Data Structures

STACKS & QUEUES (CHAPTER 3)

Array

Array								
Integer Array	64	34	25	12	22	11	90	8

Restriction

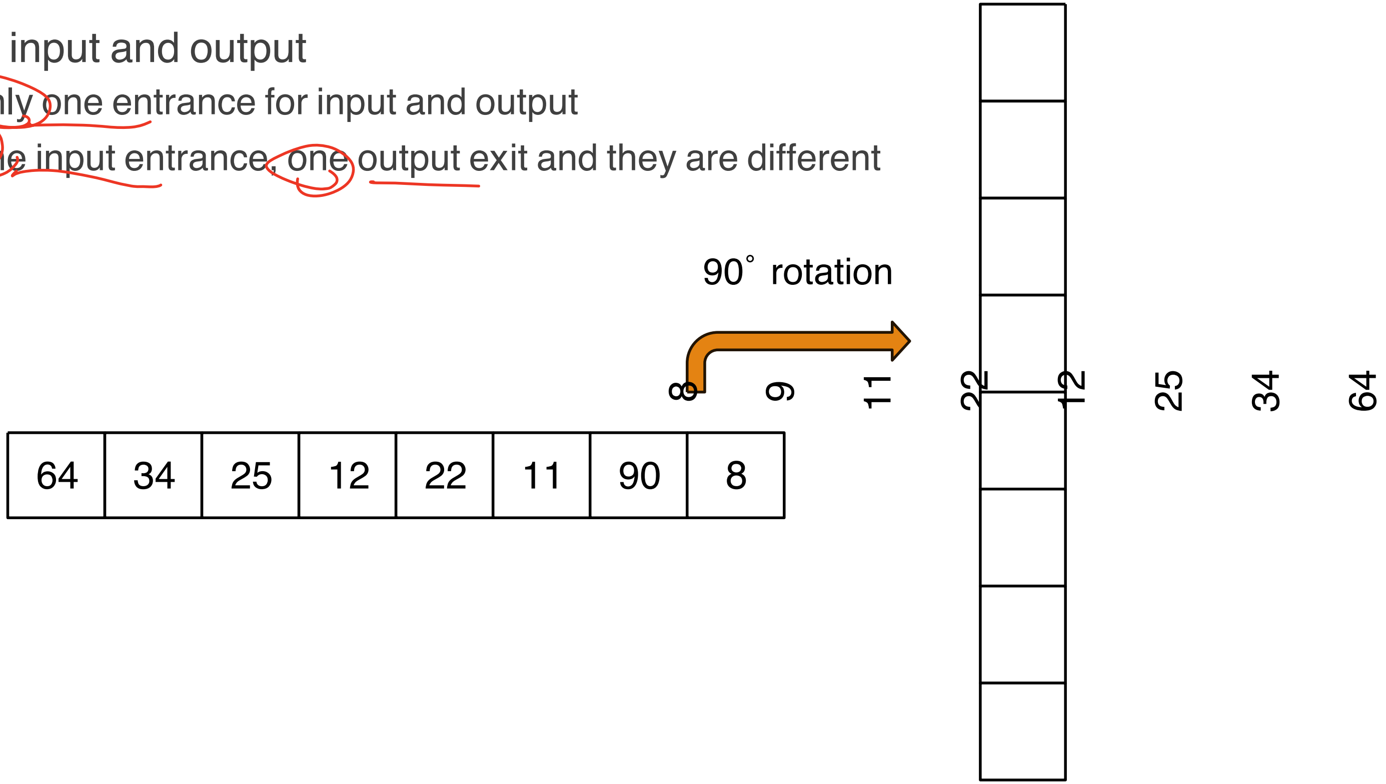
The direction of input and output

stack
- Scenario 1: Only one entrance for input and output

Queue
- Scenario 2: One input entrance, one output exit and they are different

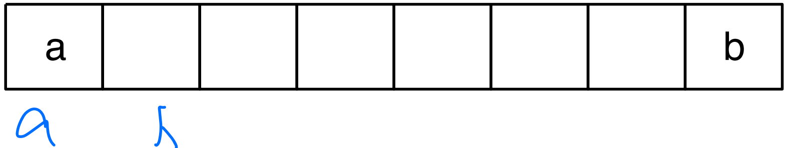
LIFO

LILO



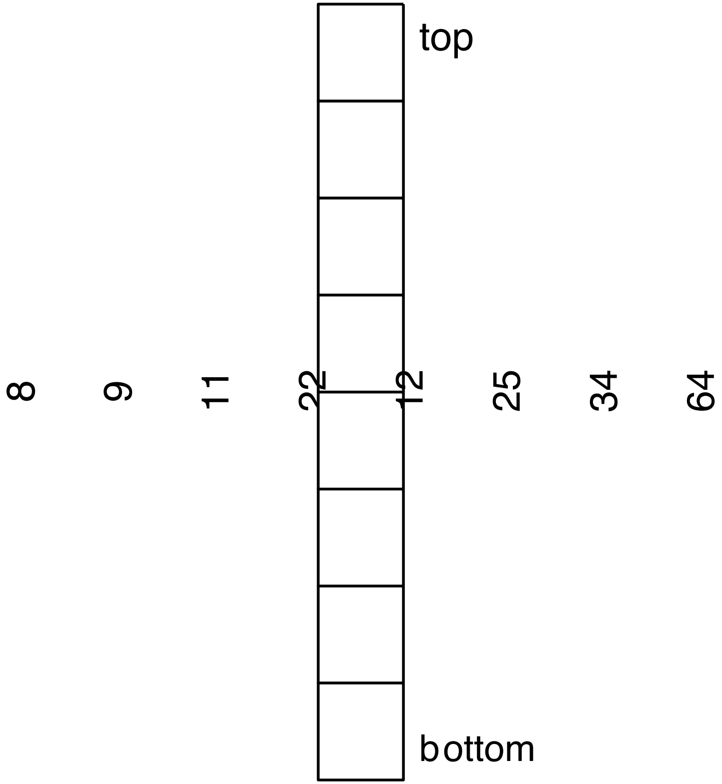
Scenario 1: Only Entrance for Input and Output

Integers: 64, 34, 25, 12, 22, 11, 90, 8



Bottom / Top:

- a or b ?
- Why?



Scenario 1: Only Entrance for Input and Output

Integers: 64, 34, 25, 12, 22, 11, 90, 8

If we choose **"a"** as the bottom of the stack

- Array
  - max_size = 8
  - top = 0

input	bottom 0	1	2	3	4	5	6	top 7
