Data Structures: Queues

Minseok Jeon
DGIST

November 2, 2025

Contents

Introduction to Queues

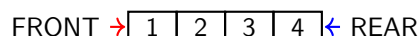
What is a Queue?

Definition

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle.

Key characteristics:

- Elements are inserted at the rear (enqueue)
- Elements are removed from the front (dequeue)
- Perfect for scheduling and buffering
- Models real-world waiting lines
- Used in OS scheduling and network packet handling



Queue Structure

Core Operations

Essential Queue Operations

Primary Operations

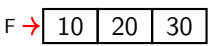
- **Enqueue**: Add element to rear
- **Dequeue**: Remove and return front element
- **Front/Peek**: View front element without removing
- **Empty**: Check if queue is empty
- **Size**: Get number of elements

C Example

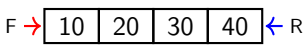
```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node* next;
} Node;

typedef struct Queue {
    Node *front, *rear;
} Queue;

void enqueue(Queue* q, int x) {
    Node* n = malloc(sizeof(Node));
    n->data = x; n->next = NULL;
    if (q->rear) q->rear->next = n;
    q->rear = n;
    if (!q->front) q->front = n;
}

int dequeue(Queue* q) {
    int x = q->front->data;
    Node* tmp = q->front;
    q->front = q->front->next;
    if (!q->front) q->rear = NULL;
    free(tmp); return x;
}
```

Queue Operations Visualization

Initial Queue

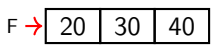


After Enqueue(40)



After Dequeue()

Returns: 10



Circular Queue

Circular Queue Concept

Why Circular Queue?

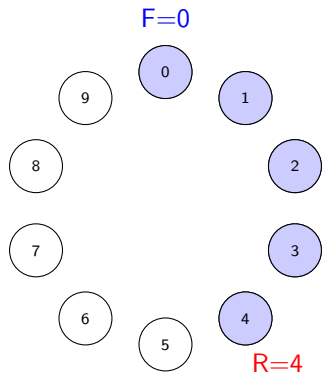
- Simple array queue wastes space
- Front pointer moves forward
- Space at beginning becomes unusable
- Solution: Wrap around using modulo

Wrap-around Formula

```
(index + 1) % capacity
```

Full/Empty Detection:

- Track size explicitly, OR
- Reserve one empty slot



Circular Buffer

Circular Queue Implementation

```python
class CircularQueue:
    def __init__(self, capacity):
        self.data = [None] * capacity
        self.capacity = capacity
        self.front = 0
        self.size = 0

    def enqueue(self, item):
        if self.size == self.capacity:
            raise Exception("Queue is full")
        rear = (self.front + self.size) % self.capacity
        self.data[rear] = item
        self.size += 1

    def dequeue(self):
        if self.size == 0:
            raise Exception("Queue is empty")
```

Deque (Double-Ended Queue)

Deque: Double-Ended Queue

Definition

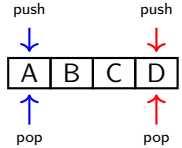
A **deque** (pronounced "deck") allows insertion and deletion at both ends.

Deque Operations

- `pushFront(x)`: Insert at front
- `pushBack(x)`: Insert at rear
- `popFront()`: Remove from front
- `popBack()`: Remove from rear

Use Cases

- Palindrome checking
- Sliding window algorithms
- Browser history (forward/back)



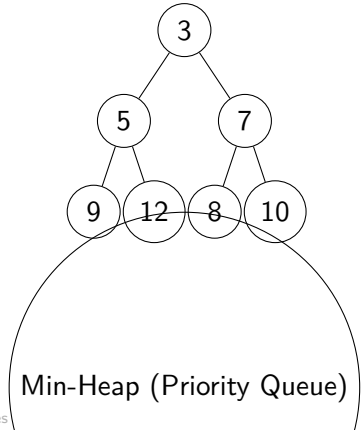
Both ends accessible

Priority Queue

Priority Queue

Definition

A **priority queue** is a queue where elements are served based on priority rather than insertion order.



