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-I


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS
#include <iostream>
using nameepace std;
int main()
{
cout << "Hello, World!";
return 0;
}

DATA STRUCTURE

— INTRODUCTION TO DATA STRUCTURE —

A **Data Structure** is a way of **organizing** and **storing** data in a computer so that it can be used **efficiently**.

WHAT IS DATA STRUCTURE?

A data structure is a **collection** of data values, the **relationships** among them, and the **functions** or **operations** that can be applied to the data.

In simple words:

It is a way to **store**, **organize** and manage data in memory so that it can be accessed and modified efficiently.

WHY DO WE NEED DATA STRUCTURES?

- **Efficiency** – Helps in processing data faster and using memory wisely.
- **Organization** – Keeps data well structured and easy to manage.
- **Easy Access** – Allows quick searching, insertion, deletion and other operations.
- **Problem Solving** – Essential for solving complex real-world problems.

CHARACTERISTICS OF DATA STRUCTURE

Organization
Data is organized in a specific manner.

Storage
Data is stored in computer memory.

Operations
Supports various operations like insert, delete, search, update.

Efficiency
Designed to perform operations in the most efficient way possible.

Reusability
Data structures can be reused in different applications.

TYPES OF DATA STRUCTURES

Data structures are broadly classified into two types:

1. Linear Data Structure

Elements are arranged in a sequential manner.

Examples:

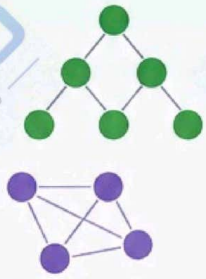
- Array
- Linked List
- Stack
- Queue

2. Non-Linear Data Structure

Elements are not arranged sequentially.

Examples:

- Tree
- Graph
- Heap



COMMON OPERATIONS

- **Insert** – Add a new element into the data structure.
- **Delete** – Remove an element from the data structure.
- **Search** – Find the location of an element.
- **Update** – Modify the value of an existing element.
- **Traverse** – Access each element exactly once.

KEY TAKEAWAY Data structures are the foundation of efficient programming. Choosing the right data structure for the right problem makes programs faster, simpler and more effective.

TYPES OF DATA STRUCTURE

Primitive & Non-Primitive

Data structures are of **two main types** based on the type of data they store and the operations that can be performed.

1. PRIMITIVE DATA STRUCTURE

Primitive data structures are the basic data types that store a **single value**.

Examples

123	3.14	'A'	True
Integer (int)	Float (float)	Character (char)	Boolean (bool)

Features

- Store a single value.
- Built-in data types of a programming language.
- Directly supported by the system.
- Fixed size and simple.

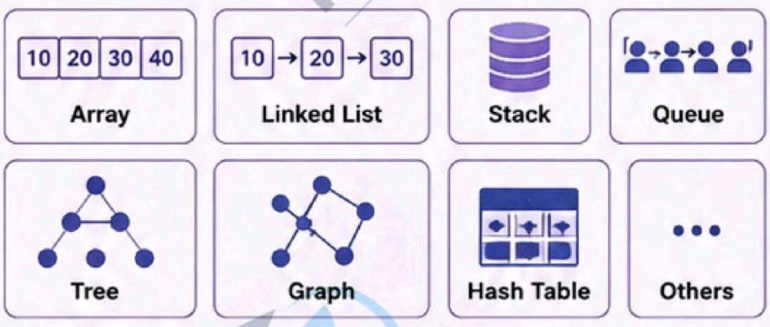
Examples in C/Java

```
int a = 10;    float b = 3.5;    char ch = 'A';    boolean flag = true;
```

2. NON-PRIMITIVE DATA STRUCTURE

Non-primitive data structures are derived types that can store **multiple values**.

Examples



Features

- Store multiple values (collection of data).
- User-defined or derived data types.
- More complex.
- Require more memory.
- Can represent relationships between data.

