CHAPTER 2:
STACKS AND QUEUES

Content

- Introduction, ADT of Stack
- Array Implementation of Stack
- Operations on Stack
- Applications of Stack-Well formedness of Parenthesis
- Infix to Postfix Conversion
- Postfix Evaluation
- Recursion
- Introduction, ADT of Queue
- Operations on Queue

Introduction to Stack

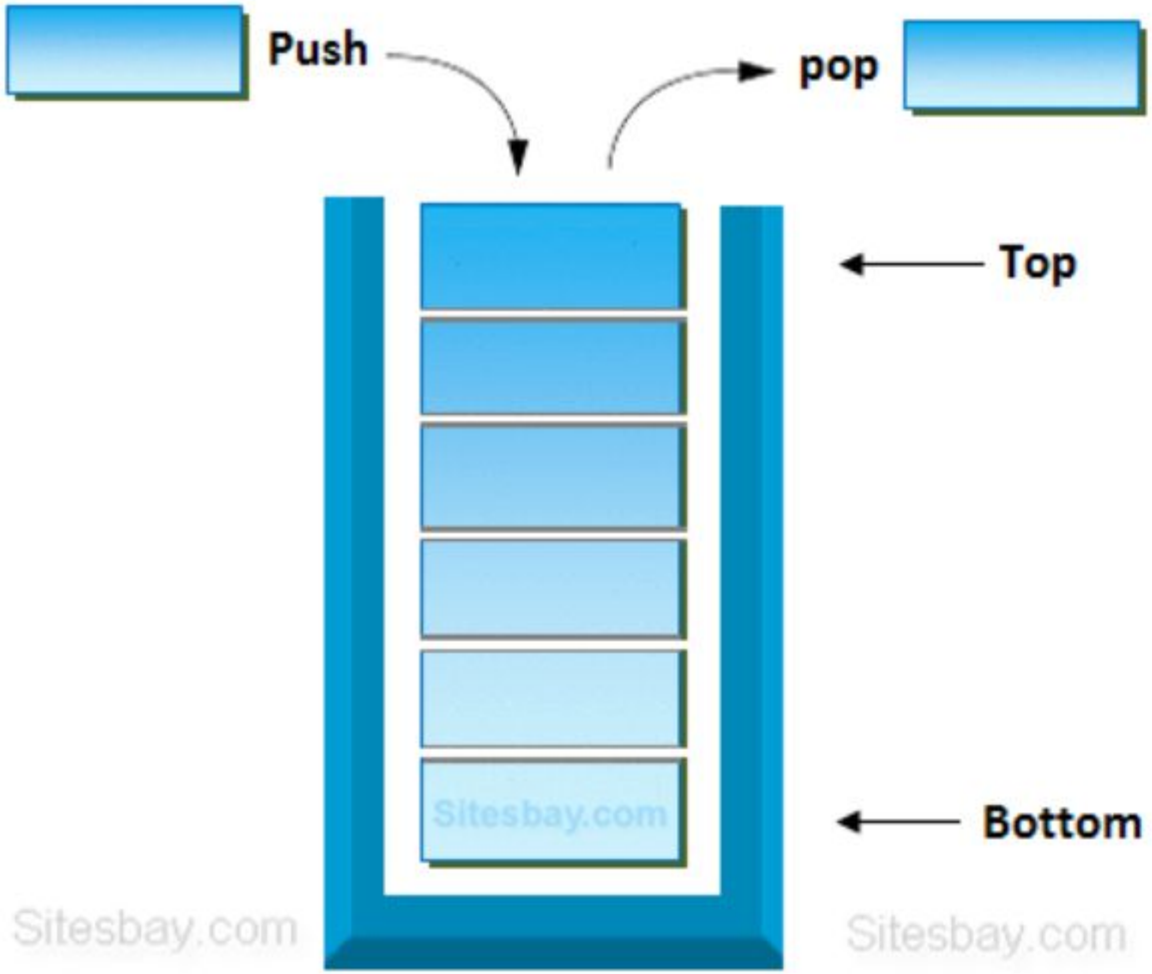
Stack – Data Structure Explanation

A **Stack** is a **linear data structure** that follows the **LIFO** principle – **Last In, First Out**
This means that the **last element** inserted into the stack is the **first one** to be remo

Real-Life Analogy

Think of a stack of plates:

- You add (push) new plates **on top**.
- You remove (pop) plates **from the top**.



TRUST

Stack ADT

Abstract Data Type (ADT) of Stack

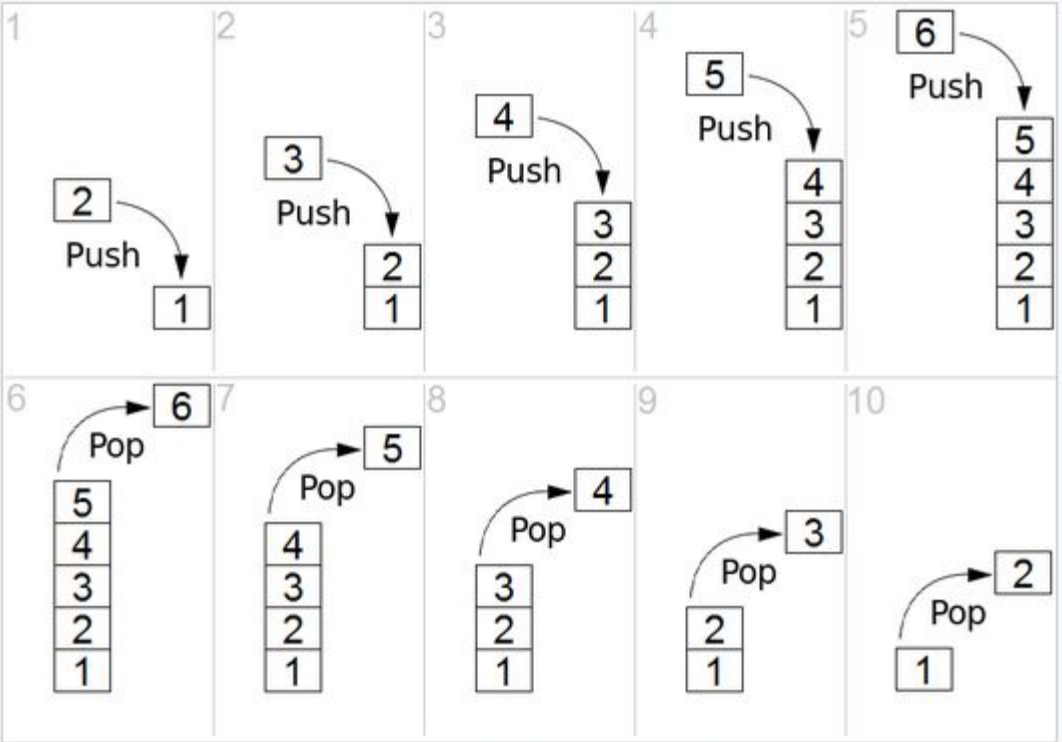
An **Abstract Data Type (ADT)** defines the *logical behavior* of a data structure — what it should do, without specifying how it should be implemented.

Stack ADT – Specification

A **Stack ADT** is a collection of elements with **LIFO (Last In, First Out)** behavior.

Operations Defined in Stack ADT

Operation	Description
`Push(x)`	Insert element `x` on top of the stack
`Pop()`	Remove and return the top element of the stack
`Peek()` or `Top()`	Return the top element **without removing** it
`IsEmpty()`	Check if the stack has no elements
`IsFull()`	(Optional, in case of fixed size) Check if the stack is full



Simple representation of a stack runtime with *push* and *pop* operations.

Stack ADT

Abstract Data Type (ADT) of Stack

An **Abstract Data Type (ADT)** defines the *logical behavior* of a data structure — what it should do, without specifying how it should be implemented.

Stack ADT – Specification

A **Stack ADT** is a collection of elements with **LIFO (Last In, First Out)** behavior.