Key Properties

- Each element has a priority
- Highest (or lowest) priority served first
- Typically implemented with binary heap
- Insert: O(log n)

Implementation Approaches

Array-based vs Linked List Implementation

Feature	Array-based	Linked List-based
Time Complexity	O(1) amortized	O(1) guaranteed
Space Efficiency	Better (contiguous)	More overhead (pointers)
Cache Performance	Excellent	Fair
Resize Cost	Occasional O(n)	Never
Capacity	Fixed (or resizable)	Dynamic

Array-based Queue

- Use circular buffer
- Track front and rear indices
- Better for bounded queues

Linked List Queue

- Maintain front and rear pointers
- No capacity concerns
- Better for unbounded queues

Linked List Queue Implementation

```
class Node:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

class QueueLL:
    def __init__(self):
        self.front = None
        self.rear = None
        self.size = 0

    def enqueue(self, x):
        new_node = Node(x)
        if self.rear:
            self.rear.next = new_node
        self.rear = new_node
        if not self.front:
```

Applications

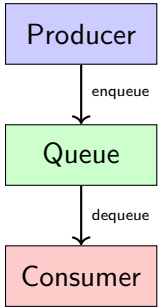
Producer-Consumer Pattern

Bounded Buffer Problem

- Producers generate data
- Consumers process data
- Queue acts as buffer
- Handles rate mismatch

Synchronization:

- If full $\rightarrow$ producer waits
- If empty $\rightarrow$ consumer waits
- Use locks/semaphores for thread safety



Examples

Print spooler, web server request handling

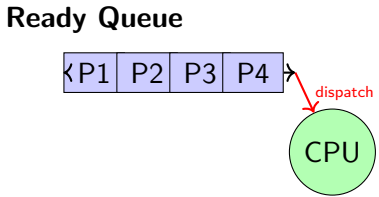
Queues in Operating Systems

CPU Scheduling

- Ready queue: processes ready to run
- Multiple priority queues
- Round-robin scheduling
- Multi-level feedback queue (MLFQ)

I/O Scheduling

- Disk request queue
- Network packet queue
- Print job queue



Queues in Networking

Network Routers and Switches

- Incoming packets queued before processing
- Multiple queues for Quality of Service (QoS)
- Different priority levels (voice, video, data)
- Scheduling algorithms: WFQ (Weighted Fair Queuing), Priority Queuing

Packet Queue Types

- High priority: VoIP, video conferencing
- Medium priority: streaming video
- Best effort: web browsing, email

Queue Management

- Drop-tail: drop when full
- Random Early Detection (RED)
- Token bucket rate limiting

Complexity Analysis

Time and Space Complexity

Structure	Enqueue	Dequeue	Peek	Space
Queue (Array)	O(1) amortized	O(1)	O(1)	O(n)
Queue (Linked List)	O(1)	O(1)	O(1)	O(n) + pointers
Circular Queue	O(1)	O(1)	O(1)	O(capacity)
Deque	O(1)	O(1)	O(1)	O(n)
Priority Queue (Heap)	O(log n)	O(log n)	O(1)	O(n)

Special Cases

- For bounded integer priorities: Use bucket queues for O(1) operations
- For monotonic priorities: Consider monotonic queue optimization
- For small fixed priorities: Array of queues (one per priority level)

Queue vs Stack

Queue vs Stack Comparison

Property	Queue	Stack
Principle	FIFO (First In, First Out)	LIFO (Last In, First Out)
Insert	Rear (enqueue)	Top (push)
Remove	Front (dequeue)	Top (pop)
Real-world	Waiting line	Plate stack
Use case	Scheduling	Recursion, undo

Queue Applications

- Breadth-First Search (BFS)
- Task scheduling
- Buffering
- Order processing

Stack Applications

- Depth-First Search (DFS)
- Expression evaluation
- Function calls
- Undo operations

Summary

Key Takeaways

Queue Fundamentals

- FIFO data structure: First In, First Out
- Essential operations: enqueue (rear), dequeue (front)
- All basic operations are O(1) time complexity

Queue Variants

- **Circular Queue**: Efficient space usage with wrap-around
- **Deque**: Double-ended queue for flexible insertion/deletion
- **Priority Queue**: Element ordering based on priority (heap-based)

Important Applications

- Producer-consumer pattern and bounded buffers
- OS scheduling (CPU, I/O, process management)
- Network packet queuing and QoS

Thank You!

Questions?