64	64							
34	64	34						
25	64	34	25					
12	64	34	25	12				
22	64	34	25	12	22			
11	64	34	25	12	22	11		
90	64	34	25	12	22	11	90	max

Scenario 1: Only Entrance for Input and Output

Integers: 64, 34, 25, 12, 22, 11, 90, 8

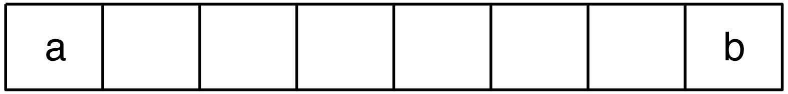
If we choose **“b”** as the bottom of the stack

- Array
  - max_size = 8
  - top = max_size - 1

	top							bottom
	0	1	2	3	4	5	6	7
input								
64								64
34							34	64
25						25	34	64
12					12	25	34	64
22				22	12	25	34	64
11			11	22	12	25	34	64
90		90	11	22	12	25	34	64

Scenario 2: One Entrance, One Exit

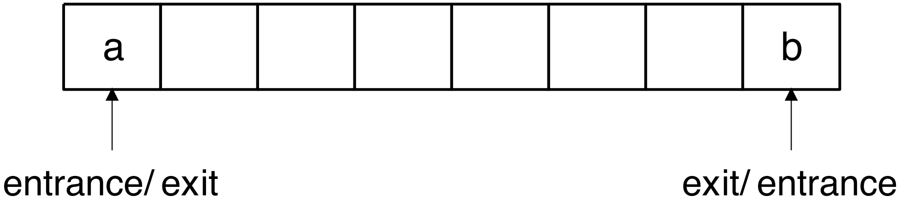
Integers: 64, 34, 25, 12, 22, 11, 90, 8



入口出口必須不同

Entrance:

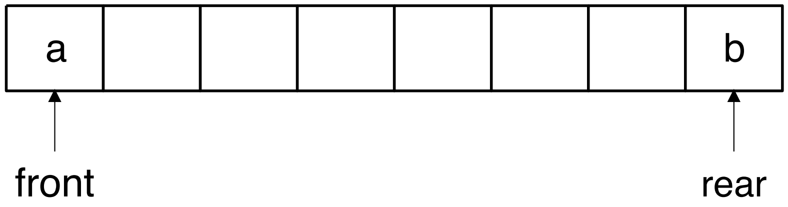
- a or b ?
- Why?