Operations Defined in Stack ADT

Operation	Description
`Push(x)`	Insert element `x` on top of the stack
`Pop()`	Remove and return the top element of the stack
`Peek()` or `Top()`	Return the top element **without removing** it
`IsEmpty()`	Check if the stack has no elements
`IsFull()`	(Optional, in case of fixed size) Check if the stack is full

Stack PUSH OPERATION

✅ Push Operation in Stack

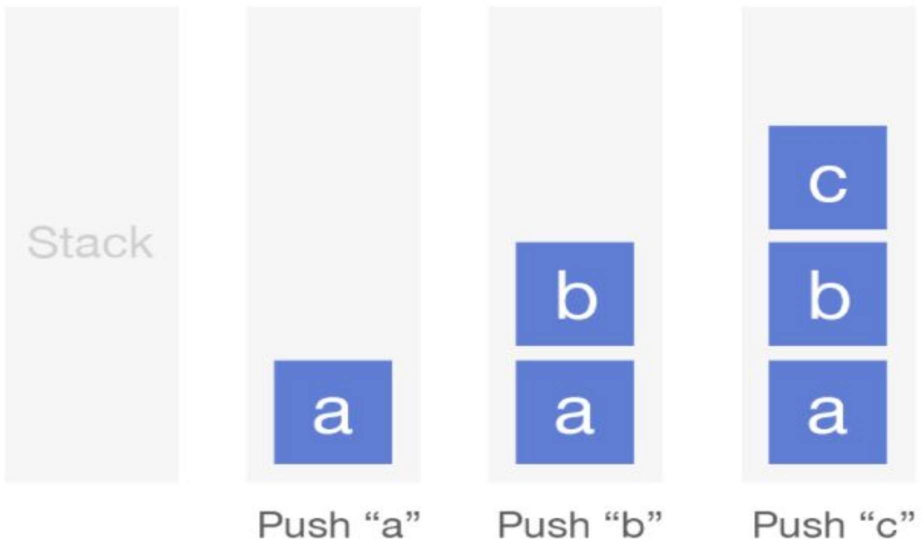
The **Push operation** is used to **insert an element** onto the **top** of the stack.

```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

📌 Key Concept:

Since a **stack** follows the **LIFO (Last In, First Out)** principle:

- The **new element** is always added **at the top**.
- The **top pointer/index** is updated accordingly.



🔧 Steps of Push Operation

Let's assume the stack is implemented using an array and a `top` variable tracks the index of the topmost element.

1. **Check for Overflow** (if the stack is full)
2. **Increment the top** pointer
3. **Insert the new element** at the new top position

⚠️ Overflow Condition

If you try to push when the stack is already full (in array-based implementation), you'll get a **stack overflow** error.

Stack POP OPERATION

✅ Pop Operation in Stack

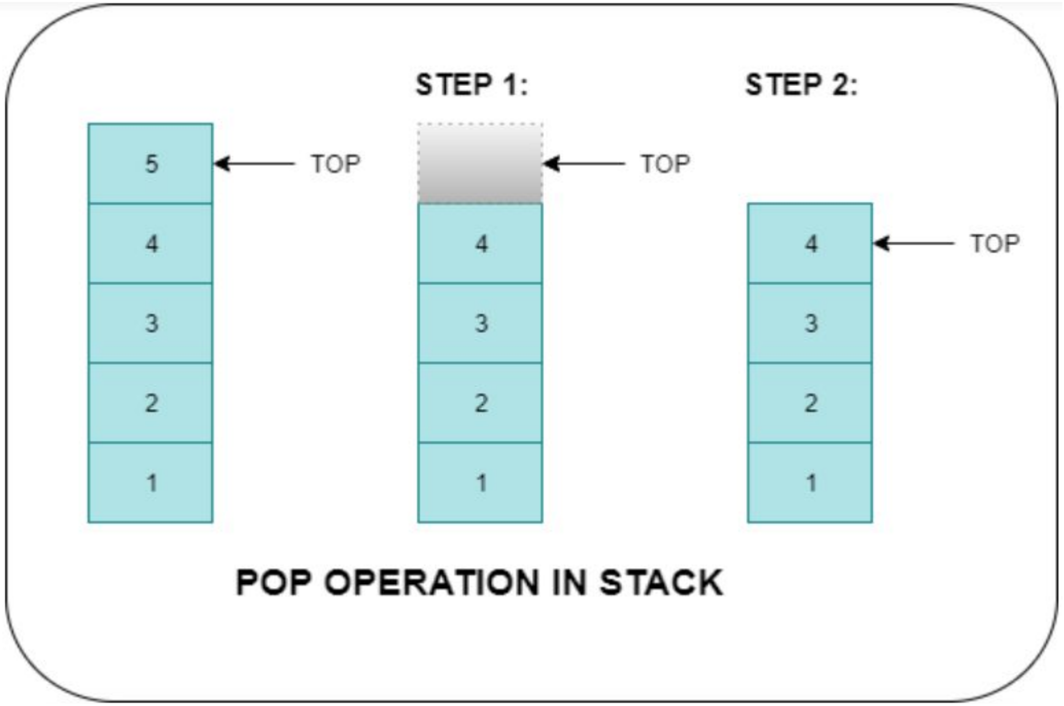
The **Pop operation** is used to **remove** and **return** the **top element** of the stack.

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

📌 Key Concept:

Since a stack works on the **LIFO (Last In, First Out)** principle:

- The **last pushed** (topmost) element is the **first to be popped**.



🔧 Steps of Pop Operation

Assuming the stack is implemented using an array and a `top` variable:

1. **Check for Underflow** (if the stack is empty)
2. **Access the element** at `top`
3. **Decrease the** `top` **pointer** by 1
4. **Return the removed element**

⚠️ Underflow Condition

If you try to pop from an empty stack (i.e., `top == -1`), you'll get a **stack underflow** error.

Stack PEEK OPERATION

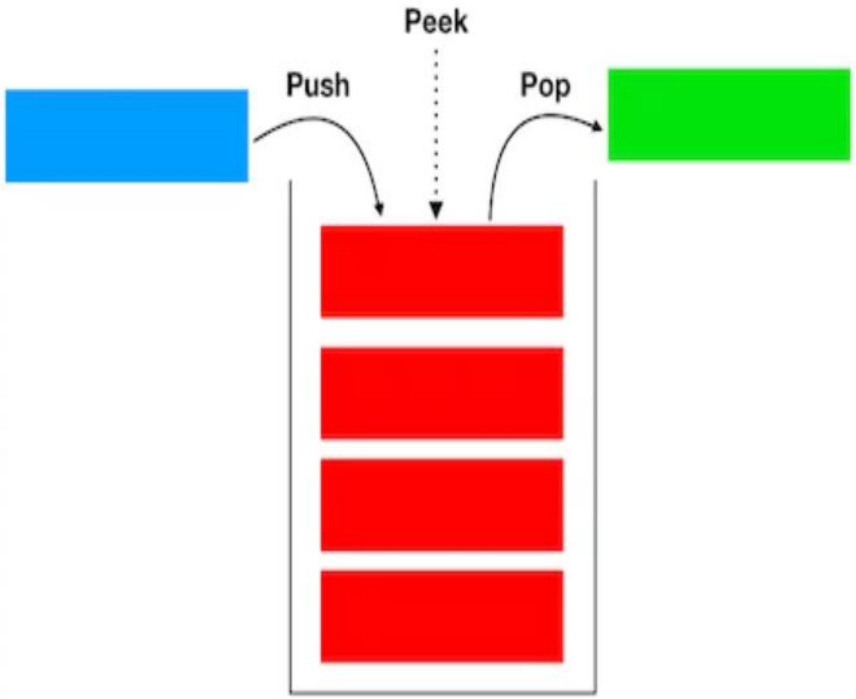
✅ Peek Operation in Stack

The **Peek** operation (also called `Top`) is used to **view the topmost element** of the stack **without removing it**.

```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

📌 Key Concept:

- Unlike `pop()`, **peek does not remove** the top element.
- It simply returns the value of the element at the top of the stack.



🔧 Steps of Peek Operation

1. **Check for Underflow** (i.e., stack is empty)
2. **Return the element** at index `top`

⚠️ Underflow Condition

If the stack is empty (`top == -1`), peek operation is **not valid** and should report **"Stack is empty"**.

Stack ISEMPTY() OPERATION

✅ IsEmpty Operation in Stack

The **IsEmpty** operation is used to **check whether the stack has any elements or not.**

📌 Key Concept:

- It returns `True` if the stack is empty.
- It returns `False` if there is **at least one element** in the stack.

📘 Pseudocode

plaintext

```
function isEmpty()
    if top == -1 then
        return true
    else
        return false
```

Somaiya TRUST

Stack ISFULL() OPERATION

✅ IsFull Operation in Stack

The **IsFull** operation checks whether the stack has reached its **maximum capacity** — i.e., no more elements can be added.

📌 Key Concept (Array-Based Stack):

- If the number of elements equals the **maximum size**, the stack is **full**.
- This check prevents **overflow** during a `push()` operation.

📘 Pseudocode

plaintext

```
function isFull()
    if top == MAX - 1 then
        return true
    else
        return false
```

ARRAY IMPLEMENTATION OF STACK

🔧 Array-Based Stack

In an **array implementation**, we use a fixed-size array and an integer variable `top` to track the index of the topmost element.

