Assignment 1 - Linear Data Structures

Software engineering involves more than just programming. It involves analysing and modifying existing software as well as designing, constructing and testing end-user applications that meet user needs. The current course, Data Structures and its Applications plays a major role in the low-level design of any software system by choosing the appropriate data structures in the right places, for efficient implementation. Let us look at one such system and try to make it efficient by implementing it with appropriate data structures.

For this assignment, we are working on a music-streaming system(like Spotify). The current system has been identified to have a few bugs in a few places, so we need to write the code for the functions described in the document later. But first, let us get to know more about the system we will be working on.

Here are some of the logical units and terminologies of the system:

1. *Song*: Each song is identified by a unique non-negative integer identifier.

2. *Playlist*: A playlist is a collection of songs. We can add songs, delete songs and search for songs from it.

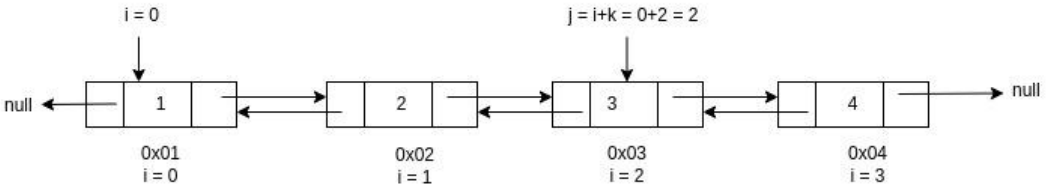
3. *Play Queue*: A song queue, which specifies the prioritised order in which the songs need to play next. If the queue is non-empty, the next song to be played is always from this queue and not from the playlist.

4. *Shuffling*: Shuffle the order of songs stored in the playlist. Shuffling is done by a combination of KSwap and reverse operations.

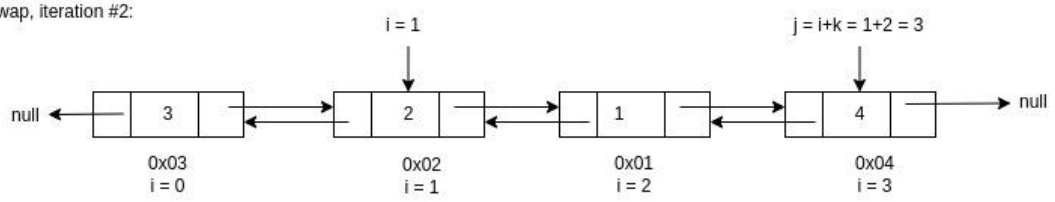
5. *KSwap*: Given an integer k, swap the node at position i with the node at position i+k up to the point where i+k is less than the size of the linked list i.e swap for the nodes that satisfy: 0 <= i+k < n where n is the length of the linked list, given 0 <= i < n and 0 <= k < n

Example for a shuffle operation: (The swaps and changes to the linked list should be done on nodes, and just value. Carefully observe the address values below nodes at each iteration in the image)

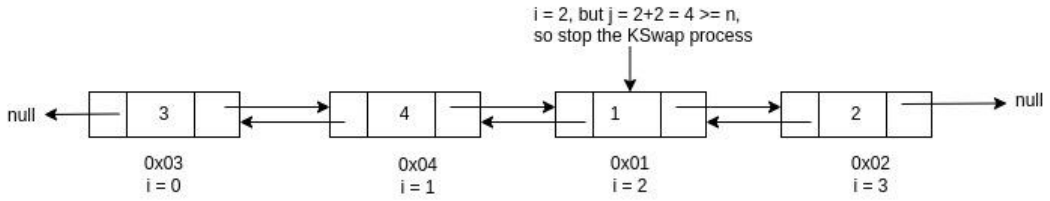
1) KSwap, iteration #1:



