Queue

Alright, my curious coders! We've established that a basic **Queue** is like a standard waiting line: **First-In, First-Out (FIFO)**. You enqueue at the back, and dequeue from the front. Simple, right?

But just like there are different kinds of lines (a normal line, an express lane, a line where you can enter from either end), there are different **types of Queues** designed for specific needs. Let's explore them, one by one!

1. The Simple (Linear) Queue

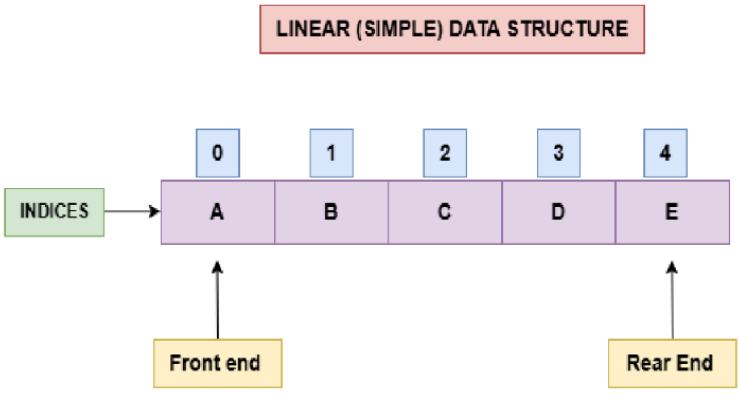
This is the most straightforward representation of a queue, often implemented using a basic array.

Concept:

Imagine a long, empty corridor. People enter from one end and leave from the other. As people leave from the front, the empty space at the beginning of the corridor is usually left unused, and everyone else shifts forward (or the front marker just moves).

- **Front:** Points to the first element.
- **Rear:** Points to the last element.

Visual (Conceptual Array-Backed):



Problem with Simple Array Queue: Wasted Space

Notice how in the example above, even after dequeuing 10 and 20, the slots at index 0 and 1 are empty but can't be used again unless we manually shift all elements forward

(which is an O(N) operation for every dequeue!). This is inefficient and leads to **wasted memory space** if the queue continuously grows and shrinks.

Basic Operations:

- **enqueue(item)**: Add to rear. Check isFull().
- **dequeue()**: Remove from front. Check isEmpty().
- **peek()**: Look at front.
- **isEmpty()**: Check if size is 0.
- **isFull()**: Check if size equals capacity.

When to Use:

- For very small, simple, fixed-size queues where efficiency of space reuse isn't a major concern.
- As a stepping stone to understanding circular queues.

2. Circular Queue

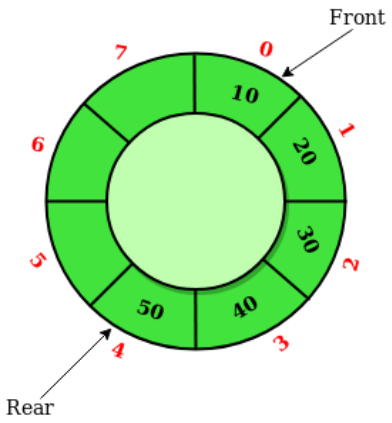
The **Circular Queue** is an elegant solution to the wasted space problem of the simple array-backed queue. It treats the array as a circle.

Concept:

Imagine our corridor, but now its ends are connected, forming a ring. When you reach the "end" of the array, you wrap around to the "beginning" if there's space.

- **Front:** Index of the first element.
- **Rear:** Index of the last element.
- **Modulus Operator (%):** Key to wrapping around. (index + 1) % capacity.

Visual (Conceptual Array-Backed):



Array: [A | B | C | D | E] (Capacity = 5)

Indexes: 0 1 2 3 4

Initial: Front = 0, Rear = -1, Size = 0

Enqueue(10): Rear = (-1 + 1) % 5 = 0. Array: [10| | | |]
Front = 0, Rear = 0, Size = 1

Enqueue(20): Rear = (0 + 1) % 5 = 1. Array: [10|20| | |]
Front = 0, Rear = 1, Size = 2

Enqueue(30): Rear = (1 + 1) % 5 = 2. Array: [10|20|30| |]
Front = 0, Rear = 2, Size = 3

Dequeue(): Item = Array[Front=0] = 10. Front = (0 + 1) % 5 = 1.
Array: [|20|30| |]
Front = 1, Rear = 2, Size = 2