Examples in C/Java

```
int arr[4] = {10, 20, 30, 40};                    // Array
struct Node { int data; struct Node* next; };     // Linked List
Stack s;    Queue q;    Tree t;    Graph g;
```

PRIMITIVE vs NON-PRIMITIVE DATA STRUCTURE

Aspect	Primitive Data Structure	Non-Primitive Data Structure
Definition	Stores a single value.	Stores multiple values (collection of data).
Built-in / User-defined	Built-in (predefined).	User-defined.
Size	Fixed size.	Variable size.
Complexity	Simple.	Complex.
Memory	Less memory.	More memory.
Examples	int, float, char, bool	Array, Linked List, Stack, Queue, Tree, Graph, etc.
Supported Operations	Basic operations only.	Many operations (insert, delete, search, traverse, etc.).

SUMMARY

Primitive data structures hold a single value and are simple.
Non-primitive data structures hold multiple values and help in organizing complex data efficiently.

LINEAR AND NON-LINEAR DATA STRUCTURE

Data structures are broadly classified based on the **arrangement** of elements and the **relationships** between them.

1. LINEAR DATA STRUCTURE

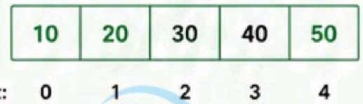
In linear data structures, elements are arranged in a **sequential order**. Each element (except the first and last) has exactly **one predecessor** and **one successor**.

Characteristics

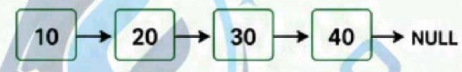
- Elements are stored in a sequence.
- There is a one-to-one relationship between elements.
- Traversal is done in a single path.

Examples

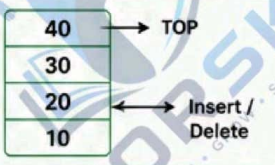
- **Array**



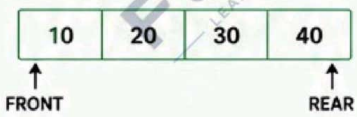
- **Linked List**



- **Stack**



- **Queue**



Applications

Managing lists of data

Function calls, undo operations

CPU scheduling, printer queue

2. NON-LINEAR DATA STRUCTURE

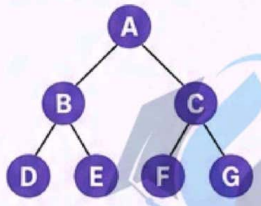
In non-linear data structures, elements are **not** arranged in a **sequential order**. One element can be connected to **multiple other elements**.

Characteristics

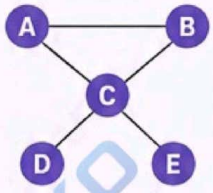
- Elements are not stored in sequence.
- There is a one-to-many or many-to-many relationship.
- Traversal may have multiple paths.

Examples

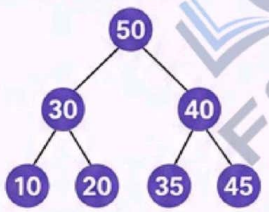
- **Tree**



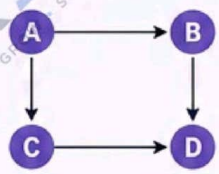
- **Graph**



- **Heap**



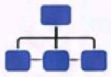
- **Graph (Directed)**



Applications

File systems, organization hierarchies

Network routing, social networks



Compilers, AI, databases

LINEAR vs NON-LINEAR DATA STRUCTURE

Aspect	Linear Data Structure	Non-Linear Data Structure
Arrangement	Elements are arranged in a sequence.	Elements are not arranged in sequence.
Relationship	One-to-one relationship.	One-to-many or many-to-many relationship.
Traversal	Single path.	Multiple paths.
Complexity	Generally simpler.	Generally more complex.
Memory Usage	Less memory (in most cases).	More memory (due to links/pointers).
Examples	Array, Linked List, Stack, Queue.	Tree, Graph, Heap, etc.
Applications	Simple applications, linear processing.	Complex applications, hierarchical relationships.

KEY TAKEAWAY Use linear data structures when data is sequential.
Use non-linear data structures when data has hierarchical or network relationships.

DATA STRUCTURE OPERATIONS

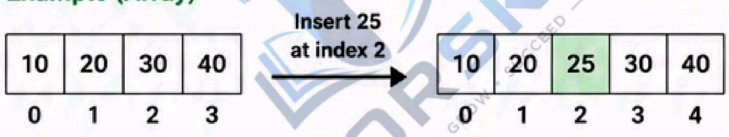
Operations are the basic actions that can be performed on data structures to manipulate the data stored in them.