Scenario 2: One Entrance, One Exit

Integers: 64, 34, 25, 12, 22, 11, 90, 8

If we choose **"a"** as the front, **"b"** will be the rear.



Scenario 2: One Entrance, One Exit

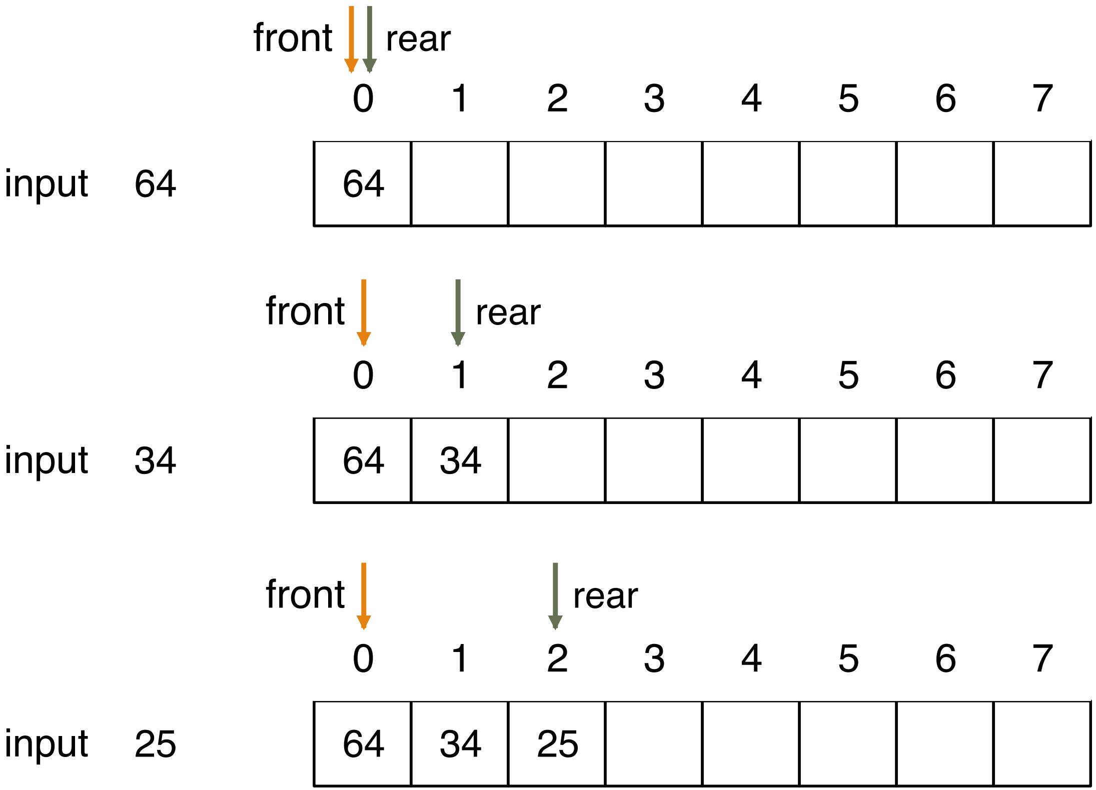
Integers: 64, 34, 25, 12, 22, 11, 90, 8

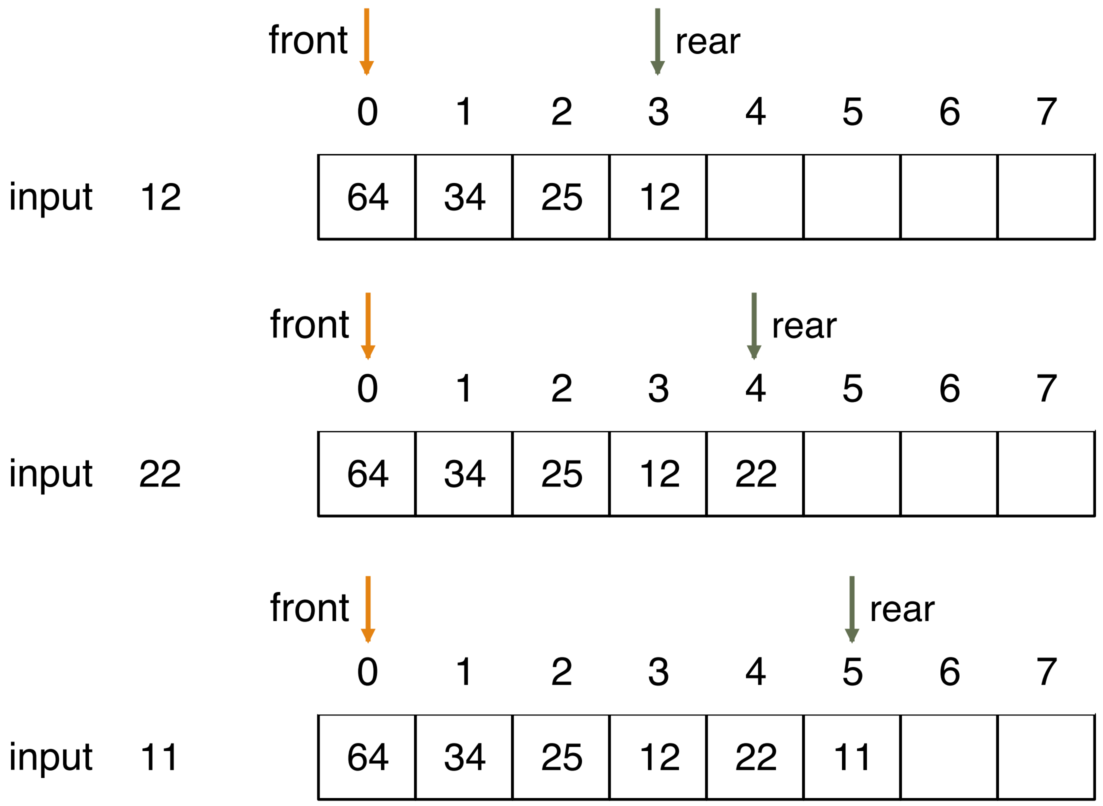
If we choose **"a"** as front, **"b"** will be the rear.

- Array
  - max_size = 8
  - front = 0
  - rear = 0

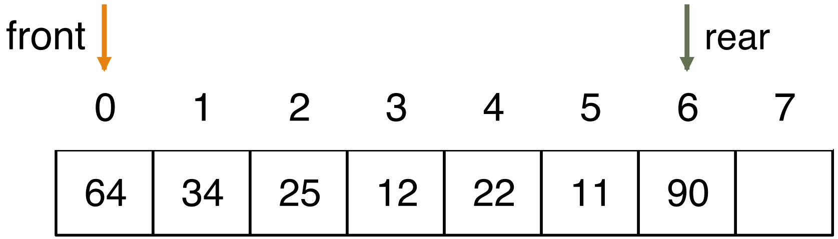
front rear

input	0	1	2	3	4	5	6	7
64	64							
34	64	34						
25	64	34	25					
12	64	34	25	12				
22	64	34	25	12	22			
11	64	34	25	12	22	11		
90	64	34	25	12	22	11	90	


front
rear
0 1 2 3 4 5 6 7
input 64
64
front
rear
0 1 2 3 4 5 6 7
input 34
64 34
front
rear
0 1 2 3 4 5 6 7
input 25
64 34 25


front
rear
0 1 2 3 4 5 6 7
input 12
64 34 25 12
front
rear
0 1 2 3 4 5 6 7
input 22
64 34 25 12 22
front
rear
0 1 2 3 4 5 6 7
input 11
64 34 25 12 22 11

input 90



Stack

A linear data structure that follows the Last-In-First-Out (LIFO) principle.

Elements are added and removed from the same end, called the “top.”

Think of it like a stack of plates - you can only add or remove plates from the top.

Common operations include push (add element) and pop (remove element).

Tower of Hanoi (河內塔)

Image credit: Amazon

ADT: Stack

ADT *Stack* is

objects:

A finite ordered list with zero or more elements.

functions:

for all *stack* ∈ *Stack*, *item* ∈ *element, maxStackSize* ∈ positive integer

Stack CreateS(*maxStackSize*) ::= create an empty stack whose maximum size is *maxStackSize*

Boolean IsFull(*stack*, *maxStackSize*) ::= if (number of elements in *stack* == *maxStackSize*)
return TRUE
else return FALSE

Boolean IsEmpty(*stack*) ::= **if** (stack == CreateS(maxStackSize))
return TRUE
else return FALSE

Stack Push(*stack*) ::= **if** (IsFull(*stack*)) *stackFull*
else insert *item* into the top of stack and **return**

Stack Pop(*stack*) ::= **if** (IsEmpty(*stack*)) **return**
else remove and return the *item* at the top of the stack.

end *Stack*

Queue

A linear data structure that follows the First-In-First-Out (FIFO) principle.

