1010101010
0101010101
1010101010
0101010101

Infographics

Visual Learning

Better understanding through visual content.

Simplified Concepts

Complex topics explained in simple and visual form.

Quick Revision

Easy to revise important points and diagrams.

Exam Ready

Helps in faster revision and better exam preparation.

Unit - II

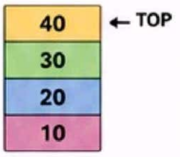
```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!";
    return 0;
}
```

STACKS

Definition and Concepts, Memory Representations

LIFO Principle

Stack follows LIFO (Last In First Out) principle.

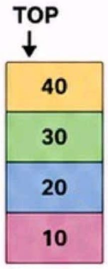


1. DEFINITION

A stack is a **linear** data structure that follows the **LIFO (Last In First Out)** principle. In a stack, insertion and deletion of elements can be performed from only **one end** called **TOP**.

2. CONCEPTS

- Stack is a collection of **similar data elements**.
- All operations (insertions and deletions) are performed at one end called TOP.
- The element at the top is the most recently inserted element.
- When a stack is full, it is called **OVERFLOW**.
- When a stack is empty, it is called **UNDERFLOW**.



Stack (Top at 40)

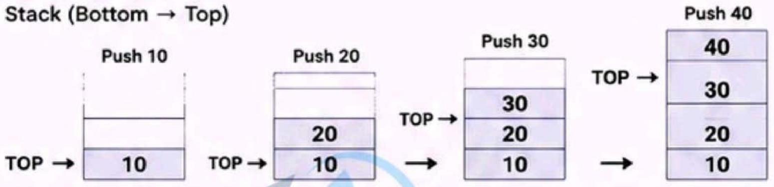
3. BASIC OPERATIONS

Operation	Description	Time Complexity
Push	Insert an element at the TOP	O(1)
Pop	Delete the element from the TOP	O(1)
Peek/Top	Return the top element	O(1)
isEmpty	Check if stack is empty	O(1)
isFull	Check if stack is full	O(1)

Example

Push 10, 20, 30, 40

Stack (Bottom → Top)

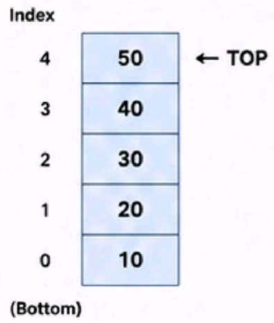


4. MEMORY REPRESENTATIONS OF STACK

4.1 STACK USING ARRAY

The stack is implemented using a one-dimensional array.

Representation



Example

After pushing 10, 20, 30, 40, 50

TOP = 4 (points to 50)

Elements are stored in contiguous memory.

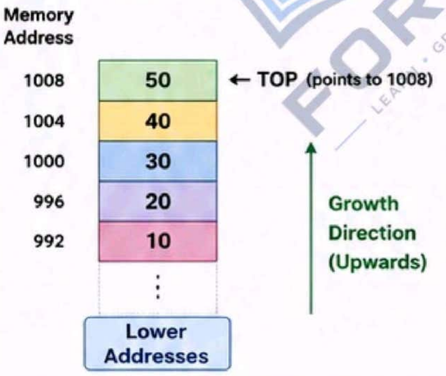
Notes

- MAX = 5 (size of stack)
- TOP = -1 → Stack is empty
- When TOP = MAX - 1
  → Stack is full (Overflow)
- New element is inserted at ++TOP
- Element is deleted from TOP--

4.2 STACK MEMORY LAYOUT

Memory addresses grow from lower to higher.

Stack grows upwards in memory.



Example

Base address (BA) = 992

Size of each element = 4 bytes

MAX = 5

Then,

Address of element at index i

= BA + (i × size)

For index 3 (40):

= 992 + (3 × 4)

= 1004

5. UNDERFLOW AND OVERFLOW

Underflow

Occurs when POP or PEEK is performed on an empty stack (TOP = -1).

Overflow

Occurs when PUSH is performed on a full stack (TOP = MAX - 1).