COMMON OPERATIONS

1 INSERT

Insertion is the process of adding a new element at a specific position in the data structure.

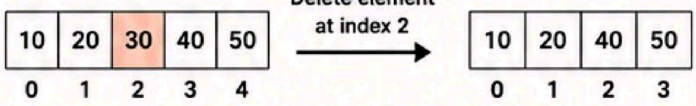
Example (Array)



2 DELETE

Deletion is the process of removing an element from the data structure.

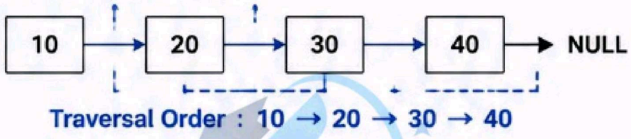
Example (Array)



3 TRAVERSE

Traversal is the process of visiting each element of the data structure exactly once in a particular order.

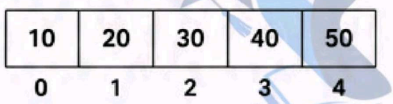
Example (Linked List)



4 SEARCH

Searching is the process of finding the location (index or address) of an element in the data structure.

Example (Array)



Search key = 40
Found at index 3.

5 UPDATE

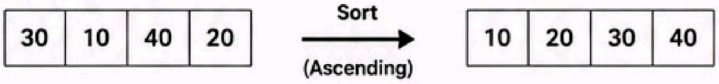
Updating is the process of modifying the value of an existing element.

Example (Array)

6 SORT

Sorting is the process of arranging elements in ascending of descending order.

Example (Array)



SUMMARY TABLE

Operation	Purpose	Where Used
1 Insert	Add a new element	Array, Linked List, Stack, Queue, Tree, Graph, etc.
2 Delete	Remove an existing element	All data structures
3 Traverse	Visit each element exactly once	Linked List, Tree, Graph, Array, etc.
4 Search	Find the position of an element	Array, Linked List, Tree, Graph, etc.
5 Update	Modify an existing element	Array, Linked List, Tree, Graph, etc.
6 Sort	Arrange elements in order	Array, List, etc.

KEY TAKEAWAY

These basic operations allow efficient manipulation of data in different data structures. The choice of data structure depends on the type of operations to be performed and the problem to be solved.

ARRAY

Definition and Concepts, Memory Representations

1. DEFINITION

An **array** is a collection of **similar data elements** stored in **contiguous memory locations** and accessed using a **common name**.

2. CONCEPTS

- All elements are of the same data type.
- Elements are stored in contiguous memory locations.
- Elements are accessed using an index (or subscript).
- The index of the first element is 0.
- For an array of size N, valid indices are 0 to N–1.
- Direct access is possible. Any element can be accessed in constant time O(1).

3. BASIC TERMINOLOGIES

- **Element** : Each value stored in an array.
- **Index** : Position of an element in the array.
- **Size** : Total number of elements in the array.
- **Array Name** : Name used to represent the array.
- **Base Address** : Address of the first memory location of the array.
- **Contiguous** : Consecutive memory locations.

Example

Let A be an array of size 5.

A = [	10	20	30	40	50]
Index:	0	1	2	3	4

Access:

A[0] = 10
A[2] = 30
A[4] = 50

4. MEMORY REPRESENTATIONS OF ARRAY

4.1 ONE-DIMENSIONAL ARRAY (1D)

An array of elements arranged in a single row.

Example: A[5] = { 10, 20, 30, 40, 50 }

Array Elements	10	20	30	40	50
Index	0	1	2	3	4
Memory Address	1000	1004	1008	1012	1016

- Assume base address (BA) = 1000
- If each integer occupies 4 bytes.

4.2 TWO-DIMENSIONAL ARRAY (2D)

An array of elements arranged in rows and columns (matrix form).

Example: A[3][4]

Row Index (i) \ Column Index (j)	0	1	2	3
0	10	20	30	40
1	50	60	70	80
2	90	100	110	120

Access:

A[0][0] = 10
A[1][2] = 70
A[2][3] = 120

- Stored in memory in **row-major order** (row by row).

