Suggest and Justify appropriate DS for the below Examples

Data Structure Overview:

1. **Array**: A simple, fixed-size collection where elements are stored contiguously. Useful when random access is needed and the number of elements is known.
2. **Linked List**: A dynamic data structure where elements (nodes) are connected through pointers. Preferred when frequent insertions and deletions are required.
3. **Singly Linked List**: A type of linked list where each node points to the next, ideal for simple, linear data that is processed in one direction.
4. **Circular Linked List**: Similar to singly linked list but the last node points to the first, used for applications like round-robin scheduling.
5. **Doubly Linked List**: Each node contains two pointers (next and previous), allowing traversal in both directions. Great for undo/redo operations.
6. **Stack**: A Last In, First Out (LIFO) data structure. Good for scenarios where you need to reverse processes or backtrack (e.g., web browser history).
7. **Queue**: A First In, First Out (FIFO) structure. Used for tasks that need sequential processing, such as ticketing systems.
8. **Circular Queue**: A queue where the end connects back to the front, enabling efficient use of space in applications like buffering.
9. **Priority Queue**: Elements are processed based on priority, not just order. Suitable for managing tasks with varying importance.
10. **Double-Ended Queue (Deque)**: A queue where elements can be added or removed from both ends, useful for scenarios like task scheduling.
11. **Binary Tree**: A hierarchical structure where each node has up to two children. Useful for structured data like expression trees or hierarchical organizations.

12. **Threaded Binary Tree**: A binary tree with additional pointers to make traversal faster by filling in "null" links with pointers to the next node in traversal order.
13. **AVL Tree**: A self-balancing binary search tree. Efficient for searching, inserting, and deleting data with balanced height.
14. **B-Tree**: A balanced tree structure commonly used in databases for storing large amounts of data in a sorted manner.

Example Questions:

1. **Question: Suggest a data structure for the 'Continue watching' list in an OTT app platform.**
   Answer: A **Singly Linked List** would be suitable here because the list grows dynamically as users add new shows, and elements can be easily added or removed when a user starts or finishes watching a show. The simple linear structure fits well since there's no need to go back and forth in this case, making traversal straightforward.

2. **Question: What data structure would be ideal for managing entry and exit from a stadium that has gates for both entry and exit?**
   Answer: A **Double-Ended Queue (Deque)** is perfect here. Gates allow both entry and exit simultaneously, which is mirrored by the deque allowing insertion and deletion from both ends.

3. **Question: You need to store user commands in an Undo/Redo feature of a text editor. Which data structure would you choose?**
   Answer: A **Doubly Linked List** is ideal because it allows traversing in both directions (undo and redo), enabling efficient command reversal and restoration.

4. **Question: You're developing an elevator system that processes requests sequentially. What data structure should you use?**
   Answer: A **Queue** is best because requests are processed in a First In, First Out (FIFO) manner, ensuring that the first request made is served first, just like passengers waiting for an elevator.

5. **Question: A print spooler needs to manage multiple print jobs where priority should be given to more urgent jobs. What would you choose?**
   Answer: A **Priority Queue** fits this situation. Jobs can be enqueued with different priorities, ensuring that high-priority jobs (urgent prints) are processed first.

6. **Question: In a music player app, songs are added to the playlist dynamically, and the user should be able to repeat the playlist infinitely. Which data structure is appropriate?**
   Answer: A **Circular Linked List** is best because it allows seamless repetition of the playlist, where the last song points back to the first.

7. **Question: A program requires managing the order of function calls, ensuring the last function called is the first one to be processed. What do you use?**
   Answer: A **Stack** is ideal because of its Last In, First Out (LIFO) behavior, which matches the function call stack mechanism in many programming environments.

8. **Question: You’re creating a scheduler for a multitasking operating system where each task must be processed in a**

round-robin manner. What would you choose?
Answer: A **Circular Queue** is perfect since it enables tasks to be cycled through efficiently, ensuring each task gets an equal share of CPU time.

9. **Question: An online reservation system needs to manage a limited number of tickets for an event where customers are processed on a first-come, first-served basis. What is your pick?**
   Answer: A **Queue** is appropriate here because customers should be processed in the order they requested tickets, maintaining a FIFO structure.

10. **Question: You're building a contact tracing app that keeps track of people a user has interacted with over a period of time. The data must be easily expandable. Which structure suits this?**
    Answer: A **Linked List** is appropriate since contacts are dynamic, and the list can grow or shrink as needed without reallocating memory.

11. **Question: For an autocomplete feature in a search bar, which data structure would you use to store the search history?**
    Answer: A **Stack** would be appropriate because the most recent search term is likely the most relevant, and the user should be able to backtrack through their previous searches.

12. **Question: In a mobile game leaderboard, you need to efficiently store and retrieve high scores in sorted order. Which data structure is the best?**
    Answer: A **Binary Tree** or more specifically a **Binary Search Tree**

is suitable, as it allows efficient insertion and retrieval of scores in sorted order.

13. **Question: A file system is required to store and retrieve files in a hierarchical manner. What is the ideal data structure?**
    Answer: A **Tree** structure (specifically a **Binary Tree**) would be ideal, as it mirrors the hierarchy of directories and subdirectories in a file system.

14. **Question: In a hospital emergency room, patients must be treated based on their severity level. What data structure should be used?**
    Answer: A **Priority Queue** is ideal since patients with more severe conditions can be prioritized over less severe cases.

15. **Question: You need to develop an algorithm to parse mathematical expressions like "3 + 5 * (2 - 8)". Which data structure would you choose?**
    Answer: A **Stack** would be the best choice here because it can handle the nested parentheses and operator precedence efficiently during parsing.

16. **Question: You're tasked with creating a data structure for a virtual memory paging system, where pages should be replaced in a circular manner when the memory is full. What would you select?**
    Answer: A **Circular Queue** is the right fit, as it allows pages to be replaced in a cyclic order without using extra memory.

17. **Question: A navigation system needs to backtrack through routes and decisions taken to reach a destination. Which structure would suit this best?**

Answer: A **Stack** is ideal since it can keep track of the decisions made and allows the system to backtrack when necessary.

18. **Question: In a messaging app, messages between two users should be displayed in the order they were sent. What structure would you use?**
    Answer: A **Queue** would be the best fit since messages need to be displayed in the order they were sent (FIFO).

19. **Question: A social media platform requires storing likes and comments in a way that allows users to quickly access the most recent activity. Which data structure should be chosen?**
    Answer: A **Stack** would work well, as the most recent activities (likes and comments) should be easily accessible.