Distinguish between Linear and Non-Linear Data Structures based on parameters

Parameter	Linear Data Structure	Non-Linear Data Structure
Definition	Elements are arranged sequentially (one after another).	Elements are arranged hierarchically or in interconnected manner.
Examples	Array, Linked List, Stack, Queue	Tree, Graph
Memory utilization	Memory may not be used efficiently (continuous allocation in arrays).	Memory is used efficiently with dynamic allocation (like trees/graphs).
Traversal	Traversed in a single run (e.g., from 1st element to last).	Traversal is complex (may need DFS, BFS).
Implementation	Easy to implement.	Difficult to implement.
Complexity	Time complexity is generally lower for simple operations.	Time complexity can be higher depending on structure.

Q2: Write the time complexities of the given code and justify the same.

Example 1:

```c
for(i=0; i<n; i++) {
   printf("Hello");
}
```

- Loop runs **n times** → **Time Complexity = O(n)**

Example 2:

```c
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
      printf("Hello");
   }
}
```

- Nested loops → **n × n = n^2** steps → **Time Complexity = $O(n^2)$**

Example 3:

```c
for(i=0; i<n; i++) {
   for(j=0; j<n; j++) {
      for(k=0; k<n; k++) {
         printf("Hello");
      }
   }
}
```

- 3 nested loops → **n × n × n = n^3** → **Time Complexity = $O(n^3)$**

Example 4:

```c
printf("Hello");
```

- Executes once → **Time Complexity = O(1)**

Justification Rule:

- **1 loop → O(n)**
- **2 nested loops → $O(n^2)$**
- **3 nested loops → $O(n^3)$**
- **No loop → O(1)**

Q3: Explain the following algorithm time complexities with appropriate example

i) O(1) – Constant Time

- Execution time does not depend on input size.
- Example:

```c
x = a + b;    // runs once
```

⏳ **Time Complexity = O(1)**

---

ii) O(n) – Linear Time

- Time grows directly with input size.
- Example:

```c
for(i=0; i<n; i++) {
   printf("Hello");
}
```

- Runs **n times → O(n)**

Q4: Explain space complexity with example

Answer:

Space Complexity of an algorithm is the total amount of memory required by the program to execute successfully. It includes memory for:

1. **Fixed part** – Independent of input size. (e.g., constants, program instructions, simple variables).
2. **Variable part** – Depends on input size. (e.g., dynamic memory allocation, recursion stack, arrays).

So, **Space Complexity = Fixed Part + Variable Part.**

Example 1:

```c
int a, b, c;       // requires constant space
for(i=0; i<n; i++) {
   printf("%d", i);
}
```

- Only fixed variables `a, b, c, i` are used.
- Memory does not depend on `n`.
- **Space Complexity = O(1)**

Example 2 (Array case):

```c
int arr[n];
for(i=0; i<n; i++) {
   arr[i] = i;
}
```

- Here an array of size **n** is created.
- Memory grows as `n`.
- **Space Complexity = O(n)**

Example 3 (Recursion):

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

- Each recursive call needs memory on stack.
- For `n` calls → stack memory is proportional to `n`.
- **Space Complexity = O(n)**

Q5: Write algorithm for the given task (Refer Algm examples given in teams)

General Algorithm Format

1. **Start**
2. Read the input(s)
3. Perform the required steps (calculation / condition / loop)
4. Display the result
5. **Stop**

Example 1: Algorithm to find the largest of 3 numbers

Algorithm: Largest_of_Three

1. Start
2. Input numbers A, B, C
3. If A > B and A > C, then Largest = A
4. Else if B > A and B > C, then Largest = B
5. Else Largest = C
6. Print Largest
7. Stop

Q6: Distinguish between Bottom-up and Top-down approach of algorithm design with example

1. Top-down Approach

1. Problem is divided into smaller **sub-problems** step by step.
2. Start with the **main function** and break into modules.
3. Focus is on the **overall system design first**, details later.
4. Common in **structured programming** (e.g., C).

Example: In a Student Management System → Start with "Main System" → divide into **Admission, Exam, Fee** modules → further divide into sub-modules.

---

2. Bottom-up Approach

1. Start solving **small sub-problems first**, then integrate them.
2. Begin with designing **modules or classes** first.
3. Focus is on **building reusable components** and then forming the system.
4. Common in **object-oriented programming** (e.g., C++, Java).

Example: In the same Student Management System → First design **Student class, Subject class, Fee class** → integrate into **Exam and Admission** modules → combine into full system.

Q7: Classify the type of algorithms

Answer:

Algorithms can be classified into several types based on their design strategy. The major categories are:

---

1. Brute Force Algorithm

- Simplest type, tries all possible solutions.
- Example: Linear Search (check each element one by one).

---

2. Divide and Conquer Algorithm

- Break problem into smaller sub-problems, solve them, and combine the results.
- Example: Binary Search, Merge Sort, Quick Sort.

---

3. Greedy Algorithm

- Makes the best choice at each step with the hope of finding global optimum.
- Example: Dijkstra's Algorithm for shortest path.

---

4. Dynamic Programming (DP)

- Breaks a problem into overlapping sub-problems and stores their solutions to avoid recomputation.
- Example: Fibonacci Series using DP, Matrix Chain Multiplication.

---

5. Backtracking Algorithm

- Tries a solution, if it fails, goes back and tries another path.
- Example: N-Queens Problem, Maze Solving.

---

6. Randomized Algorithm

- Uses random numbers in logic to make the algorithm simpler/faster.
- Example: Randomized Quick Sort.