📌 Key Components

Component	Description
Array	Stores the stack elements
Top	Points to the top element (initially -1)
MAX	Maximum capacity of the stack (fixed size)

Somaiya TRUST

ARRAY IMPLEMENTATION OF STACK

✅ Operations

1. **Push(x)**
   - **Purpose:** Add element `x` to the top
   - **Steps:**
     - Check if `top == MAX - 1` → **Overflow**
     - Else increment `top`, insert `x` at `stack[top]`

2. **Pop()**
   - **Purpose:** Remove and return top element
   - **Steps:**
     - Check if `top == -1` → **Underflow**
     - Else return `stack[top]`, then decrement `top`

3. **Peek() / Top()**
   - **Purpose:** View the top element without removing it
   - **Steps:**
     - If `top == -1`, stack is empty
     - Else return `stack[top]`

4. **isEmpty()**
   - Return `True` if `top == -1`, else `False`

5. **isFull()**
   - Return `True` if `top == MAX - 1`, else `False`

ARRAY IMPLEMENTATION OF STACK

✅ C Program: Stack Using Array

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

#define MAX 100

int stack[MAX];
int top = -1;

// Function to check if stack is empty
int isEmpty() {
    return top == -1;
}

// Function to check if stack is full
int isFull() {
    return top == MAX - 1;
}
```

```c
// Push operation
void push(int x) {
    if (isFull()) {
        printf("Stack Overflow\n");
    } else {
        top++;
        stack[top] = x;
        printf("%d pushed to stack\n", x);
    }
}

// Pop operation
int pop() {
    if (isEmpty()) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        int popped = stack[top];
        top--;
        return popped;
    }
}
```

```c
// Peek operation
int peek() {
    if (isEmpty()) {
        printf("Stack is Empty\n");
        return -1;
    } else {
        return stack[top];
    }
}

// Display stack
void display() {
    if (isEmpty()) {
        printf("Stack is Empty\n");
    } else {
        printf("Stack elements: ");
        for (int i = 0; i <= top; i++) {
            printf("%d ", stack[i]);
        }
        printf("\n");
    }
}
```

TRUST

ARRAY IMPLEMENTATION OF STACK

```c
// Main function to demonstrate stack operations
int main() {
    push(10);
    push(20);
    push(30);
    display();

    printf("Top element: %d\n", peek());

    printf("%d popped from stack\n", pop());
    display();

    return 0;
}
```

📊 Stack Visualization

Before Any Push (Empty):

```makefile
Index:   0   1   2   3   4
Stack: [   ][   ][   ][   ][   ]
Top = -1
```

After Push(10), Push(20), Push(30):

```makefile
Index:   0    1    2    3    4
Stack: [10] [20] [30] [   ] [   ]
Top = 2
```

After Pop():

```makefile
Index:   0    1    2    3    4
Stack: [10] [20] [   ] [   ] [   ]
Top = 1
```

After Filling Stack (Full):

```makefile
Index:   0    1    2    3    4
Stack: [10] [20] [30] [40] [50]
Top = 4
```

Next Push → ❌ Overflow

APPLICATIONS OF STACK – INFIX TO POSTFIX

📌 What is Infix Expression?

- Operators are written between operands
- Example: `A + B * C`

📌 What is Postfix Expression (Reverse Polish Notation)?

- Operators are written after the operands
- Example: `A B C * +`

🔁 Why Convert Infix to Postfix?

- Infix expressions need **parentheses** and **operator precedence**
- Postfix eliminates the need for **parentheses**
- Easier to evaluate using a **stack**

T

APPLICATIONS OF STACK – INFIX TO POSTFIX

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

APPLICATIONS OF STACK – INFIX TO POSTFIX

Example 7.3 Convert the following infix expression into postfix expression using the algorithm given in Fig. 7.22.

(a) A – (B / C + (D % E * F) / G)* H

Solution

Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Example:

Convert:

`A + B * C`

Step-by-step:

Symbol	Stack	Postfix Output
A		A
+	+	A
B	+	A B
*	+ *	A B
C	+ *	A B C

End: Pop stack → `* +`

Postfix: `A B C * +`

APPLICATIONS OF STACK – POSTFIX EVALUATION

✅ What is Postfix Expression?

- A **Postfix expression** (also known as **Reverse Polish Notation**) is a mathematical expression where the **operator comes after the operands**.

Example:

- Infix: `A + B`
- Postfix: `A B +`

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

📘 Example:

Convert:

`A + B * C`

Step-by-step:

Symbol	Stack	Postfix Output
A		A
+	+	A
B	+	A B
*	+ *	A B
C	+ *	A B C

End: Pop stack → `* +`

👉 **Postfix:** `A B C * +`

APPLICATIONS OF STACK – POSTFIX EVALUATION

✅ What is Postfix Expression?

- A **Postfix expression** (also known as **Reverse Polish Notation**) is a mathematical expression where the operator comes after the operands.

  Example:

  - Infix: `A + B`
  - Postfix: `A B +`

```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

🧮 Example

Evaluate: `5 3 2 * +`

Step-by-step:

Symbol	Stack	Action
5	[5]	Push operand
3	[5, 3]	Push operand
2	[5, 3, 2]	Push operand
*	[5, 6]	3 * 2 = 6, push result
+	[11]	5 + 6 = 11, push result

✅ **Answer: 11**

APPLICATIONS OF STACK
WELL FORMED NESS OF PARENTHESIS

📌 What is Well-Formed Parentheses?

A string of parentheses (or brackets) is said to be **well-formed** if:

1. **Every opening bracket has a matching closing bracket**
2. **Brackets are closed in the correct order**

🔤 Examples

Expression	Well-formed?	Reason
`(a + b)`	✅ Yes	Balanced and correct order
`[(a + b) * c]`	✅ Yes	All brackets matched properly
`(a + b]`	❌ No	Mismatched closing bracket
`((a + b)`	❌ No	One opening bracket not closed
`([a + b)]`	❌ No	Wrong order of closing brackets

APPLICATIONS OF STACK
WELL FORMED NESS OF PARENTHESIS

🔗 Algorithm to Check Well-Formedness (Using Stack)

1. Initialize an **empty stack**
2. Traverse the string character by character
3. For each character:
   - If it is an **opening bracket**: Push to stack
   - If it is a **closing bracket**:
     - Check if the **stack is empty** → ❌ Not well-formed
     - Pop the top and check if it **matches** the closing bracket
     - If it doesn't match → ❌ Not well-formed
4. After traversal:
   - If stack is **empty** → ✅ Well-formed
   - If stack is **not empty** → ❌ Not well-formed

📘 Example

Input: `{ [ ( ) ] }`

Step	Character	Stack	Action
1	`{`	`{`	Push
2	`[`	`{ [`	Push
3	`(`	`{ [ (`	Push
4	`)`	`{ [`	Pop `(`, match
5	`]`	`{`	Pop `[`, match
6	`}`	empty	Pop `{`, match

✅ Well-formed!

Somaiya TRUST

RECURSION

✅ What is Recursion?

📌 Definition:

Recursion is a programming technique in which a **function calls itself** to solve a problem.

A recursive function **breaks down a problem** into smaller sub-problems, solving each by calling itself — until a **base condition** is met.