Enqueue(40): Rear = (2 + 1) % 5 = 3. Array: [|20|30|40|]
Front = 1, Rear = 3, Size = 3

Enqueue(50): Rear = (3 + 1) % 5 = 4. Array: [|20|30|40|50]
Front = 1, Rear = 4, Size = 4

Dequeue(): Item = Array[Front=1] = 20. Front = (1 + 1) % 5 = 2.
Array: [| |30|40|50]
Front = 2, Rear = 4, Size = 3

Enqueue(60): Rear = (4 + 1) % 5 = 0. (Wraps around!) Array: [60| |30|40|50]
Front = 2, Rear = 0, Size = 4

Basic Operations (O(1) Time Complexity):

- **enqueue(item)**:
  - if (isFull()) { handle_overflow; }
  - rear = (rear + 1) % capacity;
  - queueArray[rear] = item;
  - size++;
- **dequeue()**:
  - if (isEmpty()) { handle_underflow; }

    - dequeuedItem = queueArray[front];
    - front = (front + 1) % capacity;
    - size--;
    - return dequeuedItem;
- **peek()**: Return queueArray[front]. Check isEmpty().
- **isEmpty()**: return size == 0;
- **isFull()**: return size == capacity;
- **size()**: return size;

Advantages:

- **Efficient Space Utilization:** Reuses dequeued slots, preventing wasted space.
- **Constant Time Operations:** All basic operations remain O(1).
- **Simple Implementation:** Relatively straightforward with modulo arithmetic.

Disadvantages:

- **Fixed Size:** Still has a fixed maximum capacity defined at creation.
- **Empty/Full Distinction:** Requires careful management of front, rear, and size (or leaving one slot empty) to correctly distinguish between a full queue and an empty one.

When to Use:

- When you have a **fixed buffer size** for incoming data (e.g., streaming data, network packets).
- In **resource pooling** where you recycle limited resources.
- When you need constant time performance for adds and removals and can bound the maximum number of items.

3. Priority Queue

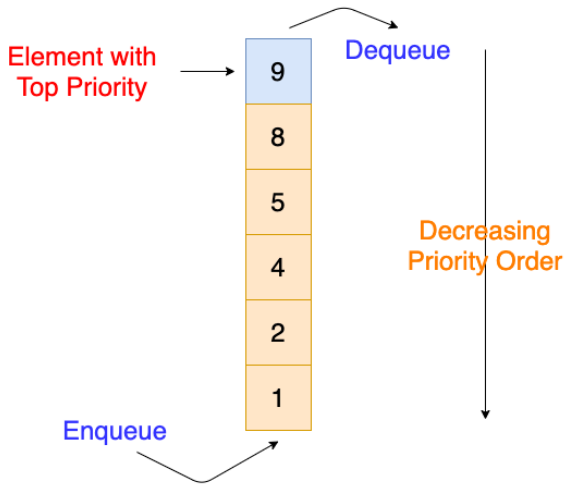
This type of queue breaks the traditional FIFO rule. It's like an emergency room where the most critical patients are seen first, regardless of when they arrived.

Concept:

In a **Priority Queue**, each element has a **priority** associated with it. When you dequeue an item, it's not necessarily the one that has been in the queue the longest, but rather the one with the **highest (or lowest, depending on implementation) priority**.

- **Higher priority items are served before lower priority items.**
- If two items have the same priority, their relative order might follow FIFO or be arbitrary, depending on the implementation.

Analogy: The emergency room triage system, or a printer queue where urgent documents jump ahead of regular ones.



Implementation:

Priority Queues are most commonly implemented using a **Heap** data structure (specifically, a Max-Heap for highest priority first, or a Min-Heap for lowest priority first). Heaps guarantee that adding and removing the highest/lowest priority element are efficient operations.

Basic Operations:

- **add(item) / offer(item)**: Adds an item with its associated priority.
  - *Time Complexity:* O(logN) (due to heap property maintenance).
- **poll() / remove()**: Removes and returns the item with the highest (or lowest) priority.
  - *Time Complexity:* O(logN) (due to heap property maintenance).
- **peek() / element()**: Returns the item with the highest (or lowest) priority *without* removing it.
  - *Time Complexity:* O(1) (just looking at the root of the heap).
- **isEmpty() / size()**: Standard queue methods. O(1).

