from any given position is relatively simple. This function has constant complexity O(1), as the size of the playlist does not affect the runtime.