4.3 ADDRESS CALCULATION

Let,
BA = Base Address of array
W = Size of each element in bytes
i, j = Row and Column indices
N = Number of columns

(a) 1D Array

Address of A[i]

$$= BA + i \times W$$

Where, $0 \le i < N$

(b) 2D Array (Row-Major Order)

Address of A[i][j]

$$= BA + ((i \times N) + j) \times W$$

Where, $0 \le i < Rows$, $0 \le j < Columns$

Note:

Row-major order stores row by row in contiguous memory locations.

5. EXAMPLE: ADDRESS CALCULATION

(a) 1D Array

A[5], BA = 1000, W = 4 bytes
Find address of A[3]
Address = 1000 + 3 × 4 = 1012

(b) 2D Array

A[3][4], BA = 2000, W = 4 bytes, N = 4 (columns)
Find address of A[2][3]
Address = 2000 + ((2 × 4) + 3) × 4
= 2000 + (11 × 4) = 2044

6. CHARACTERISTICS OF ARRAY

- Fixed size (static memory allocation).
- Homogeneous elements (same data type).
- Contiguous memory allocation.
- Random access: O(1) time.
- Efficient traversal using loops.
- Insertion and deletion are costly (because of shifting).

7. COMMON OPERATIONS

- Traversal (visit each element)
- Insertion (at specific position)
- Deletion (at specific position)
- Searching (linear / binary)
- Sorting (ascending / descending)
- Updating (modify element)

KEY TAKEAWAY

Arrays store similar elements in contiguous memory locations and provide direct access to any element using its index. Understanding their memory representation is essential for efficient programming.

ARRAY OPERATIONS

Traversing, Insertion, Deletion

An array is a collection of **similar elements** stored in **contiguous memory** locations and accessed using an **index**.

1. TRAVERSING AN ARRAY

Traversing means **visiting each element** of the array exactly once in a specific order (usually from first to last).

Example

Let A = | 10 | 20 | 30 | 40 | 50 |

Index: 0 1 2 3 4

Traversal Order: 10 → 20 → 30 → 40 → 50

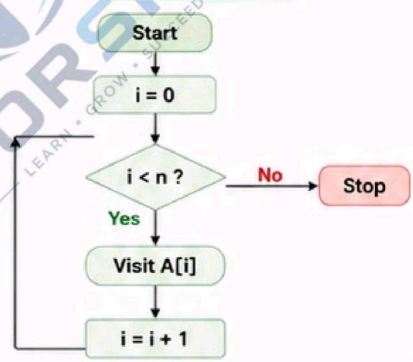
ALGORITHM (TRAVERSAL)

```
for i = 0 to n-1
    visit A[i]
end for
```

PSEUDOCODE

```
for i ← 0 to n - 1
    print A[i]
end for
```

Flow / Illustration



2. INSERTION IN AN ARRAY

Insertion means **adding a new element** at a specific position. Elements from that position are shifted one place to the right.

Example

Insert 25 at index 2 in A = [10 | 20 | 30 | 40 | 50] (Size = 5)

Step 1: Shift elements to the right from index 2

Index:	0	1	2 →	3 →	4 →
	10	20	30	40	50

Step 2: Insert 25 at index 2

Index:	0	1	(2)	3	4	5
	10	20	25	30	40	50

ALGORITHM (INSERTION)

1. Check if the array is full.
2. For i = n - 1 down to pos
   A[i + 1] = A[i]
3. A[pos] = newElement
4. n = n + 1

PSEUDOCODE

```
if n == MAX
    print "Array Overflow"
else
    for i ← n-1 downto pos
        A[i+1] ← A[i]
    A[pos] ← newElement
    n ← n + 1
end if
```

Illustration

Before Insertion

Index:	0	1	2	3	4
	10	20	30	40	50

↑ Insert 25 at index 2

After Insertion

Index:	0	1	2	3	4	5
	10	20	25	30	40	50

3. DELETION FROM AN ARRAY

Deletion means **removing an element** from a specific position. Elements after that position are shifted one place to the left.