Advantages:

- **Priority-Based Processing:** Guarantees that the most important tasks or items are handled first.
- **Efficient Priority Operations:** O(logN) for adding and removing, which is very good for dynamic prioritization.

Disadvantages:

- **No Random Access:** Cannot access elements by index.
- **Not Strict FIFO:** Loses the "first-come, first-served" order, which might be a requirement.
- **More Complex Implementation:** Requires understanding of heaps or using built-in classes.

When to Use:

- **Task Scheduling:** In operating systems or real-time systems where tasks have different urgencies.
- **Event Simulation:** Processing events in the order of their occurrence time.
- **Graph Algorithms:** Like Dijkstra's algorithm (shortest path) and Prim's algorithm (minimum spanning tree), where you always need to pick the "best" next step.
- **Data Compression:** Algorithms like Huffman coding.

4. Deque (Double-Ended Queue)

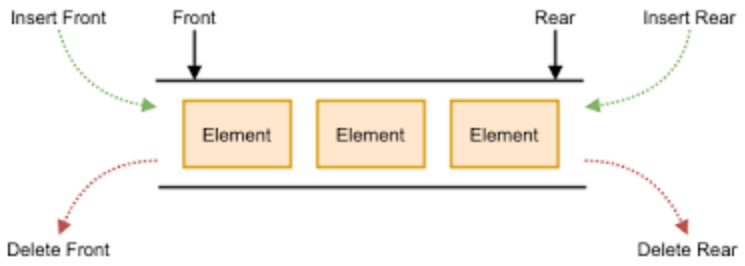
A **Deque** (pronounced "deck") stands for **"Double-Ended Queue."** It's the most flexible of our queue types, as it allows additions and removals from *both* ends.

Concept:

Imagine a line where people can enter or leave from *either* the front *or* the back. It's a combination of a Queue and a Stack!

- It has both a front and a rear.
- You can add to front or rear.
- You can remove from front or rear.
- You can peek at front or rear.

Visual:



Relationship to Stack and Queue:

- A **Stack** can be implemented using a Deque: Use addFirst as push, and removeFirst as pop.
- A **Queue** can be implemented using a Deque: Use addLast as enqueue, and

removeFirst as dequeue.

Implementation:

Deques are typically implemented using either a **doubly linked list** or a **resizable array** (like ArrayDeque in Java). Both allow for O(1) operations at both ends.

Basic Operations (all typically O(1) Time Complexity):

- **From Front:**
  - addFirst(item) / push(item): Adds to the front.
  - removeFirst() / pop(): Removes from the front.
  - peekFirst() / peek(): Returns item at the front.
- **From Rear:**
  - addLast(item) / offer(item): Adds to the rear.
  - removeLast(): Removes from the rear.
  - peekLast(): Returns item at the rear.

Advantages:

- **Ultimate Flexibility:** Supports both LIFO (stack-like) and FIFO (queue-like) behavior.
- **Efficient End Operations:** All operations at either end are O(1).
- **Dynamic Size:** Can grow and shrink as needed (if implemented with dynamic structures).

Disadvantages:

- **No Random Access:** Still cannot access elements by index efficiently.
- **More Complex Logic:** Managing two ends and four sets of operations can be slightly more complex to implement from scratch than a simple queue or stack.

When to Use:

- When you need a data structure that can act as **both a Stack and a Queue**.
- When you need efficient additions and removals from **both ends** of a list.
- For algorithms that require efficient management of elements at either extreme, like:
  - **Implementing a browser history** (where you might add/remove from both ends).
  - **Finding sliding window maximum/minimum**.
  - **Stealing work** in work-stealing schedulers.

Conclusion: Choose Your Queue Wisely!

You've now explored the diverse family of Queues!

- The **Simple/Linear Queue** is conceptually easy but can waste space in arrays.
- The **Circular Queue** brilliantly solves space inefficiency for array-backed queues by wrapping around.
- The **Priority Queue** ditches FIFO for importance, making it perfect for task scheduling based on urgency, typically powered by a Heap.
- The **Deque** is the versatile superhero, offering operations from both ends, effectively combining the powers of a Stack and a Queue.

Each type has its specific strengths and is designed to solve particular problems most efficiently. Understanding these differences is key to becoming a master of Data Structures and Algorithms!