Somaiya TRUST

RECURSION

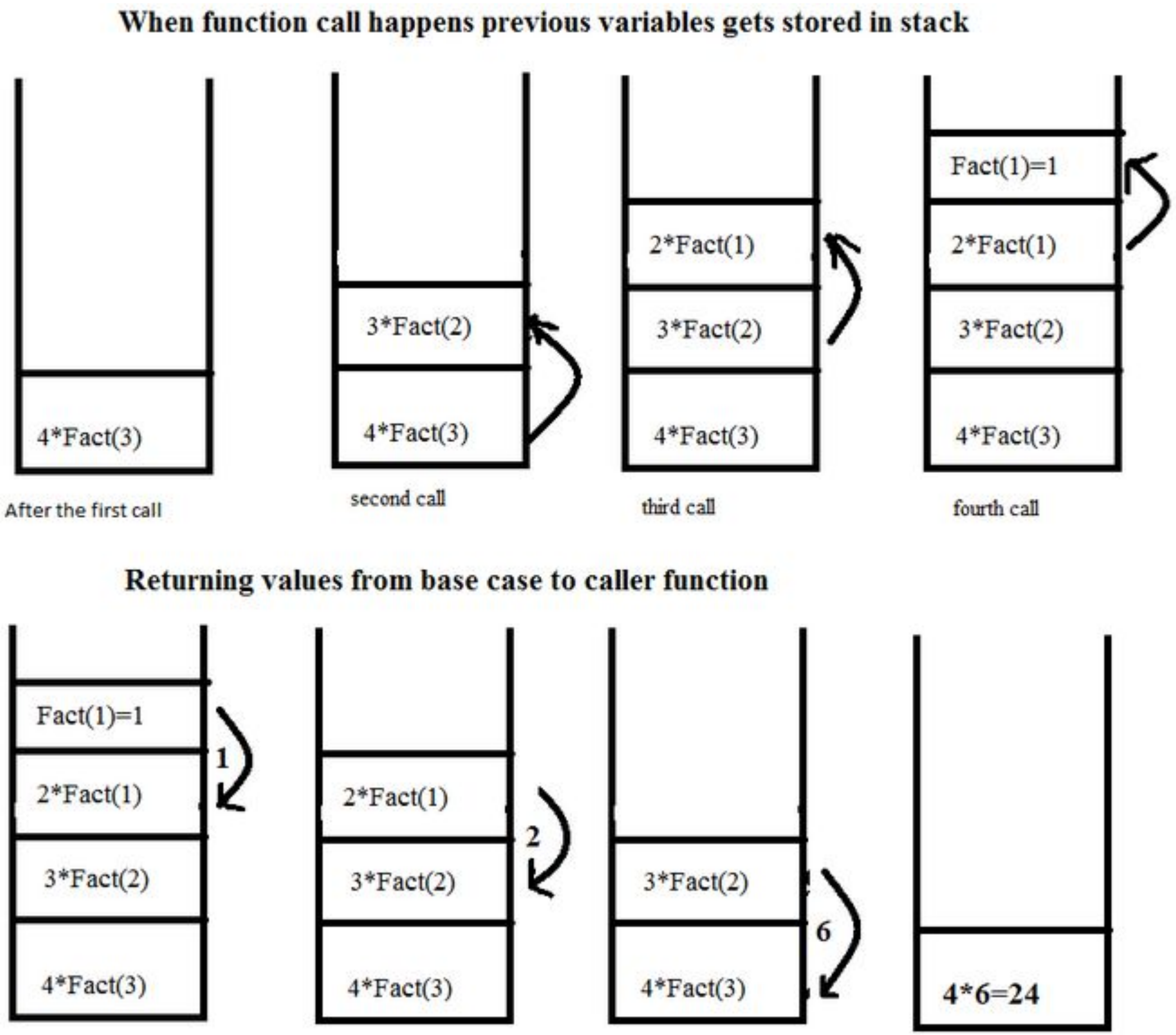
Example: Factorial Using Recursion

```c
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

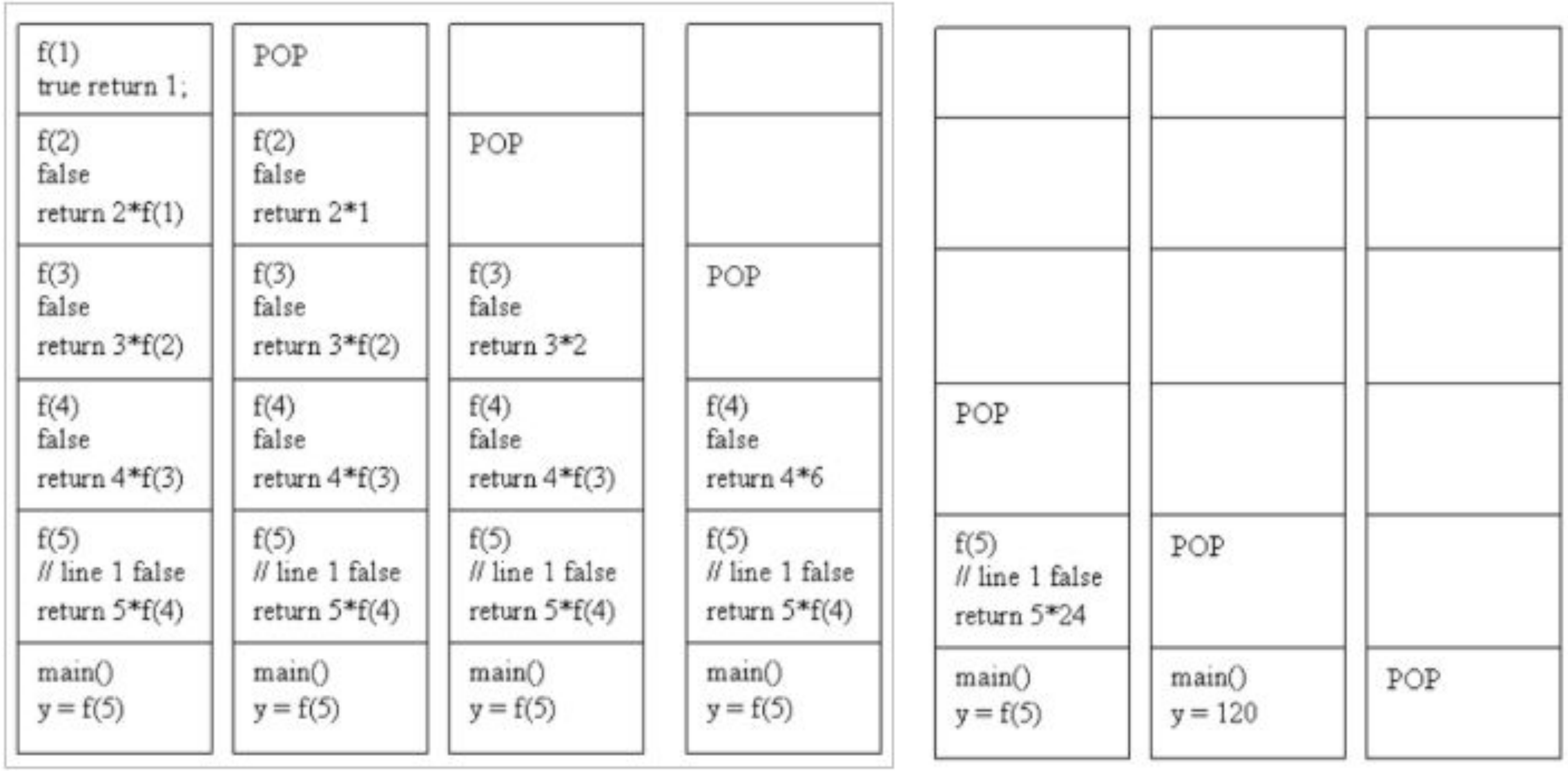
Call: `factorial(3)`

Calls stack:

```matlab
factorial(3)
→ 3 * factorial(2)
→ 3 * 2 * factorial(1)
→ 3 * 2 * 1 * factorial(0)
→ 3 * 2 * 1 * 1 = 6
```



RECURSION



f(1)
true return 1;
f(2)
false
return 2*f(1)
f(3)
false
return 3*f(2)
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(2)
false
return 2*1
f(3)
false
return 3*f(2)
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(3)
false
return 3*2
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(4)
false
return 4*6
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(5)
// line 1 false
return 5*24
main()
y = f(5)
POP
main()
y = 120
POP

INTRODUCTION TO QUEUE

A **Queue** is a **linear data structure** that follows the **FIFO** principle – **First In, First Out**. This means that the element inserted first is the one that will be removed first.

Basic Concept

Imagine a line (queue) at a ticket counter:

- The person who comes first gets served first.
- The new person joins at the **rear** of the queue.
- The person at the **front** gets served (removed) first.

INTRODUCTION TO QUEUE

Main Operations

1. **Enqueue** – Insert an element at the **rear** of the queue.
2. **Dequeue** – Remove an element from the **front** of the queue.
3. **Front** – Get the element at the front without removing it.
4. **IsEmpty** – Check if the queue is empty.
5. **IsFull** – (in case of fixed size queue) check if the queue is full.

Types of Queues

Type	Description
Simple Queue	Basic FIFO queue
Circular Queue	Rear wraps around to the front when the queue reaches the end
Priority Queue	Elements are dequeued based on priority, not order
Deque (Double-Ended Queue)	Insert and deletion from both front and rear ends

QUEUE- ADT

🎯 Queue ADT – Overview

A **Queue ADT** is a linear data structure that follows the **FIFO** principle:

First In, First Out

→ The element inserted **first** is removed **first**.

🛠 Core Operations in Queue ADT

Operation	Description
`enqueue(x)`	Insert element `x` at the **rear** of the queue
`dequeue()`	Remove and return the element at the **front**
`peek()`	Return the **front** element without removing it
`isEmpty()`	Return `true` if the queue is empty
`isFull()`	(For fixed size queues) return `true` if queue is full

Somaiya
TRUST

ENQUEUE OPERATION

✅ Enqueue Operation in Queue

```
Step 1: IF REAR = MAX-1
              Write OVERFLOW
              Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
              SET FRONT = REAR = 0
        ELSE
              SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

📌 Definition:

Enqueue is the operation used to **insert (add) an element** at the **rear** (end) of the queue.

Since a queue follows the **FIFO (First In, First Out)** principle, new elements always enter from the **rear**.

🧠 Terminology:

- `front` → points to the **first** element (for deletion)
- `rear` → points to the **last** element (for insertion)

⚠️ Overflow Condition:

Occurs when:

- Array is full (`rear == MAX - 1`)
- No more insertions possible unless elements are dequeued

Somaiya
TRUST

DEQUEUE OPERATION

✅ Dequeue Operation in Queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

📌 Definition:

The **Dequeue** operation **removes and returns** the **front element** from the queue.