6. APPLICATIONS OF STACK

- ✓ Function calls (Call Stack)
- ✓ Recursion
- ✓ Expression evaluation and conversion
- ✓ Backtracking (e.g., maze, N-Queens)
- ✓ Syntax checking (e.g., parentheses matching)
- ✓ Undo operations in editors / browsers

7. SUMMARY TABLE

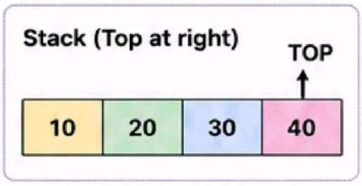
Feature	Stack
Type	Linear Data Structure
Principle	LIFO (Last In First Out)
Insertion	At TOP only (Push)
Deletion	From TOP only (Pop)
Access	Only TOP element accessible
Memory	Contiguous (Array) / Dynamic (Linked List)
Time Complexity	O(1) for all basic operations

KEY TAKEAWAY

Stacks follow LIFO principle. All operations are performed at TOP. Using arrays, stack elements are stored in contiguous memory locations and grow upwards.

STACK OPERATIONS

Traversing, Insertion, Deletion



Stack follows **LIFO** (Last In First Out) principle.
All operations are performed at one end called **TOP**.

1 TRAVERSING A STACK

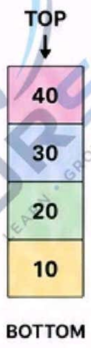
Traversing means accessing/visiting each element of the stack from **TOP** to **BOTTOM** without removing any element.

Example

Stack elements (Bottom → Top):

10	20	30	40

Indices: 0 1 3



ALGORITHM (TRAVERSAL)

1. If stack is empty, return.
2. Set a temporary variable temp = TOP.
3. While temp ≠ -1
   - `Print/visit STACK[temp]`
   - `temp = temp - 1`
4. Stop.

PSEUDOCODE

```
if TOP == -1
    print "Stack is Empty"
else
    temp = TOP
    while temp != -1
        print STACK[temp]
        temp = temp - 1
```

Illustration (Traversal)

Step	temp	Element Visited	Order
Initial	3	—	—
1	3	40	40
2	2	30	40, 30
3	1	20	40, 30, 20
4	0	10	40, 30, 20, 10
5	-1	Stop	Done

Traversal Order: 40 → 30 → 20 → 10
(From TOP to BOTTOM)

2 INSERTION (PUSH) IN STACK

Insertion operation is called **PUSH**.
A new element is inserted at **TOP**.

Example

Initial Stack (Bottom → Top):

10	20	30

TOP = 2

Push 40 into stack.

ALGORITHM (PUSH)

```
1. If TOP == MAX - 1
     → Stack Overflow
2. Else
     → TOP = TOP + 1
     → STACK[TOP] = newElement
```

PSEUDOCODE

```
if TOP == MAX - 1
    print "Stack Overflow"
else
    TOP = TOP + 1
    STACK[TOP] = newElement
```

Illustration (Push 40)

TOP is updated to 3 and 40 is inserted.
(Stack size increased by 1)

3 DELETION (POP) FROM STACK

Deletion operation is called **POP**.
The element at TOP is removed.

Example

Initial Stack (Bottom → Top):

10	20	30	40

TOP = 3

Pop the top element.

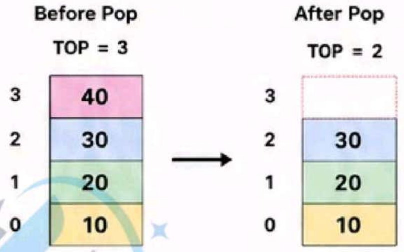
ALGORITHM (POP)

```
1. If TOP == -1
     → Stack Underflow
2. Else
     → element = STACK[TOP]
     → TOP = TOP - 1
     → Return element
```

PSEUDOCODE

```
if TOP == -1
    print "Stack Underflow"
else
    element = STACK[TOP]
    TOP = TOP - 1
    return element
```

Illustration (Pop)



Popped element = 40
TOP is updated to 2.
(Stack size decreased by 1)

SUMMARY TABLE

Operation	Purpose	Effect on TOP	Time Complexity
Traversal	Visit each element from TOP to BOTTOM	No Change	O(n)
Insertion (Push)	Add new element at TOP	TOP = TOP + 1	O(1)
Deletion (Pop)	Remove element from TOP	TOP = TOP - 1	O(1)

KEY TAKEAWAY

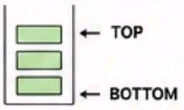
- ✓ Stack follows LIFO (Last In First Out) principle.
- ✓ All operations (except traversal) are done at TOP only.
- ✓ Overflow: Occurs when trying to push in a full stack.
- ✓ Underflow: Occurs when trying to pop from an empty stack.
- ✓ Traversal does not remove elements; it only visits them.