Example

Delete element at index 2 (value 30) from A = [10 | 20 | 30 | 40 | 50] (Size = 5)

Step 1: Shift elements to the left from index 2

Index:	0	1	2 →	3 →	4
	10	20	30	40	50

Step 2: Reduce size by 1

Index:	0	1	2	3
	10	20	40	50

ALGORITHM (DELETION)

1. Check if the array is empty.
2. For i = pos to n - 2
   A[i] = A[i + 1]
3. n = n - 1

PSEUDOCODE

```
if n == 0
    print "Array Underflow"
else
    for i ← pos to n-2
        A[i] ← A[i+1]
    n ← n - 1
end if
```

Illustration

Before Deletion

Index:	0	1	2	3	4
	10	20	30	40	50

↑ Delete element at index 2 (30)

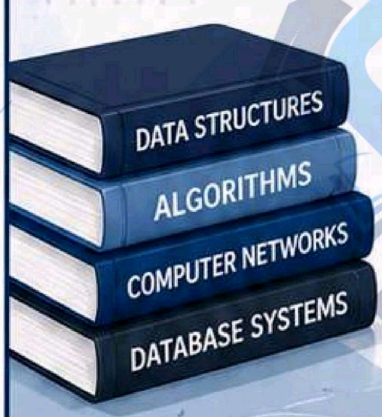
After Deletion

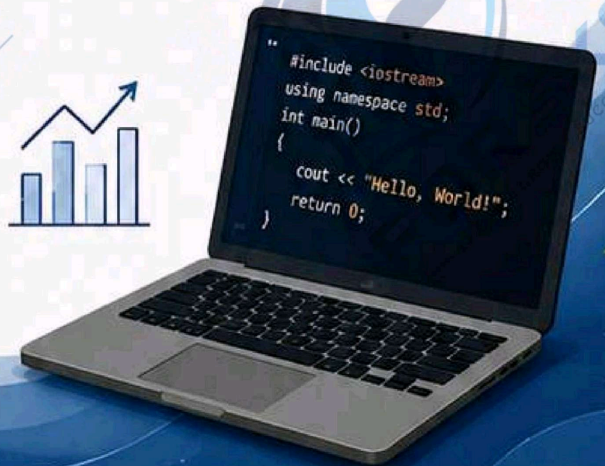
Index:	0	1	2	3	
	10	20	40	50	

KEY TAKEAWAY

- Traversing visits each element of the array.
- Insertion adds a new element and may require shifting.
- Deletion removes an element and may require shifting.

Unit - II


DATA STRUCTURES
ALGORITHMS
COMPUTER NETWORKS
DATABASE SYSTEMS

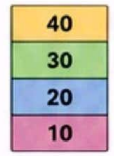

#include <iostream>
using namespace std;
int main()
{
cout << "Hello, World!";
return 0;
}

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)

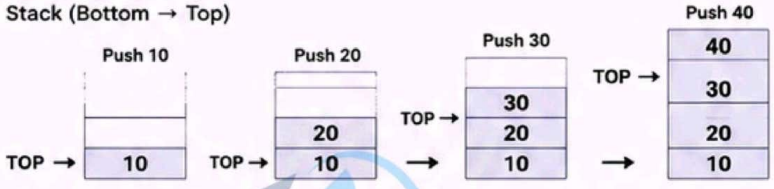
3. BASIC OPERATIONS

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)

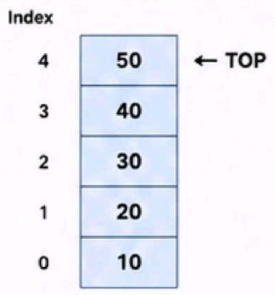


4. MEMORY REPRESENTATIONS OF STACK

4.1 STACK USING ARRAY

The stack is implemented using a one-dimensional array.

Representation



Example

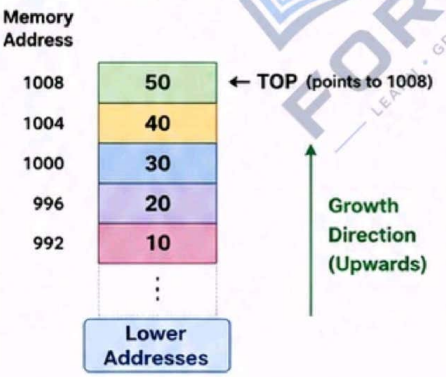
After pushing 10, 20, 30, 40, 50
TOP = 4 (points to 50)
Elements are stored in contiguous memory.

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.