Since a queue follows the **FIFO (First In, First Out)** principle, the **oldest (first inserted)** element is removed first.

🧠 Terminology:

Term	Meaning
`front`	Points to the element to remove (dequeue)
`rear`	Points to the last inserted element

⚠️ Underflow Condition:

Occurs when:

- `front == -1` (queue was never filled)
- OR `front > rear` (all elements were dequeued)

ISEMPTY() OPERATION

✅ isEmpty Operation in Queue

📌 Definition:

The `isEmpty()` operation is used to **check whether the queue contains any elements or not.**

It helps to **prevent underflow** during a `dequeue()` operation.

🔧 Condition to Check if Queue is Empty

👉 For Array-Based Queue:

c

```c
if (front == -1 || front > rear)
    → Queue is Empty
```

📘 Pseudocode:

plaintext

```
function isEmpty()
    if front == -1 or front > rear then
        return true
    else
        return false
```

Somaiya TRUST

ISFULL() OPERATION

📌 Definition:

The `isFull()` operation checks whether the **queue has reached its maximum capacity**, i.e., no more elements can be inserted.

It is **used to avoid overflow** before performing an `enqueue()` operation — especially in **array-based queues**.

🔧 Condition to Check if Queue is Full

👉 For **Linear Array-Based Queue**:

c

Copy Edit

```c
if (rear == MAX - 1)
    → Queue is Full
```

- `MAX` is the size of the array
- This happens when the rear reaches the end, even if there are free spaces at the front (due to dequeued elements)

📘 Pseudocode (Linear Queue):

plaintext

```
function isFull()
    if rear == MAX - 1 then
        return true
    else
        return false
```

TRUST

ARRAY IMPLEMENTATION OF QUEUE

Queue Using Array – Basic Structure

We implement a queue using:

- A **fixed-size array** (e.g., `int queue[MAX]`)
- Two integer variables:
  - `front` : Index of the first element (used for **deletion**)
  - `rear` : Index of the last element (used for **insertion**)

Basic Operations in Array-Based Queue

---

1. Enqueue (Insert element at rear)

- Check if `rear == MAX - 1` → **Overflow**
- If queue is initially empty (`front == -1`), set `front = 0`
- Increment `rear` and insert the new element

```c
rear++;
queue[rear] = value;
```

2. Dequeue (Remove element from front)

- Check if `front == -1` or `front > rear` → **Underflow**
- Remove and return the element at `queue[front]`
- Increment `front` by 1

```c
value = queue[front];
front++;
```

---

3. Peek (View front element)

- Return `queue[front]` without removing it
- Ensure queue is not empty before accessing

4. isEmpty()

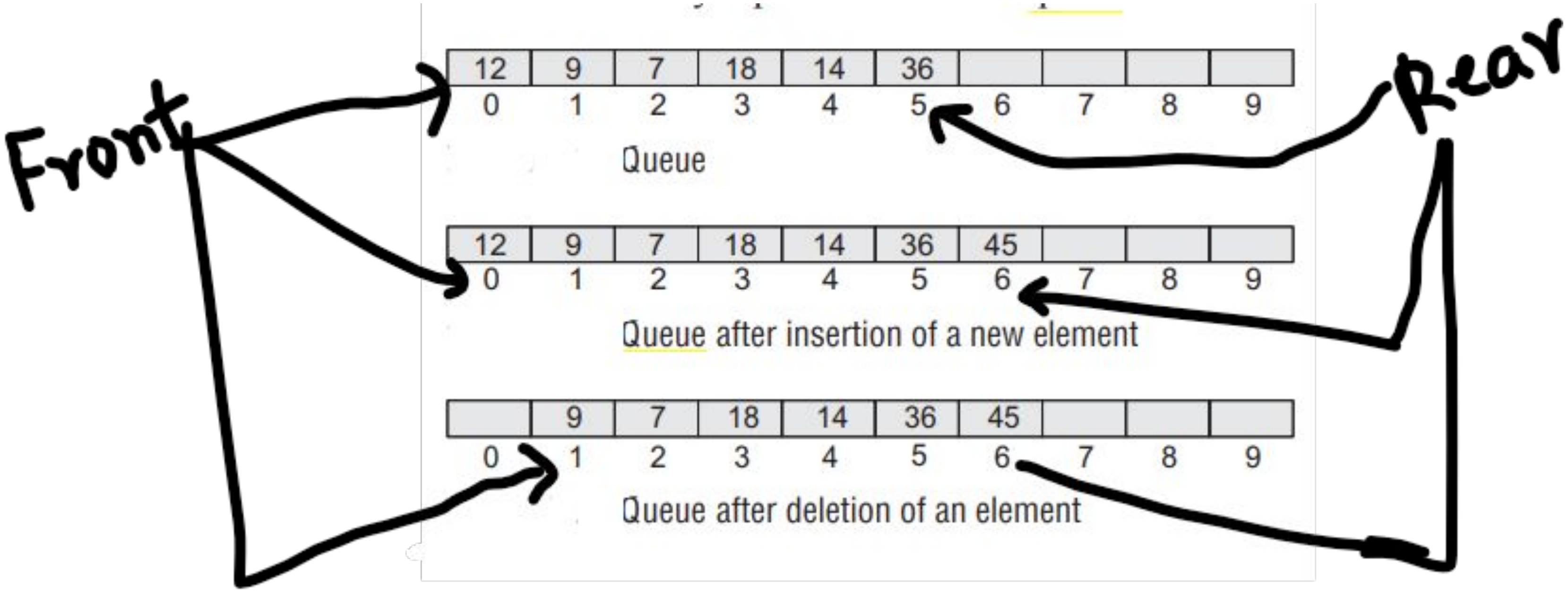
```c
if (front == -1 || front > rear)
```

---

5. isFull()

```c
if (rear == MAX - 1)
```

ARRAY IMPLEMENTATION OF QUEUE



Queue

Queue after insertion of a new element

Queue after deletion of an element

ARRAY IMPLEMENTATION OF QUEUE

✅ C Program: Linear Queue Using Array

```c
#include <stdio.h>
#define MAX 100

int queue[MAX];
int front = -1, rear = -1;

// Check if queue is empty
int isEmpty() {
    return front == -1 || front > rear;
}

// Check if queue is full
int isFull() {
    return rear == MAX - 1;
}
```

```c
// Enqueue operation
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow\n");
        return;
    }
    if (front == -1) {
        front = 0;
    }
    rear++;
    queue[rear] = value;
    printf("%d inserted into queue\n", value);
}

// Dequeue operation
int dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow\n");
        return -1;
    }
    int removed = queue[front];
    front++;
    return removed;
}
```

```c
// Peek operation
int peek() {
    if (isEmpty()) {
        printf("Queue is Empty\n");
        return -1;
    }
    return queue[front];
}

// Display the queue
void display() {
    if (isEmpty()) {
        printf("Queue is Empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = front; i <= rear; i++) {
        printf("%d ", queue[i]);
    }
    printf("\n");
}
```

ARRAY IMPLEMENTATION OF QUEUE

```c
// Main function
int main() {
    enqueue(10);
    enqueue(20);
    enqueue(30);
    display();

    printf("Front element: %d\n", peek());

    printf("%d removed from queue\n", dequeue());
    display();

    return 0;
}
```

📊 Expected Console Output (Visualization)

```plaintext
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue contents: FRONT → [10] [20] [30] ← REAR

Dequeued: 10
Queue contents: FRONT → [20] [30] ← REAR

Enqueued: 40
Enqueued: 50
Queue Overflow
Queue contents: FRONT → [20] [30] [40] [50] ← REAR
```

📌 Explanation of Output Visualization:

- `FRONT →` shows the **start** of the queue
- Values are enclosed in `[ ]` brackets
- `← REAR` shows the **end** of the queue
- You get **real-time updates** as elements are inserted or removed

Somaiya TRUST

TYPES OF QUEUES

- CIRCULAR QUEUES
- DOUBLE ENDED QUEUE
- PRIORITY QUEUE

CIRCULAR QUEUE

✅ What is a Circular Queue?

A **Circular Queue** is a linear data structure that uses a **fixed-size array** in a **circular manner**.
Unlike a regular (linear) queue, when the **rear** reaches the end of the array and there's space at the **front**, it **wraps around** to reuse the empty space.

🧠 Why Circular Queue?

In a normal array-based queue:

- After several `dequeue()` operations, space is wasted at the front
- Even if the array has free space, the queue can become "full"

👉 A **circular queue** solves this by treating the array as **circular** — when the end is reached, it wraps around to the beginning.