LEGEND

MAX = Maximum size of stack
TOP = Index of top element
-1 = Stack is empty

Push(x) : Insert x at TOP
Pop() : Remove and return top element

Peek/Top() : Return top element without removing it
isEmpty() : Check if stack is empty
isFull() : Check if stack is full

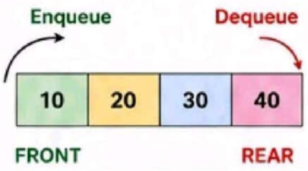


QUEUE

Definition and Concepts, Memory Representations

FIFO Principle

Queue follows FIFO (First In First Out) principle.



1. DEFINITION

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

Insertion of an element is done at one end called **REAR** and deletion is done from the other end called **FRONT**.

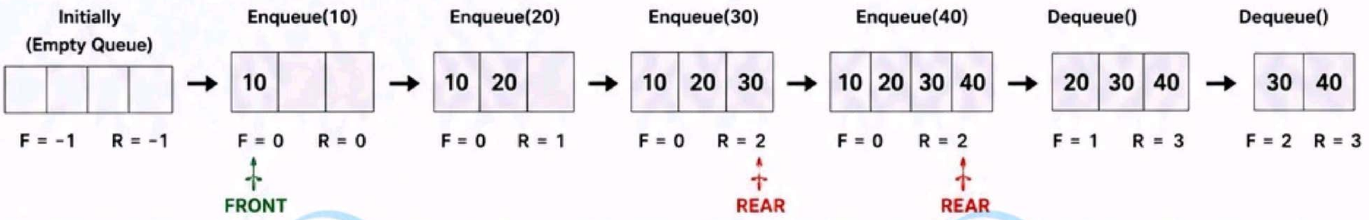
2. CONCEPTS

- Elements are added at **REAR** (Enqueue).
- Elements are removed from **FRONT** (Dequeue).
- The first element inserted is the first element removed.
- If queue is full, insertion causes **OVERFLOW**.
- If queue is empty, deletion causes **UNDERFLOW**.

3. BASIC OPERATIONS

Operation	Description	Time Complexity
Enqueue	Insert an element at REAR	O(1)
Dequeue	Delete an element from FRONT	O(1)
Front/Peek	Return the front element	O(1)
Rear	Return the rear element	O(1)
isEmpty	Check if queue is empty	O(1)
isFull	Check if queue is full	O(1)

Example Perform Enqueue(10), Enqueue(20), Enqueue(30), Enqueue(40), Dequeue(), Dequeue().



Initially (Empty Queue) → Enqueue(10) → Enqueue(20) → Enqueue(30) → Enqueue(40) → Dequeue() → Dequeue()

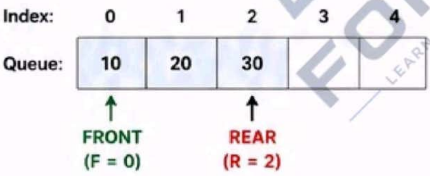
Initially (Empty Queue)	Enqueue(10)	Enqueue(20)	Enqueue(30)	Enqueue(40)	Dequeue()	Dequeue()
	10	10 20	10 20 30	10 20 30 40	20 30 40	30 40
F = -1 R = -1	F = 0 R = 0	F = 0 R = 1	F = 0 R = 2	F = 0 R = 2	F = 1 R = 3	F = 2 R = 3
	FRONT		REAR	REAR		

4. MEMORY REPRESENTATIONS OF QUEUE

4.1 QUEUE USING ARRAY (LINEAR QUEUE)

Queue is implemented using a one-dimensional array. Elements are stored in contiguous memory locations.