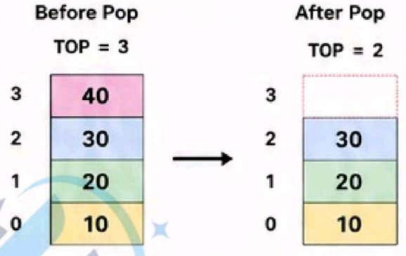
ALGORITHM (POP)

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

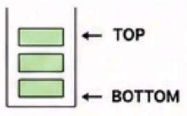
- Stack follows LIFO (Last In First Out) principle.
- All operations (except traversal) are done at TOP only.
- Overflow: Occurs when trying to push in a full stack.
- Underflow: Occurs when trying to pop from an empty stack.
- Traversal does not remove elements; it only visits them.

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

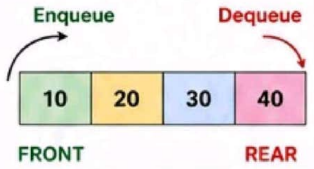


QUEUE

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**.

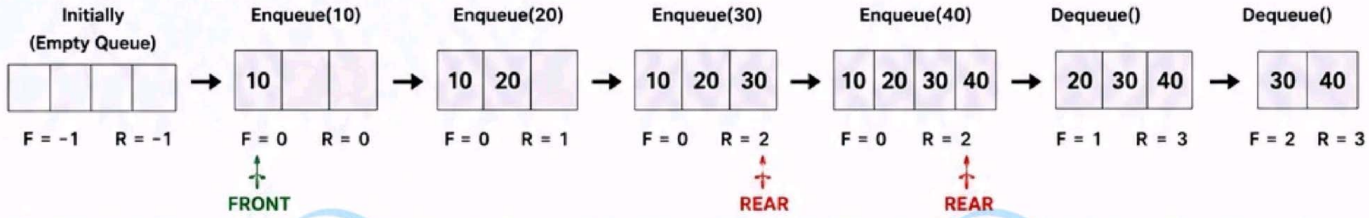
2. CONCEPTS

- 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().

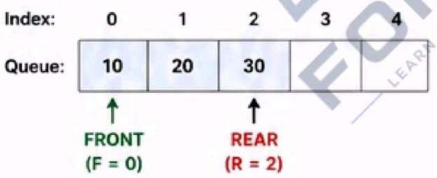


4. MEMORY REPRESENTATIONS OF QUEUE

4.1 QUEUE USING ARRAY (LINEAR QUEUE)

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

Representation

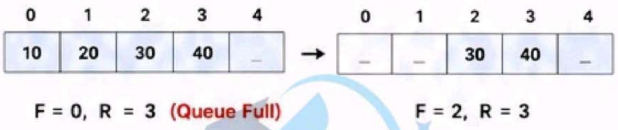


Notes

- 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

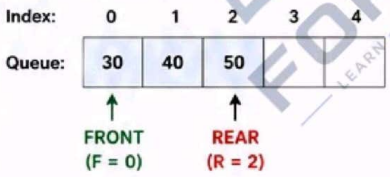
Array size = 5
Insert 10, 20, 30, 40



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.

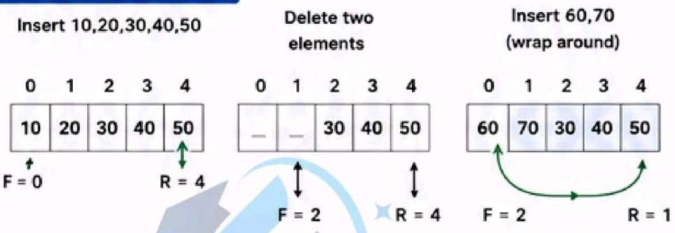
Representation



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)



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)
- Real-world ticket counters, customer lines, etc.

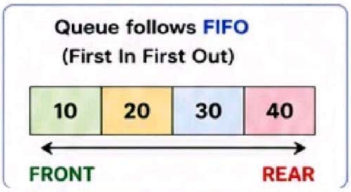
7. SUMMARY TABLE

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



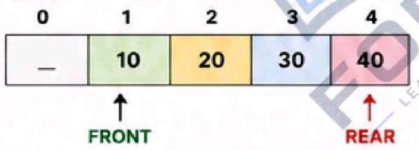
A queue is a linear data structure that follows **FIFO** (First In First Out) principle.
Insertion is done at **REAR** and deletion is done from **FRONT**.