Elements are added at one end (rear) and removed from the other end (front).

It's like a line of people waiting - the first person in line is the first to be served.

Main operations are enqueue (add element) and dequeue (remove element).

Checkout Queue

Image credit: https://codefinity.com/courses/v2/212d3d3e-af15-4df9-bb13-5cbbb8114954/3a2558c0-2edb-4b98-a8d6-88b105cbcdda/126bf1eb-5380-4517-b488-f6f209d56675

ADT: Queue

```
ADT Queue is
  objects:
        A finite ordered list with zero or more elements.
  functions:
        for all queue ∈ Queue, item ∈ element, maxQueueSize ∈ positive integer
        Queue CreateQ(j, list)            ::=   create an empty queue whose maximum size is maxQueueSize
        Boolean IsFullQ(queue, maxQueueSize)    ::=   if (number of elements in queue == maxQueueSize)
                                                return TRUE
                                                else return FALSE
        Boolean IsEmptyQ(queue)                 ::=   if (stack == CreateS(maxQueueSize))
                                                return TRUE
                                                else return FALSE
        Queue AddQ(queue)                 ::=   if (IsFull(queue)) stackFull
                                                      else insert item at rear of queue and return
        Queue DeleteQ(queue)              ::=   if (IsEmpty(queue)) return
                                                 else remove and return the item at front of queue.
end Queue
```

Think: Stack vs. Queue

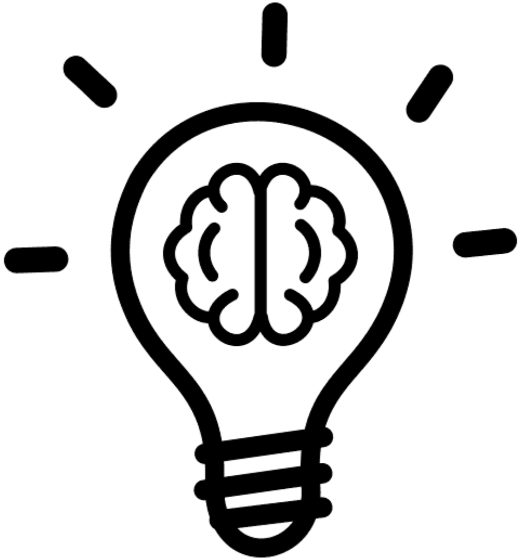


Image credit: https://uxwing.com/idea-icon/

Using Array to Implement Stack/Queue

Stack (Array-based)

Queue (Array-based)

Stack (Array-based)

Variables required

- stack[MAX_SIZE] → actual array storage
- top → integer index of the current top element (initially –1)
- MAX_SIZE → maximum capacity

Operations

- Push: check top < MAX_SIZE–1, then stack[++top] = value
- Pop: check top >= 0, then value = stack[top––]
- Peek: return stack[top]

Extra burden

- Must check overflow (top == MAX_SIZE–1) and underflow (top == –1)
- Resizing requires allocating a bigger array and copying data (if dynamic arrays used)

Queue (Array-based)

Variables required

- queue[MAX_SIZE] → actual array storage
- front → index of the first element
- rear → index of the last element
- MAX_SIZE → maximum capacity
- Sometimes count → number of elements (optional, but simplifies full/empty check)

Operations

- Enqueue: rear = (rear + 1) % MAX_SIZE; queue[rear] = value;
- Dequeue: front = (front + 1) % MAX_SIZE; value = queue[front];