📌 Key Terms:

Term	Meaning
`front`	Points to the **first** element (to dequeue)
`rear`	Points to the **last** element (to enqueue)

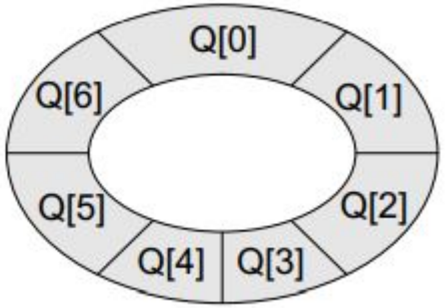


Figure 8.15 Circular queue

CIRCULAR QUEUE

Circular Queue

Drawbacks of normal queue:

54	9	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Figure 8.13 Linear queue

Here, FRONT = 0 and REAR = 9.

		7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

Figure 8.14 Queue after two successive deletions

CIRCULAR QUEUE

Conditions in Circular Queue:

1. **Empty Queue:**

```c
front == -1
```

2. **Full Queue:**

```c
(rear + 1) % MAX == front
```

3. **Enqueue:**
   - If empty → set `front = rear = 0`
   - Else → `rear = (rear + 1) % MAX`

4. **Dequeue:**
   - If `front == rear` → reset both to -1
   - Else → `front = (front + 1) % MAX`

CIRCULAR QUEUE- ENQUEUE()

Circular Queue Implementation

To insert an element we now have to check for the following three conditions:

- If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 1which illustrates this point.
- If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig. 2
- If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig. 3

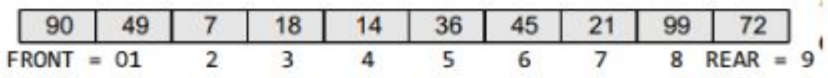


Figure 1 Full queue

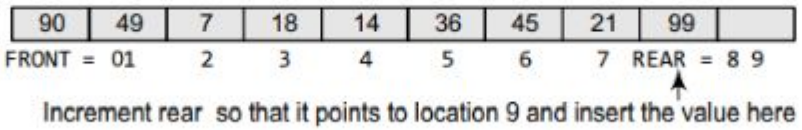
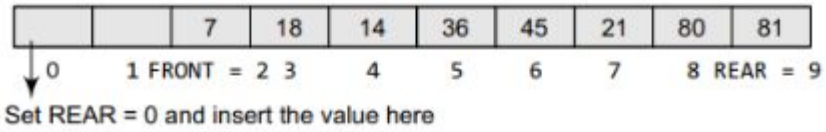


Figure 2 Queue with vacant locations




```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

CIRCULAR QUEUE- DEQUEUE()

To delete an element, again we check for three conditions.

- Look at Fig. 1. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 2.
- If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0. This is shown in Fig. 3

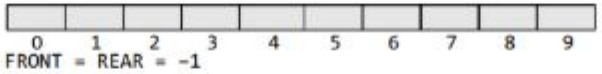


Figure 1 Empty queue

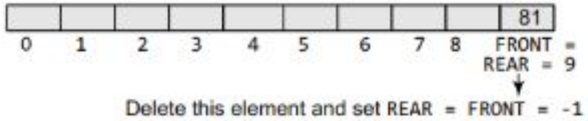


Figure 2 Queue with a single element

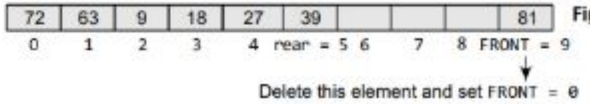
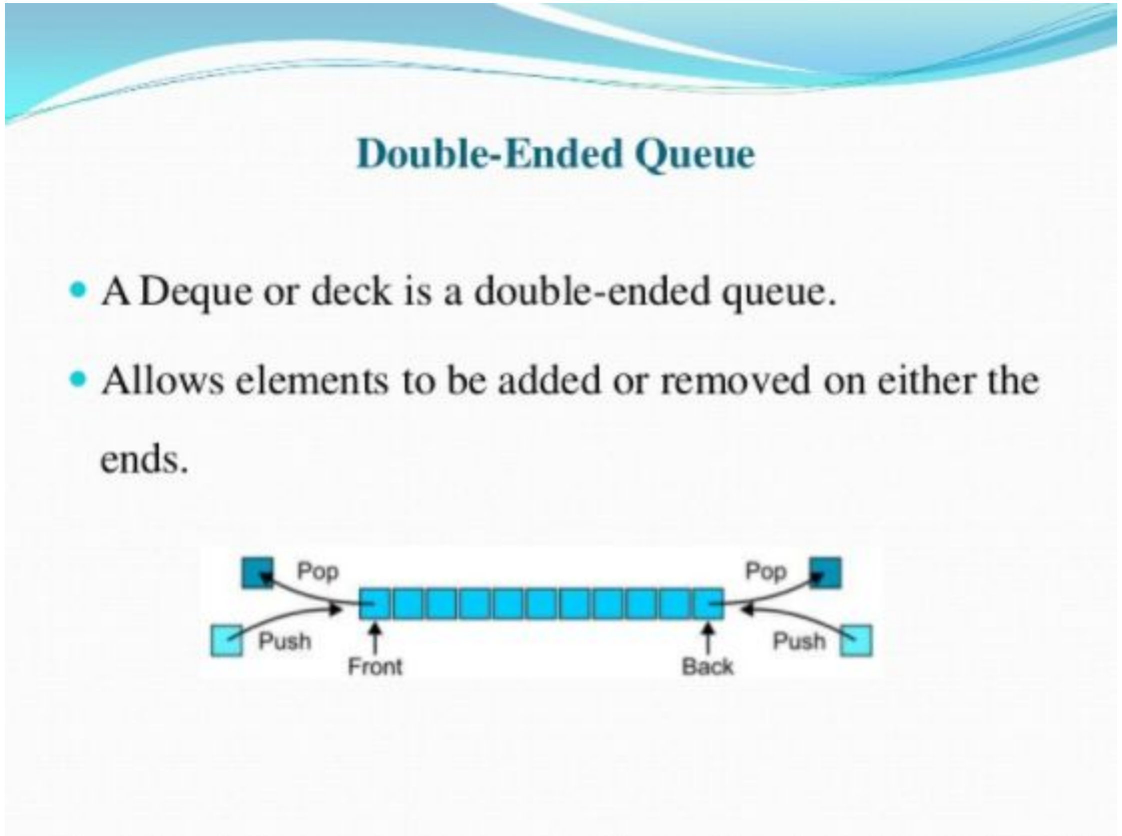


Figure 3 Queue where FRONT = MAX-1 before deletion

```
Step 1: IF FRONT = -1
              Write "UNDERFLOW"
              Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
              SET FRONT = REAR = -1
         ELSE
              IF FRONT = MAX -1
                     SET FRONT = 0
              ELSE
                     SET FRONT = FRONT + 1
              [END of IF]
        [END OF IF]
Step 4: EXIT
```

Figure 8.23 Algorithm to delete an element from a circular queue

DOUBLE ENDED QUEUE (DEQUE)



DOUBLE ENDED QUEUE(DEQUE)

TYPES OF DEQUE

- **Input restricted Deque**
  - Elements can be inserted only at one end.
  - Elements can be removed from both the ends.
- **Output restricted Deque**
  - Elements can be removed only at one end.
  - Elements can be inserted from both the ends.

Deque as Stack and Queue

As STACK

- When insertion and deletion is made at the same side.

As Queue

- When items are inserted at one end and removed at the other end.

DOUBLE ENDED QUEUE(DEQUE)

OPERATIONS IN DEQUE

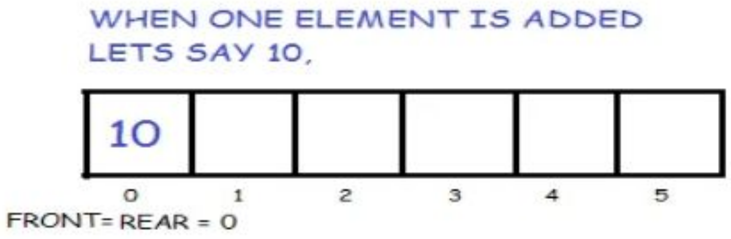
- Insert element at back
- Insert element at front
- Remove element at front
- Remove element at back

DOUBLE ENDED QUEUE(DEQUE)- INSERTION

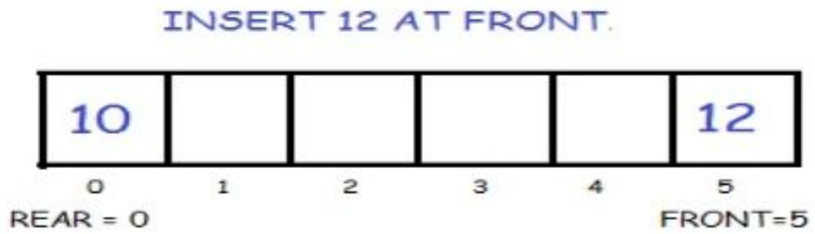
Insert element at front

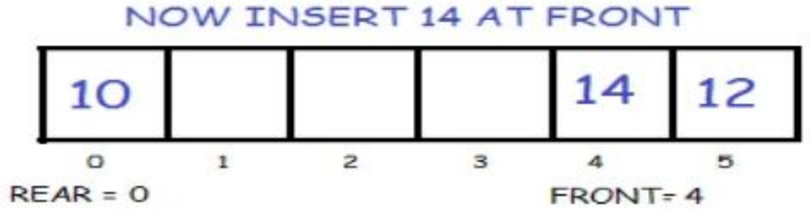
First we check if the queue is full. If its not full we insert an element at front end by following the given conditions :

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.



- Else we decrement front and insert the element. Since we are using circular array, we have to keep in mind that if front is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.






