CodInClub
Powered By BridgeLabz

# Best Practices for Stacks and Queues

## Stacks

1. **Use for Reversible or Nested Problems**:
   Stacks are ideal for problems involving recursion, backtracking, or nested structures (e.g., balanced parentheses, undo functionality).
2. **Optimize Stack Size**:
   Avoid memory overflows by setting a proper size for stacks in fixed-size implementations, or use dynamic structures (like Java's `Stack` class) for scalability.
3. **Avoid Infinite Loops in Recursive Algorithms**:
   Ensure a clear base case in recursive stack operations to prevent stack overflow errors.
4. **Push and Pop Atomically**:
   When dealing with multi-threaded environments, ensure stack operations are atomic to avoid race conditions. Use synchronized stacks like `java.util.concurrent.ConcurrentLinkedDeque` in Java.
5. **Check Stack Underflow and Overflow**:
   Always validate operations to avoid popping an empty stack or pushing into a full stack (if the stack has a fixed size).
6. **Use Collections Framework for Robustness**:
   Instead of implementing stacks from scratch, use robust implementations like `Deque` or `LinkedList` from Java's Collections Framework for better performance and maintainability.
7. **Track the Minimum or Maximum Value**:
   For problems where you frequently need the minimum or maximum element, maintain an auxiliary stack to store these values for O(1) retrieval.

---

## Queues

1. **Use for FIFO (First In, First Out) Problems**:
   Queues are well-suited for sequential processing problems, like task scheduling, breadth-first search (BFS), and producer-consumer scenarios.
2. **Choose the Right Type of Queue**:
   - **Simple Queue**: For basic FIFO needs.
   - **Deque (Double-Ended Queue)**: For flexibility to add/remove from both ends.
   - **Priority Queue**: When elements must be processed based on priority rather than order.

3. **Optimize Memory Usage**:
   When using circular queues, keep track of head and tail pointers efficiently to avoid wasting memory.
4. **Handle Concurrency with Thread-Safe Queues**:
   In multi-threaded environments, use thread-safe implementations like `BlockingQueue` or `ConcurrentLinkedQueue`.
5. **Validate Queue Underflow and Overflow**:
   Ensure proper handling of scenarios where the queue is empty (during dequeue operations) or full (in fixed-size queues).
6. **Lazy Deletion for Priority Queues**:
   When frequent deletions are involved, mark elements as deleted and process cleanup later to avoid immediate restructuring costs.
7. **Avoid Polling Empty Queues**:
   Always check if the queue is empty before dequeue operations to avoid exceptions or errors.

# Sample Problems for Stacks and Queues

1. **Implement a Queue Using Stacks**
   - **Problem:** Design a queue using two stacks such that enqueue and dequeue operations are performed efficiently.
   - **Hint:** Use one stack for enqueue and another stack for dequeue. Transfer elements between stacks as needed.
2. **Sort a Stack Using Recursion**
   - **Problem:** Given a stack, sort its elements in ascending order using recursion.
   - **Hint:** Pop elements recursively, sort the remaining stack, and insert the popped element back at the correct position.
3. **Stock Span Problem**
   - **Problem:** For each day in a stock price array, calculate the span (number of consecutive days the price was less than or equal to the current day's price).
   - **Hint:** Use a stack to keep track of indices of prices in descending order.
4. **Sliding Window Maximum**
   - **Problem:** Given an array and a window size `k`, find the maximum element in each sliding window of size `k`.
   - **Hint:** Use a deque (double-ended queue) to maintain indices of useful elements in each window.
5. **Circular Tour Problem**
   - **Problem:** Given a set of petrol pumps with petrol and distance to the next pump, determine the starting point for completing a circular tour.

- **Hint:** Use a queue to simulate the tour, keeping track of surplus petrol at each pump.

---

# Sample Problems for Hash Maps & Hash Functions

1. **Find All Subarrays with Zero Sum**
   - **Problem:** Given an array, find all subarrays whose elements sum up to zero.
   - **Hint:** Use a hash map to store the cumulative sum and its frequency. If a sum repeats, a zero-sum subarray exists.
2. **Check for a Pair with Given Sum in an Array**
   - **Problem:** Given an array and a target sum, find if there exists a pair of elements whose sum is equal to the target.
   - **Hint:** Store visited numbers in a hash map and check if `target - current_number` exists in the map.
3. **Longest Consecutive Sequence**
   - **Problem:** Given an unsorted array, find the length of the longest consecutive elements sequence.
   - **Hint:** Use a hash map to store elements and check for consecutive elements efficiently.
4. **Implement a Custom Hash Map**
   - **Problem:** Design and implement a basic hash map class with operations for insertion, deletion, and retrieval.
   - **Hint:** Use an array of linked lists to handle collisions using separate chaining.
5. **Two Sum Problem**
   - **Problem:** Given an array and a target sum, find two indices such that their values add up to the target.
   - **Hint:** Use a hash map to store the index of each element as you iterate. Check if `target - current_element` exists in the map.