Queues Cheatsheet

Topic Overview

Queues in Java are FIFO (First In, First Out) structures, used for task scheduling and breadth-first search. This cheatsheet covers queue-based techniques.

Prerequisites

Arrays, Linked List

List of Subtopics

- Array-Based Queue
- Linked List-Based Queue
- Enqueue Operation
- Dequeue Operation
- Front Operation
- Check Empty
- Circular Queue
- Deque (Double-Ended Queue)
- Sliding Window Maximum
- Level Order Traversal

Key Concepts Explained

- **Array-Based Queue**: Uses a fixed-size array with front and rear pointers.
- **Circular Queue**: Wraps around the array end to reuse space.
- **Deque**: Allows additions/removals from both ends.

Approaches to Solve Problems with Step-by-Step Algorithms

- **Array-Based Queue**:
  - **Algorithm**:
    1. Initialize an array with a max size, front and rear at -1.
    2. Use front to track dequeue, rear for enqueue.
  - **Context**: O(1) time, O(n) space, limited by size.
- **Linked List-Based Queue**:
  - **Algorithm**:
    1. Use nodes with data and next pointers, front and rear pointers.
    2. Enqueue at rear, dequeue from front.
  - **Context**: O(1) time, O(n) space, dynamic size.
- **Enqueue Operation**:
  - **Algorithm**:
    1. Check if queue is full (array) or create node (linked list).
    2. Increment rear, add element at rear (array) or link new node (linked list).
  - **Context**: O(1) time.
- **Dequeue Operation**:
  - **Algorithm**:
    1. Check if queue is empty, return error if true.
    2. Retrieve front element, increment front (array) or update front to next (linked list).
  - **Context**: O(1) time.
- **Front Operation**:
  - **Algorithm**:
    1. Check if queue is empty, return error if true.
    2. Return element at front without modifying it.
  - **Context**: O(1) time.
- **Check Empty**:
  - **Algorithm**:
    1. Return true if front is -1 or front equals rear+1 (array).
  - **Context**: O(1) time.
- **Circular Queue**:

  - **Algorithm**:
    1. Use modulo arithmetic to wrap indices (rear+1) % size.
    2. Enqueue at rear, dequeue at front, adjust with modulo.
  - **Context**: O(1) time, optimizes space usage.

- **Deque (Double-Ended Queue)**:
  - **Algorithm**:
    1. Use a doubly linked list with front and rear pointers.
    2. Allow enqueue/dequeue at both ends.
  - **Context**: O(1) time, O(n) space.

- **Sliding Window Maximum**:
  - **Algorithm**:
    1. Use a deque to store indices of decreasing elements.
    2. Remove smaller elements from rear, add current index.
    3. Slide window, remove out-of-range indices from front.
  - **Context**: O(n) time, O(k) space for window size k.

- **Level Order Traversal**:
  - **Algorithm**:
    1. Use a queue, enqueue root node.
    2. Dequeue node, process it, enqueue children.
    3. Repeat until queue is empty.
  - **Context**: O(n) time, O(w) space where w is max width.

Common LeetCode Problems with Approaches

- **Implement Queue using Stacks (232)**: Use two stacks for enqueue/dequeue.
- **Sliding Window Maximum (239)**: Use deque for maximum in each window.
- **Rotting Oranges (994)**: Use BFS with queue for spreading rot.
-
- **First Non-Repeating Character (387)**: Use queue with frequency check.
-
- **Number of Recent Calls (933)**: Use queue to track recent calls.

Time & Space Complexities

- Enqueue/Dequeue/Front: O(1)
- Search: O(n)
- Space: O(n)

Important Tips & Tricks

- Use circular queues to optimize array space.
- Handle overflow/underflow with checks.
- Use deques for flexible operations.
- Optimize BFS with queue-based level tracking.
- Preallocate space for array-based queues.