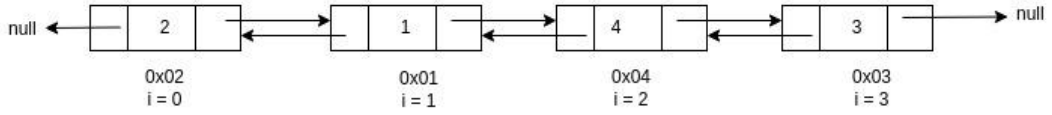
2) KSwap, iteration #2:



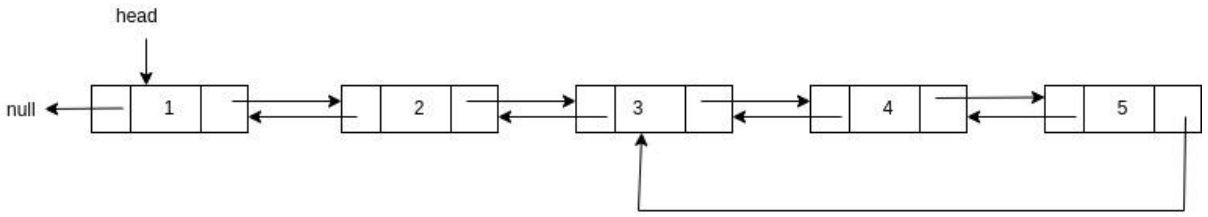
3) KSwap, iteration #3:



4) Reverse:



6. *Loops*: Few of the old playlists are found to have a loop or a cycle in the linked list due to buggy implementation.
   Example linked list with a cycle or loop:



   Here the node with data = 3 is the “start of the cycle or loop”.

Choice of data structures for implementing the problem at hand:

1) *Playlist*: The ideal data structure of choice for modelling a playlist would be a doubly linked list. This would come in handy as new songs can be added or the old ones can be removed from the playlist at any point in a doubly linked list. Also, it should support playing(querying) the previous song, next song and (current + i)'th song on the playlist.

2) *Play Queue*: A queue abstract data-type to follow FIFO. The choice of implementation of the ADT can be either a linked list or an array. For this assignment, we stick on to the linked list implementation of the queue as it gives us the flexibility of dynamically changing the size of the data structure.

Expected deliverables:

In a nutshell, the deliverables are **completing the implementations of given functions in 3 files**: dll.c which will be like a library of doubly linked list functions, queue.c which will be like a library of queue functions and music_player.c which is the implementation file which makes use of the functions from dll.c and queue.c, for its own functions described below.

1. dll.c:
    a. Implementation file for all the basic doubly linked lists operations.
        i. **void insert_front(list_t* list, int data);** // inserts data to the beginning of the linked list
        ii. **void insert_back(list_t* list, int data);** // inserts data to the end of the linked list
        iii. **void insert_after(list_t* list, int data, int prev);** // inserts data after the node with data as "prev"
        iv. **void delete_front(list_t* list);** // delete the start node from the linked list.
        v. **void delete_back(list_t* list);** // delete the end node from the linked list.
        vi. **void delete_node(list_t* list, int data);** // delete the first occurrence of the node with data as "data" from the linked list.
        vii. **node_t* search(list_t* list, int data);** // returns the pointer to the node with data as "data" field

2. queue.c: // make use of the already implemented functions in dll.c
    a. Implementation file for all the basic queue operations: (Students have the liberty to either use array or linked list implementation of the queue abstract data type)
        i. **void enqueue(queue_t* q, int data);** // insert data to the queue
        ii. **int dequeue(queue_t* q);** // return the data at the front of a queue and also delete it. Return -1 if queue is empty
        iii. **int front(queue_t* q);** // return the data at the front of a queue. Return -1 if the queue is empty