Representation



Index:	0	1	2	3	4
Queue:	10	20	30		
	FRONT (F = 0)		REAR (R = 2)		

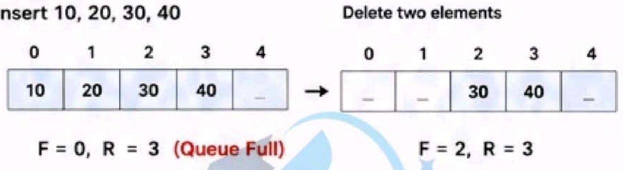
Notes

- MAX = 5 (size of array)
- F = Front index
- R = Rear index
- Initially: F = -1, R = -1
- Queue is Full when R = MAX – 1
- Queue is Empty when F = -1 or F > R

Example

Array size = 5
Insert 10, 20, 30, 40

Delete two elements



0	1	2	3	4
10	20	30	40	_

F = 0, R = 3 **(Queue Full)**

→

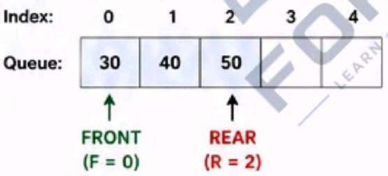
0	1	2	3	4
_	_	30	40	_

F = 2, R = 3

4.2 QUEUE USING CIRCULAR ARRAY (CIRCULAR QUEUE)

To efficiently utilize memory, the last position is connected to the first position to form a circle.

Representation

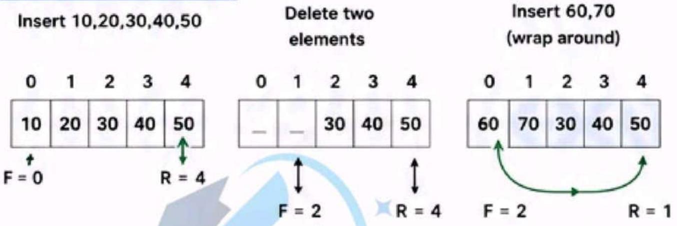


Index:	0	1	2	3	4
Queue:	30	40	50		
	FRONT (F = 0)		REAR (R = 2)		

Notes

- MAX = 5
- Next position after last index (4) is 0
- Condition for Full: (R + 1) % MAX = F
- Condition for Empty: F = -1

Example (Circular Behavior)



Insert 10,20,30,40,50

0	1	2	3	4
10	20	30	40	50

F = 0 R = 4

Delete two elements

0	1	2	3	4
_	_	30	40	50

F = 2 R = 4

Insert 60,70 (wrap around)

0	1	2	3	4
60	70	30	40	50

F = 2 R = 1

5. UNDERFLOW AND OVERFLOW

Underflow

Occurs when DEQUEUE or FRONT/PEEK is performed on an empty queue.
(No element to remove).

Overflow

Occurs when ENQUEUE is performed on a full queue.
(No space to insert).

6. APPLICATIONS OF QUEUE

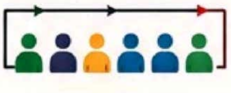
- ✓ CPU scheduling (Round Robin)
- ✓ Process management in OS
- ✓ Printer spooling
- ✓ I/O buffering
- ✓ Breadth First Search (BFS) in graphs
- ✓ Real-world ticket counters, customer lines, etc.

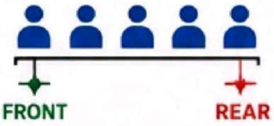
7. SUMMARY TABLE

Feature	Linear Queue	Circular Queue
Structure	Array (Linear)	Array (Circular)
Space Utilization	May waste space (after deletions)	Better space utilization
Full Condition	R = MAX – 1	(R + 1) % MAX = F
Empty Condition	F = -1 or F > R	F = -1
Complexity	O(1) for all basic operations	O(1) for all basic operations

KEY TAKEAWAY

- Queue follows FIFO (First In First Out) principle.
- Insertion is done at REAR and deletion is done from FRONT.
- Circular queue is preferred to avoid memory wastage.





QUEUE OPERATIONS:
TRAVERSING, INSERTION, DELETION & TYPES OF QUEUE

Queue follows **FIFO**
(First In First Out)

10	20	30	40

FRONT ⟷ REAR

A queue is a linear data structure that follows **FIFO** (First In First Out) principle.
Insertion is done at **REAR** and deletion is done from **FRONT**.

1 TRAVERSING (DISPLAY)

Traversing means visiting/printing all elements of the queue from **FRONT** to **REAR** without removing any element.