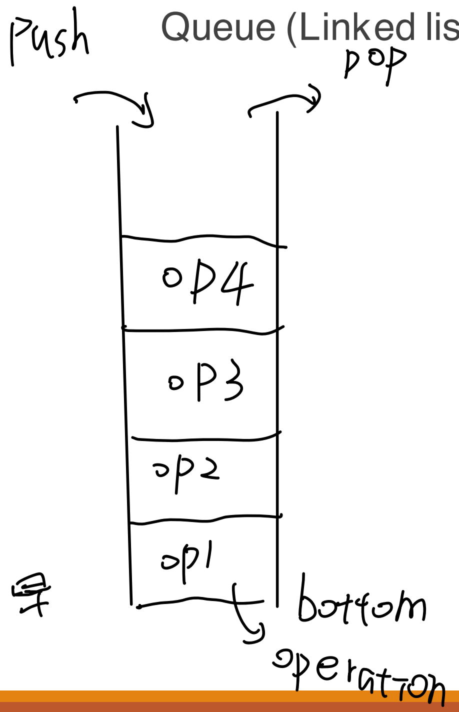
Extra burden

- Must manage circular buffer logic (wrap–around with modulo)
- Conditions for overflow ((rear+1) % MAX_SIZE == front) and underflow (front == rear)
- Need two pointers (front & rear), sometimes count

Using Linked List to Implement Stack/Queue

Stack (Linked list-based)

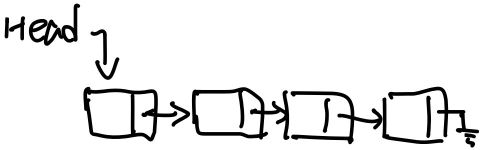
Queue (Linked list-based)



office

redo：直接pop拿出

undo：pop出的data放入 stack2，需要時再pop stack2



入口=出口 → Head (頻煩改head，易錯) & head不動

Tail → (traverse)

Stack (Linked List)

Variables required

- Node* top → pointer to the head node (stack top)

Operations

- Push: create new node, set newNode–>next = top, update top = newNode
- Pop: check if top != NULL, set top = top–>next

Extra burden

- Dynamic memory allocation (malloc / free) each operation
- Pointer management (avoiding leaks, dangling pointers)

Queue (Linked List)

Variables required

- Node* front → pointer to first node
- Node* rear → pointer to last node

Operations

- Enqueue: create new node, set rear–>next = newNode, update rear = newNode
- Dequeue: check if front != NULL, set front = front–>next

Extra burden

- Need two pointers (front & rear) for O(1) operations
- Must handle special case when queue becomes empty (set rear = NULL)
- Memory allocation overhead for each node

Comparison of Stacks and Queues (Array vs. Linked List Implementation)

Feature	Stack (LIFO)	Queue (FIFO)
Access Pattern	Last-In, First-Out (push/pop at the same end)	First-In, First-Out (enqueue at one end, dequeue at the other)
Direction of I/O	Both operations happen at the **top**	Input at **rear**, output at **front**
Array Implementation	- Easy to implement with a fixed-size array is full ? – `push`: add at end (increment top) – `pop`: remove from end (decrement top) - Limitation: **overflow** if capacity exceeded	- Usually implemented as **circular array** to reuse space – `enqueue`: add at rear – `dequeue`: remove at front - Requires managing two pointers (front, rear)
Linked List Implementation	- Each node points to next – `push`: insert at head – `pop`: remove from head - No fixed size (dynamic memory)	- Each node points to next – `enqueue`: insert at rear – `dequeue`: remove from front - Needs pointers for both front and rear
Memory Management	Array may waste unused slots if not full	Linked list uses extra memory for pointers
Performance	- Array: O(1) push/pop (if no resize) - Linked list: O(1) push/pop at head	- Array: O(1) enqueue/dequeue with circular buffer - Linked list: O(1) enqueue (at tail) and dequeue (at head)
Overflow Handling	Fixed-size array may overflow Dynamic array requires resizing	Circular array may overflow Linked list has no overflow unless memory exhausted
Use Cases	Undo functionality, function calls, expression evaluation	Task scheduling, resource sharing, buffering (e.g., I/O queues, printer queues)

Comparison of Stacks and Queues (Array vs. Linked List Implementation)

Structure	Array Implementation	Linked List Implementation
Stack	top, MAX_SIZE, array storage	top pointer
Queue	front, rear, MAX_SIZE, array storage, sometimes count	front and rear pointers
Memory Management	Pre-allocated, may waste unused space; resizing cost	Dynamic allocation per node, pointer overhead
Overflow/Underflow	Must check indices; circular logic for queue	Only “overflow” if heap memory is exhausted
Performance	O(1) push/ pop/ enqueue/ dequeue (except resizing)	O(1) push/ pop/ enqueue/ dequeue
Extra Burden	Track indices (front, rear, top), modulo arithmetic for queue	Manage pointers carefully, handle empty cases, memory free