```
void insert_left()
{
        int val;
        printf("\n Enter the value to be added:");
        scanf("%d", &val);
        if((left == 0 && right == MAX-1) || (left == right+1))
        {
                printf("\n OVERFLOW");
                return;
        }
```

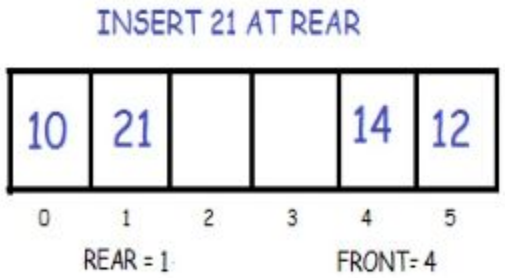
```
        if (left == -1)/*If queue is initially empty*/
        {
                left = 0;
                right = 0;
        }
        else
        {
                if(left == 0)
                        left=MAX-1;
                else
                        left=left-1;
        }
        deque[left] = val;
}
```

DOUBLE ENDED QUEUE(DEQUE)- INSERTION

Insert element at Back

Again we check if the queue is full. If its not full we insert an element at back by following the given conditions:

- If the queue is empty then intialize front and rear to 0. Both will point to the first element.
- Else we increment rear and insert the element. Since we are using circular array, we have to keep in mind that if rear is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.




```
void insert_right()
{
        int val;
        printf("\n Enter the value to be added:");
        scanf("%d", &val);
        if((left == 0 && right == MAX-1) || (left == right+1))
        {
                printf("\n OVERFLOW");
                return;
        }
        if (left == -1) /* if queue is initially empty */
        {
                left = 0;
                right = 0;
        }
        else
        {
                if(right == MAX-1) /*right is at last position of queue */
                        right = 0;
                else
                        right = right+1;
        }
        deque[right] = val ;
}
```

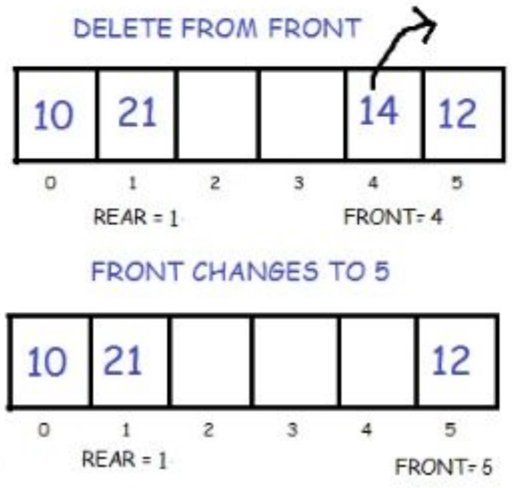
Somaiya TRUST

DOUBLE ENDED QUEUE(DEQUE)- DELETION

Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

- If only one element is present we once again make front and rear equal to -1.
- Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.

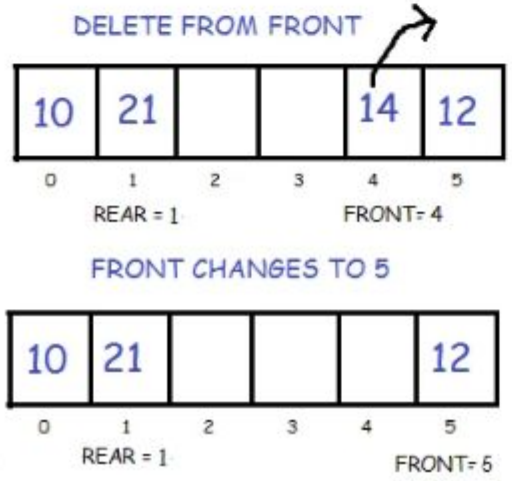


DOUBLE ENDED QUEUE(DEQUE)- DELETION

Delete First Element

In order to do this, we first check if the queue is empty. If its not then delete the front element by following the given conditions :

- If only one element is present we once again make front and rear equal to -1.
- Else we increment front. But we have to keep in mind that if front is equal to SIZE-1 then instead of increasing it by 1 we make it equal to 0.



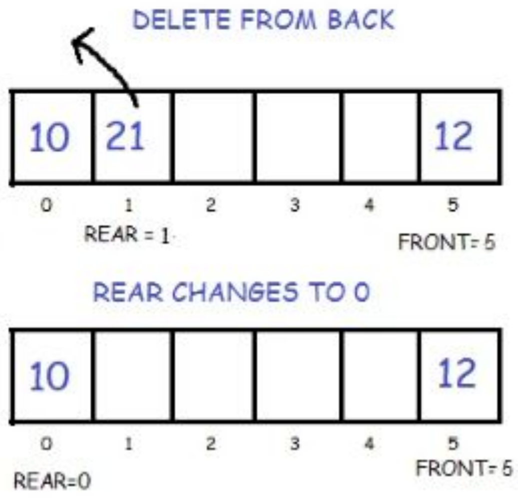

```
void delete_left()
{
        if (left == -1)
        {
                printf("\n UNDERFLOW");
                return ;
        }
        printf("\n The deleted element is : %d", deque[left]);
        if(left == right) /*Queue has only one element */
        {
                left = -1;
                right = -1;
        }
        else
        {
                if(left == MAX-1)
                        left = 0;
                else
                        left = left+1;
        }
}
```

DOUBLE ENDED QUEUE(DEQUE)- DELETION

Delete Last Element

Inorder to do this, we again first check if the queue is empty. If its not then we delete the last element by following the given conditions :

- If only one element is present we make front and rear equal to -1.
- Else we decrement rear. But we have to keep in mind that if rear is equal to 0 then instead of decreasing it by 1 we make it equal to SIZE-1.