Example

Queue (F = 1, R = 4)

0	1	2	3	4
_	10	20	30	40

↑ FRONT ↑ REAR

Algorithm

```
1. If queue is empty, return.
2. Set i = FRONT.
3. Repeat until i = REAR
     (a) Visit/print Q[i]
     (b) i = i + 1
4. Stop.
```

Pseudocode

```
if (F == -1)
    print "Queue is Empty"
else
    for i = F to R
        print Q[i]
```

Illustration (Traversal)

Output:

10 → 20 → 30 → 40

Step	i	Element Visited	Order
Initial	1	–	–
1	1	10	10
2	2	20	10, 20
3	3	30	10, 20, 30
4	4	40	10, 20, 30, 40
End	–	Done	Done

Traversal Order: 10 → 20 → 30 → 40
(From **FRONT** to **REAR**)

2 INSERTION (ENQUEUE)

Insertion operation is called **ENQUEUE**.
A new element is inserted at **REAR**.

Example

Initial Queue (F = 1, R = 3)

0	1	2	3	4
_	10	20	30	_

↑ FRONT ↑ REAR

Enqueue(40)

After Insertion (F = 1, R = 4)

0	1	2	3	4
_	10	20	30	40

↑ FRONT ↑ REAR

Algorithm (Enqueue)

```
1. If queue is full, report OVERFLOW.
2. If queue is empty
     Set F = R = 0
3. Else
     Set R = R + 1
4. Insert new element at Q[R].
```

Pseudocode

```
if (queue is full)
    print "Queue Overflow"
else if (queue is empty)
    F = R = 0
else
    R = R + 1
  Q[R] = newElement
```

Illustration (Enqueue 40)

Before Enqueue

0	1	2	3	4
_	10	20	30	_

↑ F = 1 ↑ R = 3

↓

After Enqueue(40)

0	1	2	3	4
_	10	20	30	40

↑ F = 1 ↑ R = 4

REAR is updated and 40 is inserted.
(Queue size increases by 1)

3 DELETION (DEQUEUE)

Deletion operation is called **DEQUEUE**.
An element is removed from **FRONT**.

Example

Initial Queue (F = 1, R = 4)

0	1	2	3	4
_	10	20	30	40

↑ FRONT ↑ REAR

Dequeue()

After Deletion (F = 2, R = 4)

0	1	2	3	4
_	_	20	30	40

↑ FRONT ↑ REAR

Algorithm (Dequeue)

```
1. If queue is empty, report UNDERFLOW.
2. Store and remove element at FRONT.
3. If F = R (only one element was present)
     Set F = R = -1   (Queue becomes empty)
4. Else
     Set F = F + 1
```

Pseudocode

```
if (queue is empty)
    print "Queue Underflow"
else
  removedElement = Q[F]
  if (F == R)
     F = R = -1      // Queue becomes empty
  else
     F = F + 1
   return removedElement
```

Illustration (Dequeue)

Before Dequeue

0	1	2	3	4
_	10	20	30	40

↑ F = 1 ↑ R = 4

↓

After Dequeue()

0	1	2	3	4
_	_	20	30	40

↑ F = 2 ↑ R = 4

Element 10 is removed.
FRONT is updated. (Queue size decreases by 1)

4 TYPES OF QUEUE

1. LINEAR QUEUE

Insertion at REAR and deletion from FRONT in a linear manner.

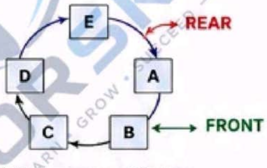
_	A	B	C	D	_

↑ FRONT ↑ REAR

- Simple implementation
- May cause wastage of space (insertion limited by size)

2. CIRCULAR QUEUE

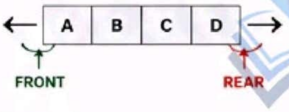
Last position is connected to the first position to form a circle.



- Efficient memory utilization
- No space wastage

3. DEQUE (Double Ended Queue)

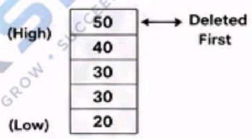
Insertion and deletion are allowed at both FRONT and REAR.



- More flexible
- Can act as queue or stack

4. PRIORITY QUEUE

Each element has a priority.
Element with highest (or lowest) priority is removed first.



- Based on priority
- Used in scheduling, etc.

