1. Why is a ring buffer useful and/or when should it be used?

Usage: Ring buffers are particularly useful for their efficiency in managing data in a FIFO (First-In-First-Out) manner without the need for data shifting, which can be costly in terms of performance. excel in scenarios requiring constant time operations $O(1)$ for adding and removing elements, without the overhead of shifting data in the structure. This makes them ideal for use cases where data is continuously produced and consumed at different rates, such as in streaming data applications, real-time systems, and buffering for hardware interfaces. The circular nature of ring buffers allows for optimal use of memory; once the buffer is filled, the oldest data is overwritten by new data if it has not been consumed yet. This characteristic is beneficial in embedded systems or any system where memory is limited and must be efficiently managed.

Pros: The primary advantage of ring buffers is their efficiency in memory use and performance, especially in constrained environments like embedded systems. They eliminate the need for dynamic memory allocation, reducing overhead and the risk of memory leaks. This makes them ideal for high-speed data transmission and real-time processing tasks.

Cons: However, ring buffers have a fixed size, which can be a limitation if the buffer size is not adequately estimated, leading to potential data loss when the buffer overflows or underutilization of memory if the buffer is too large. Furthermore, the circular logic requires careful management to avoid errors in wrapping the buffer's endpoints.

Reference:

Circular Queue or Ring Buffer: https://towardsdatascience.com/circular-queue-or-ring-buffer-92c7b0193326

2. Why is a stack useful and/or when should it be used?

Usage: Stacks are inherently simple yet powerful structures, with their LIFO operation principle making them ideal for scenarios such as function call management in programming languages and expression evaluation. The complexity of push and pop operations is O(1), offering efficient and predictable performance. A classic example of stack usage is in the execution of recursive algorithms, where each function call's state is preserved until it returns. Stacks are also used in evaluating and parsing expressions in programming languages and calculators, where operators and operands are pushed and popped from the stack to maintain correct order and precedence.

Pros: Stacks provide simplicity and efficiency, especially in scenarios requiring the last-in-first-out access pattern. This makes them invaluable in managing execution contexts in programming languages, where call stacks track function calls and returns. Their ease of implementation and operation makes them a fundamental tool in software development.

Cons: The limitation of stacks lies in their inability to access non-top elements without removing the preceding elements, which can be restrictive for certain applications requiring access to arbitrary elements in the data structure. This necessitates alternative data structures like arrays or linked lists where such access patterns are needed, increasing complexity.

Reference:

Stacks and Its Applications: https://byjus.com/gate/stack-and-its-applications/