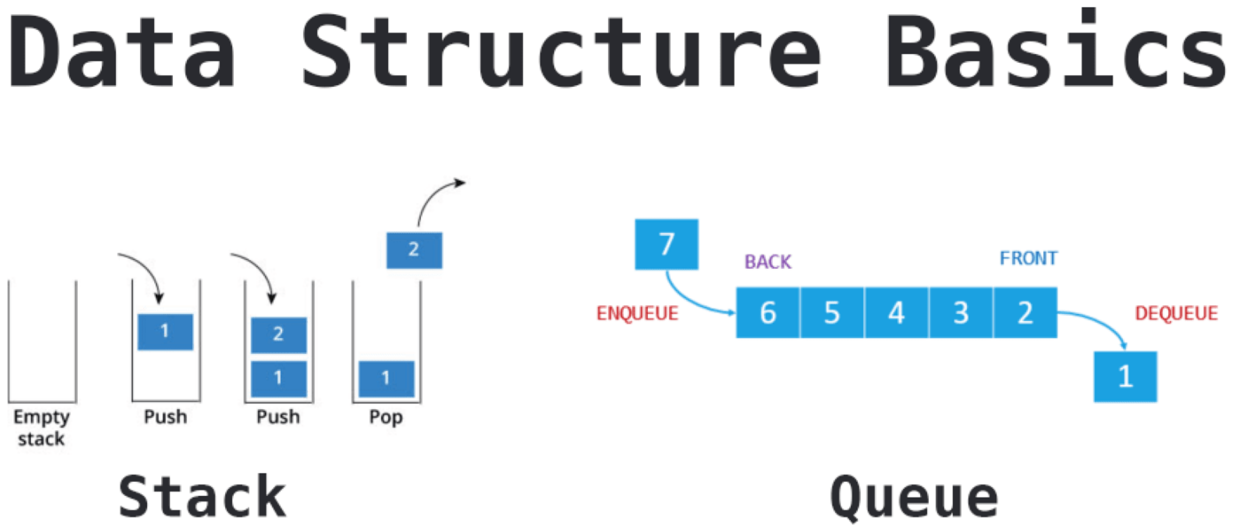
1.) Identify the best-suited data structure for a given situation and do compare/contrast analysis with other data structures to justify why it was the "best" solution.

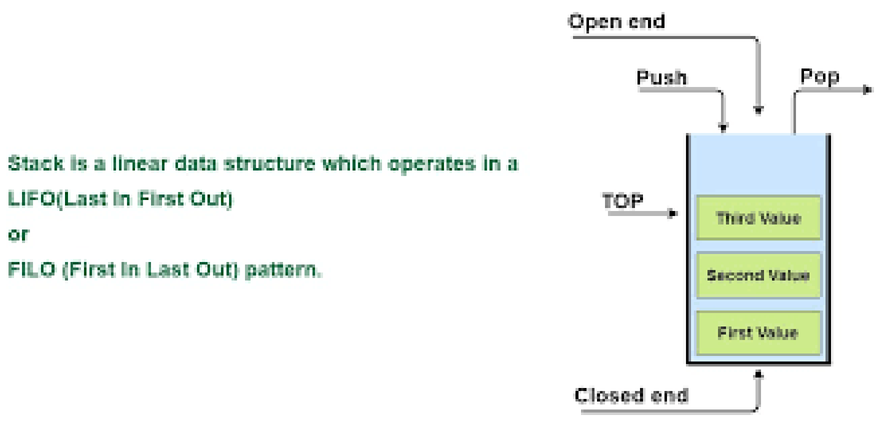
- The given situation I will be doing an analysis on is the action of parsing raw playlist data as I am doing in my project. I send a POST request to the Spotify API and receive a chunk of the raw data concerning the playlist I'm trying to archive within a .json file. I process this data by splitting each individual track into its own object with four properties (title, artist, album, link).
- As I separate the tracks into their own respective objects, I then store each object in a queue (with enqueue()), and then move on to the next track.
- Once all objects have been stored in the queue, I then remove each object from the queue to place it into the final .json output file for the program (with dequeue()).

- I chose a queue as the best-suited data structure for this specific situation thanks to the fact that it is a FIFO (first in first out) data structure. This allowed me to only have to use two very simple calls when dealing with the data structure, and made the code extremely simple to implement. I just had to call "enqueue()" every time I got a new object, and then "dequeue()" to remove the elements one by one.
- There was no traversal or any other type of function necessary to be called on the queue, since I knew they would be removed in the exact same order I initially stored them. This allowed me to preserve the initial order of the tracks within the playlist when outputted to the json file with minimal effort.

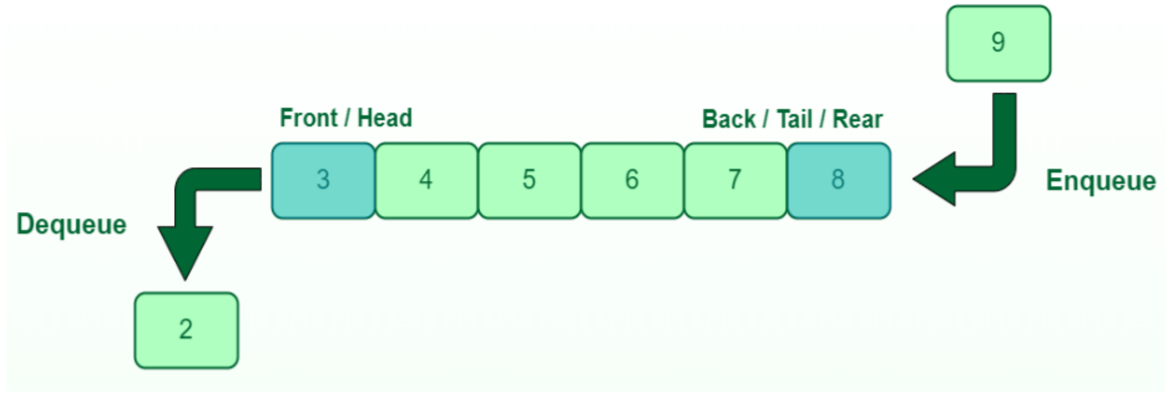
- For this reason, one of the worst data structures I could have used for this task would have been a stack. If I would have used a stack for this task, the tracks from the playlist would have either been outputted in reverse order, or I would have had to waste a large amount of compute in rearranging the stack every single time I needed to get the next object, which would have always existed at the very bottom of the stack.
- If we were just trying to archive a collection of songs into one file without worrying about their order, a stack would have worked fine, but part of what makes a playlist a playlist is the order in which the songs take place - not just the songs that exist within it.

- I will attach illustrations of examples of queues and stacks below to further illustrate the difference, and why I chose a queue for this specific task.





Stack is a linear data structure which operates in a LIFO(Last In First Out) or FILO (First In Last Out) pattern.



Queue Data Structure

- With that being said, using a regular linked list, or really any version of a linked list that is not a queue, would have also been far more difficult to work with for the simple fact that they don't come with built-in functionality for "enqueue()" and "dequeue()".
- If we were to use a basic linked list, we would have to deal with reassigning pointers after every removed element, since that functionality does not come built in with specific functions like queues and stacks. Implementing these functions would have taken considerably more time and effort than it took to just use a queue.

- A binary tree/tree/graph would not have been optimal or really worked either since one of the main points was archiving the songs in order, and none of those data structures are optimal for maintaining order of elements along a singular line.