```
void delete_right()
{
        if (left == -1)
        {
                printf("\n UNDERFLOW");
                return ;
        }
        printf("\n The element deleted is : %d", deque[right]);
        if(left == right) /*queue has only one element*/
        {
                left = -1;
                right = -1;
        }
        else
        {
                if(right == 0)
                        right=MAX-1;
                else
                        right=right-1;
        }
}
```

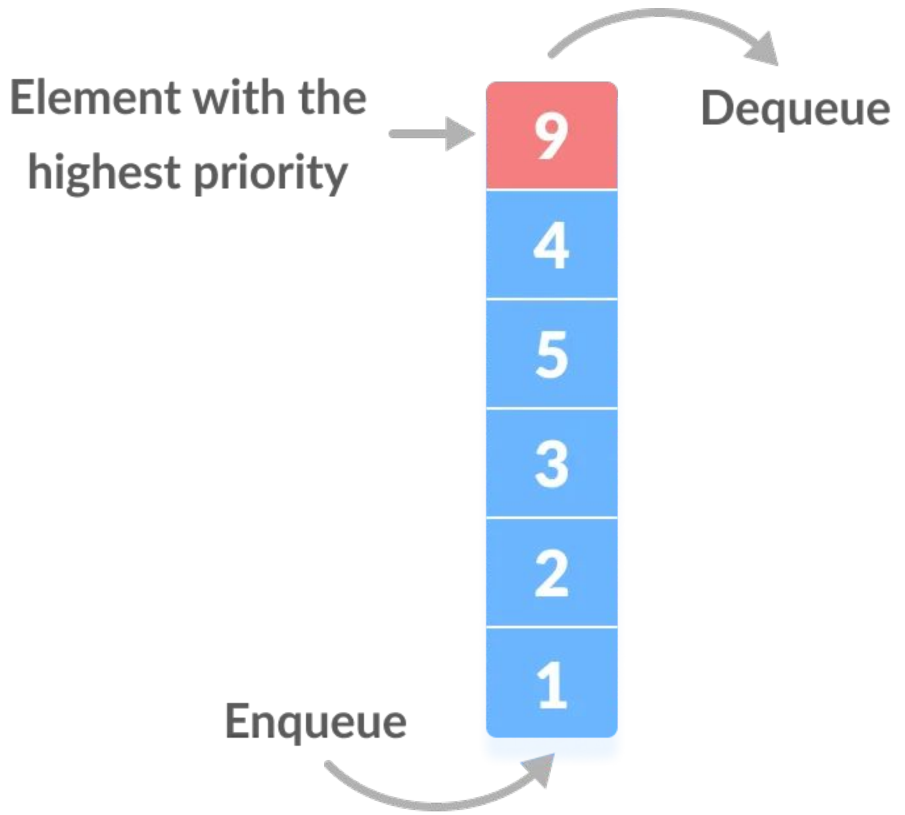
PRIORITY QUEUE

📌 What is a Priority Queue?

A **Priority Queue** is a special type of queue in which **each element is associated with a priority**, and **elements are dequeued based on their priority**, not just their insertion order.

👉 Unlike regular queues (FIFO), in a **priority queue**:

- The element with the **highest priority is served first**
- If two elements have the same priority, they are dequeued in **insertion order**



🧠 Real-Life Example:

Think of a hospital emergency room:

- Patients with **higher urgency** are treated first, not just the ones who arrived first.

🔧 Operations in Priority Queue

Operation	Description
`insert(data, priority)`	Insert element with a specific priority
`delete()`	Remove and return element with **highest priority**
`peek()`	View element with highest priority (without removing)

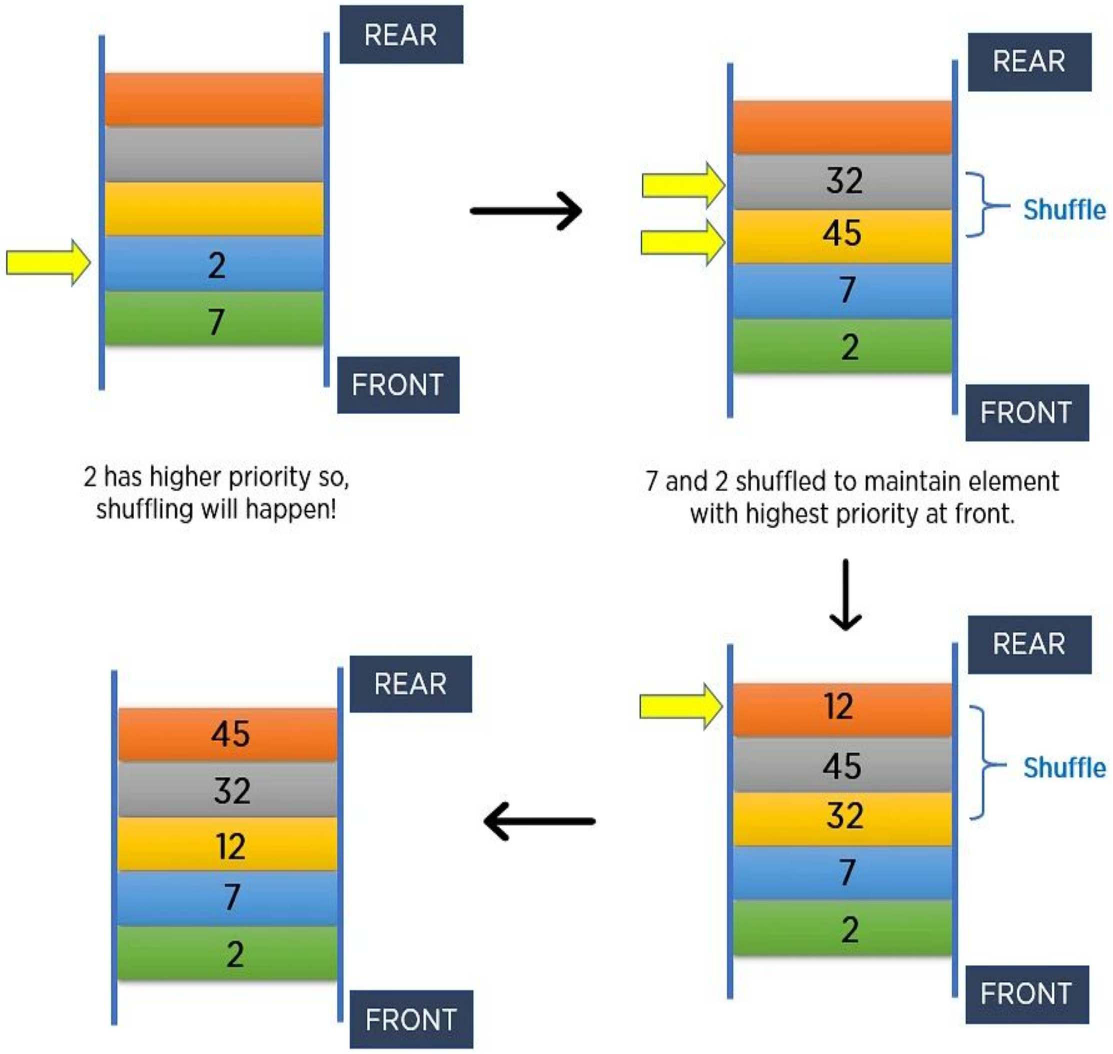
PRIORITY QUEUE

PRIORITY QUEUE

Consider you have to insert 7, 2, 45, 32, and 12 in a priority queue.

The element with the least value has the highest property.

Thus, you should maintain the lowest element at the front node.



APPLICATIONS OF QUEUE

Queues are widely used in **computer science**, **operating systems**, **networking**, and **real-life systems** because of their **FIFO (First In, First Out)** behavior.

Here's a categorized explanation:

🔧 1. Operating Systems

Application	Description
CPU Scheduling	Jobs are kept in a queue for execution based on arrival time or priority
Disk Scheduling	Disk I/O requests are queued and serviced sequentially
Print Spooler	Print jobs are queued and printed one after another
Task Management	Background tasks are scheduled and executed using queues

Somaiya
TRUST

APPLICATIONS OF QUEUE

🌐 2. Networking

Application	Description
Packet Scheduling	Network routers use queues to manage packets waiting for transmission
Buffering	Audio/video streaming uses queues for smooth data playback
Load Balancing	Requests are queued before assigning to servers

🔁 3. Data Structures and Algorithms

Application	Description
Breadth First Search (BFS)	BFS in graphs/trees uses a queue to track nodes to visit
Tree Traversals	Level-order traversal uses a queue
Topological Sorting	Uses queue to process nodes with 0 in-degree

Somaiya TRUST

APPLICATIONS OF QUEUE

🏢 4. Real-Life Applications

Application	Description
Ticket Booking Counters	Customers are served in the order they arrive
Call Centers / Helpdesks	Caller requests are processed in queue order
Elevator Systems	Floors requested are queued and served
Banking Systems	Customers wait in a queue for service

🔄 5. Software Development

Application	Description
Task Scheduling	Asynchronous tasks (like in JavaScript or Python) are placed in event queues
Message Queues (MQs)	Systems like RabbitMQ, Kafka use queues for inter-process communication
Job Queues in Web Servers	Jobs are queued for background processing (e.g., Celery, Redis Queue)

SAMPLE QUESTIONS

QUESTION NO.	SAMPLE QUESTIONS MODULE 2
1	Explain stack data structure with algorithm using example
2	Explain Queue DS with algorithm and using example
3	Explain the various operations performed on stack
4	Explain various operations performed on queue
5	Explain the concept of a circular queue? How is it better than a linear queue?
6	Explain how recursion can be used in stack
7	Explain push and pop operations of stack with algorithm
8	Explain the enqueue() and dequeue() operations of queue with algorithm
9	Write algorithm for circular queue insertion and deletion
10	Write algorithm for double ended queue insertion at the front, at the back, deletion at the front and deletion at the back
11	Explain priority queue with example

Somaiya
TRUST

SAMPLE QUESTIONS

<table>
<tr><th>QUESTION NO.</th><th>SAMPLE QUESTIONS MODULE 2</th></tr>
<tr><td>11</td><td>Draw the stack structure in each case when the following operations are performed on an empty stack. 10M
(a) Add A, B, C, D, E, F
(b) Delete two letters
(c) Add G
(d) Add H
(e) Delete four letters
(f) Add I</td></tr>
<tr><td>12</td><td>5. Consider the queue given below which has FRONT = 1 and REAR = 5.
<table><tr><td></td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td></td><td></td><td></td><td></td></tr></table>Now perform the following operations on the queue:
(a) Add F (b) Delete two letters
(c) Add G (d) Add H
(e) Delete four letters (f) Add I</td></tr>
<tr><td>13</td><td>Explain well formed ness of parenthesis using algorithm and example</td></tr>
<tr><td>14</td><td>STUDY PROBLEMS ON INFIX TO POSTFIX AND POSTFIX EVALUATION (MOST IMPORTANT)</td></tr>
</table>

T R U S T