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)



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)

Enqueue(40)

After Insertion (F = 1, R = 4)

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].

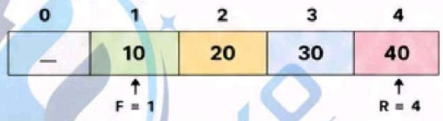
Pseudocode

```
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

After Enqueue(40)



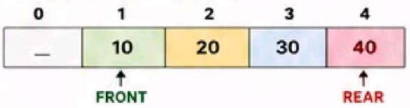
REAR is updated and 40 is inserted.
(Queue size increases by 1)

3 DELETION (DEQUEUE)

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

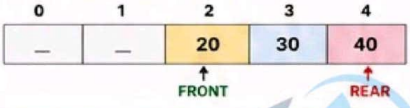
Example

Initial Queue (F = 1, R = 4)



Dequeue()

After Deletion (F = 2, R = 4)



Algorithm (Dequeue)

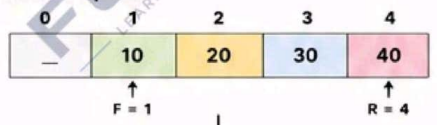
1. If queue is empty, report UNDERFLOW.
2. Store and remove element at FRONT.
3. If F = R (only one element was present)
   Set F = R = -1 (Queue becomes empty)
4. Else
   Set F = F + 1

Pseudocode

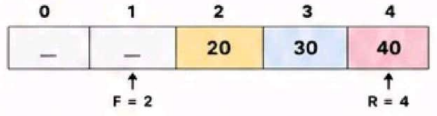
```
if (queue is empty)
    print "Queue Underflow"
else
   removedElement = Q[F]
  if (F == R)
     F = R = -1      // Queue becomes empty
  else
     F = F + 1
   return removedElement
```

Illustration (Dequeue)

Before Dequeue



After Dequeue()

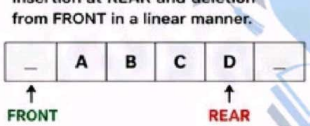


Element 10 is removed.
FRONT is updated. (Queue size decreases by 1)

4 TYPES OF QUEUE

1. LINEAR QUEUE

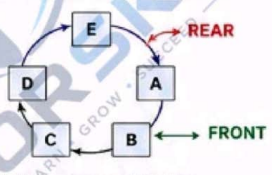
Insertion at REAR and deletion from FRONT in a linear manner.



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

2. CIRCULAR QUEUE

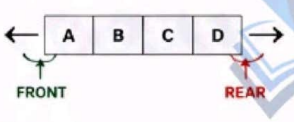
Last position is connected to the first position to form a circle.



- Efficient memory utilization
- No space wastage

3. DEQUE (Double Ended Queue)

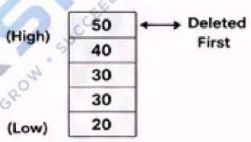
Insertion and deletion are allowed at both FRONT and REAR.



- 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

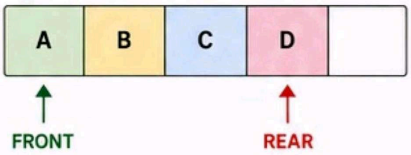
Common Terms

- 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)

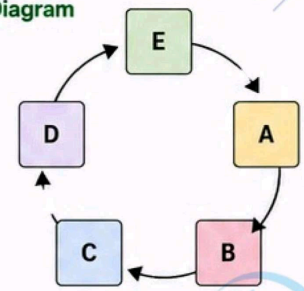
Applications

- 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

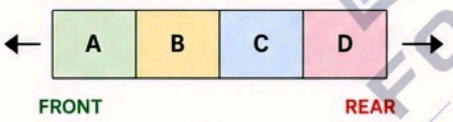
Applications

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

Features

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

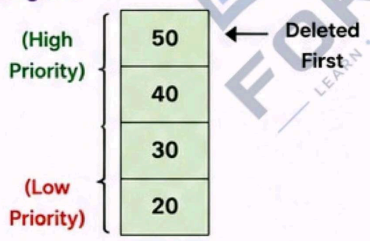
Applications

- 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.