3. music_player.c: // make use of the functions in dll.c and queue.c
    a. Implementation file for all the music player platform operations:
        i. **void add_song(playlist_t* list, int song_id, int where);** // add a song to the playlist:
            1. If where is -1, add the song to the start of the playlist
            2. If where is -2, add the song to the end of the playlist
            3. If where is non-negative, add the song after the song with id = where in the playlist.
        ii. **void delete_song(playlist_t* playlist, int song_id);** // remove the song with given song id from the playlist
        iii. **song_t* search_song(playlist_t* playlist, int song_id);** // return a pointer to the node where the song_id is present in the playlist
        iv. **void play_next(playlist_t* playlist, music_queue_t* q);** // play the next song in the linked list with reference to the last played song or play from the queue considering the following rules:
            1. play the next song in the linked list if the queue is empty.
            2. If the queue is not empty, play from the queue rather than the next song in the list.
            3. If there was no song that was played last, play the first song on the list.
            4. If there is no next song in the playlist, play the first song of the playlist and follow the list order for further play_next calls.
            5. If there are no songs yet in the playlist, return from the function without doing anything.
        v. **void play_previous(playlist_t* playlist);** // play the previous song in the linked list with reference to the last played song.
            1. If the previous song in the playlist is not present, play the last song of the list and follow the reverse order of the list from the last song for further play_previous calls.
            2. No need to check for previously played songs from the queue(would have been removed from the queue), i.e always play the previous song from the playlist when this function is called.
            3. If there was no song that was played last, play the last song on the list.
            4. If there are no songs yet in the playlist, return from the function without doing anything.
        vi. **void search_and_play(playlist_t* playlist, int song_id);** // play the song with the given song_id from the list(no need to bother about the queue. Call to this function should always play the given song and further calls to play_next and play_previous)
        vii. **void play_from_queue(playlist_t* q);** // play a song from the queue
        viii. **void insert_to_queue(music_queue_t* q, int song_id);** // insert a song id to the queue
        ix. **void reverse(playlist_t* playlist);** // reverse the order of the songs in the given playlist. (Swap the nodes, not data)

x. **void k_swap(playlist_t* playlist);** // swap the node at position i with the node at position i+k up to the point where i+k is less than the size of the linked list

xi. **void shuffle(playlist_t* playlist, int k);** // perform k_swap and reverse in the order k_swap first and reverse next.

xii. **song_t* debug(playlist_t* playlist);** // if the given linked list has a cycle, return the start of the cycle, else return null. Check cycles only in the forward direction i.e with the next pointer. No need to check for cycles in the backward pointer.

Instructions:

1. Playing a song should be simulated by calling the void play_song(int song_id); function which is already implemented.
2. Any boundary conditions in which you feel that you need to include some print statement(Example: underflow or overflow or linked list is empty etc), **shouldn't print anything to the STDOUT** explicitly. Rather just return from the function without doing anything unless explicitly specified.
3. **Do not take any inputs from STDIN** and **do not print anything to the STDOUT**.
4. The operations implemented as a part of the libraries should handle all the cases well.
5. **Remove all the printf and scanf statements** used for debugging purposes except the ones already provided.
6. **Do not change any code already given**, especially the client file, function prototypes and struct declarations.
7. We recommend you test your implementation on any Unix/Linux(Ubuntu)/macOS operating systems with either gcc or clang compilers of any versions. C compilers on Windows tend to bypass segmentation fault errors which plays a major role when dealing with linked lists and dynamic memory allocations.
8. Free the dynamically allocated memory wherever necessary. Avoid dangling pointers and memory leaks which may lead to segfaults.
9. We have provided you with the header files, a client file, makefile(s), two sample inputs and expected outputs. Please **write your implementation based on these files ONLY.**
   Link to the files: A1
   Use "makefile" for linux/unix/macOS and "makefile.mk" for Windows.
   Makefile commands for OSes except Windows:
   1. **make compile** - Creates an executable
   2. **make run** - Runs the executable
   3. **make clean** - deletes the executable and object files.

   Makefile commands for Windows OS:
   1. **make -f makefile.mk compile** - Creates an executable
   2. **make -f makefile.mk run** - Runs the executable
   3. **make -f makefile.mk clean** - deletes the executable and object files.