SUMMARY TABLE (OPERATIONS)

Operation	Purpose	Condition Checked	Time Complexity
Traversal (Display)	Visit all elements from FRONT to REAR	Check if Empty	O(n)
Insertion (Enqueue)	Add element at REAR	Check if Full	O(1)
Deletion (Dequeue)	Remove element from FRONT	Check if Empty	O(1)

CONDITIONS

- Queue is Empty → F = -1 and R = -1
- Queue is Full (Linear Queue)
  → R = MAX - 1
- Queue is Full (Circular Queue)
  → (R + 1) % MAX = F
- Queue is Empty (Circular Queue)
  → F = -1

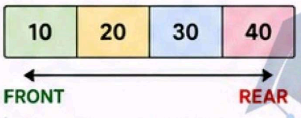
KEY TAKEAWAY

- ✓ Queue follows FIFO principle.
- ✓ Insertion is done at REAR.
- ✓ Deletion is done from FRONT.
- ✓ Choose the right type of queue as per the application.

TYPES OF QUEUE

Queue is a linear data structure that follows FIFO (First In First Out) principle.

FIFO Principle



Insertion at REAR
Deletion from FRONT

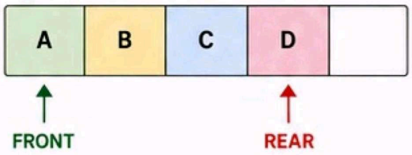
Common Terms

FRONT : Position of first element
REAR : Position of last element
ENQUEUE : Insert at REAR
DEQUEUE : Delete from FRONT

1. LINEAR QUEUE

Elements are inserted at REAR and deleted from FRONT in a linear manner.

Diagram



Features

- Simple structure
- Easy to implement
- May cause wastage of space (empty positions cannot be used)

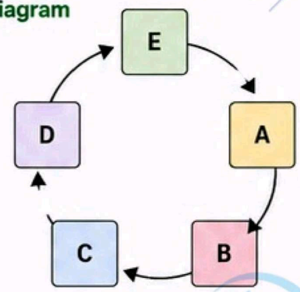
Applications

- CPU scheduling
- Print spooling
- Basic buffering

2. CIRCULAR QUEUE

Last position is connected to the first position to form a circle.

Diagram



Features

- Efficient memory utilization
- No space wastage
- Used modulo arithmetic for indexing

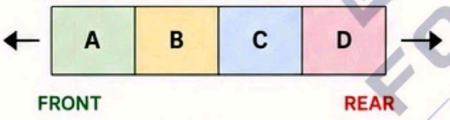
Applications

- Real-time systems
- I/O buffering
- Circular scheduling

3. DEQUE (DOUBLE ENDED QUEUE)

Insertion and deletion are allowed at both FRONT and REAR.

Diagram



Operations

- Insert at FRONT
- Delete from FRONT
- Insert at REAR
- Delete from REAR

Features

- More flexible than simple queue
- Can be implemented using array or linked list

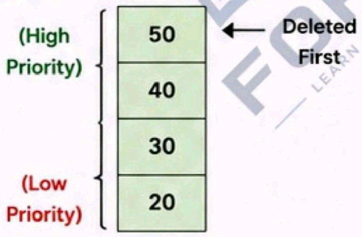
Applications

- Palindrome checking
- Undo/Redo operations
- Sliding window problems

4. PRIORITY QUEUE

Each element has a priority. Element with highest (or lowest) priority is removed first.

Diagram



Features

- Elements are served based on priority
- Higher (or lower) priority element is dequeued first
- Can have same priority for multiple elements

Applications

- Job scheduling
- Dijkstra's algorithm
- Emergency services
- Process management

COMPARISON OF QUEUE TYPES

Type	Insertion	Deletion	Ends Used	Memory Utilization	Best Used For
Linear Queue	At REAR	From FRONT	One (Linear)	May waste space	Simple applications
Circular Queue	At REAR (modulo)	From FRONT (modulo)	One (Circular)	No space waste	When memory is limited
Deque	At both ends	From both ends	Two (Both ends)	No space waste (if circular)	Undo/Redo, Palindrome, Sliding window
Priority Queue	By Priority	By Priority	One	Depends on implementation	Scheduling, Path finding, Priority based tasks

KEY TAKEAWAY

- ✓ All queues follow FIFO principle.
- ✓ Linear Queue is simplest but may waste space.
- ✓ Circular Queue is efficient in memory utilization.
- ✓ Deque allows operations at both ends.
- ✓ Priority Queue serves by priority, not by arrival order.