Q8: Explain the following notations with example

---

i) Big O Notation (O-notation)

- Represents the **upper bound** of time complexity.
- Describes the **worst-case** performance of an algorithm.
- Tells us the maximum time an algorithm will take.

Example:

In linear search, if array size is `n`, in worst case we may need to check all elements → **O(n)**.

---

ii) Omega Notation (Ω-notation)

- Represents the **lower bound** of time complexity.
- Describes the **best-case** performance of an algorithm.
- Tells us the minimum time the algorithm will take.

Example:

In linear search, if the element is at the first position, only 1 comparison → **Ω(1)**.

---

iii) Theta Notation (Θ-notation)

- Represents the **tight bound** of time complexity.
- Describes the **average-case** performance.
- Gives a range (both upper and lower bound).

Example:

In linear search, on average the element will be found in middle of array → **Θ(n/2) = Θ(n)**.

9. Classify the type of loops .

1. Entry-Controlled Loops

The loop condition is checked **before** entering the loop body.

- **Examples:** `for` loop, `while` loop
- **Characteristics:**
  - The loop may **not execute even once** if the condition is false.
  - Used when the number of iterations is known or depends on a condition.

Example in C:

c Copy Edit

```c
int i = 1;
while(i <= 5) {
    printf("%d ", i);
    i++;
}
```

---

2. Exit-Controlled Loops

The loop condition is checked **after** executing the loop body.

- **Example:** `do-while` loop
- **Characteristics:**
  - The loop **executes at least once**, regardless of the condition.
  - Used when the loop body must execute at least once.

Example in C:

c Copy Edit

```c
int i = 1;
do {
    printf("%d ", i);
    i++;
} while(i <= 5);
```

Q10: Write an algorithm to add/remove an element from an array (demonstrate steps with the given array)

Algorithm to Insert (Add) an element into an array

Algorithm: Insert_Array

1. Start
2. Input array `A` with size `n`
3. Input `pos` (position where new element is to be inserted)
4. Input `item` (new element to insert)
5. For `i = n-1` down to `pos`, shift element right → `A[i+1] = A[i]`
6. Place new element → `A[pos] = item`
7. Increase size → `n = n+1`
8. Print updated array
9. Stop

Example:

Array = [10, 20, 30, 40, 50], `n=5`

Insert 25 at position 2

Result → [10, 20, **25**, 30, 40, 50]

Algorithm to Delete (Remove) an element from an array

Algorithm: Delete_Array

1. Start
2. Input array `A` with size `n`
3. Input `pos` (position of element to be deleted)
4. Store `A[pos]` in `item`
5. For `i = pos` to `n-1`, shift element left → `A[i] = A[i+1]`
6. Decrease size → `n = n-1`
7. Print updated array
8. Stop

Example:

Array = [10, 20, 30, 40, 50], `n=5`

Delete element at position 3

Result → [10, 20, 40, 50]

1. Selection Sort

Algorithm:

1. Start
2. Repeat for `i = 0 to n-2`
   - Find the index of the minimum element from `i` to `n-1`.
   - Swap the minimum element with `arr[i]`.
3. Stop

Time Complexity:

- Best Case: **O(n²)**
- Average Case: **O(n²)**
- Worst Case: **O(n²)**

Space Complexity:

- **O(1)** (in-place, no extra memory)

---

2. Bubble Sort

Algorithm:

1. Start
2. Repeat for `i = 0 to n-1`
   - For `j = 0 to n-i-2`
     - If `arr[j] > arr[j+1]`, swap them.
3. Stop

Time Complexity:

- Best Case: **O(n)** (when array already sorted, only 1 pass)
- Average Case: **O(n²)**
- Worst Case: **O(n²)**

Space Complexity:

- **O(1)**

3. Insertion Sort

Algorithm:

1. Start
2. For `i = 1 to n-1`
   - Take `key = arr[i]`.
   - Compare `key` with previous elements and shift them right until correct position is found.
   - Insert `key` at the correct place.
3. Stop

Time Complexity:

- Best Case: **O(n)** (already sorted array)
- Average Case: **O(n²)**
- Worst Case: **O(n²)**

Space Complexity:

- **O(1)**

---

4. Linear Search

Algorithm:

1. Start
2. For `i = 0 to n-1`
   - If `arr[i] == key`, return position.
3. If not found, return -1.
4. Stop

Time Complexity:

- Best Case: **O(1)** (first element is the key)
- Average Case: **O(n/2) → O(n)**
- Worst Case: **O(n)**

Space Complexity:

- **O(1)**

5. Binary Search (works on sorted array)

Algorithm:

1. Start
2. Set `low = 0`, `high = n-1`.
3. While `low <= high`:
   - `mid = (low + high) / 2`
   - If `arr[mid] == key`, return mid.
   - If `key < arr[mid]`, set `high = mid - 1`.
   - Else, set `low = mid + 1`.
4. If not found, return -1.
5. Stop

Time Complexity:

- Best Case: **O(1)**
- Average Case: **O(log n)**
- Worst Case: **O(log n)**

Space Complexity:

- Iterative: **O(1)**
- Recursive: **O(log n)** (due to recursion stack)

6. Quick Sort

Algorithm:

1. Start
2. If `low < high`:
   - Partition the array around a pivot (elements smaller on left, larger on right).
   - Recursively apply QuickSort on left and right subarrays.
3. Stop

Time Complexity:

- Best Case: **O(n log n)**
- Average Case: **O(n log n)**
- Worst Case: **O(n²)** (when pivot is always smallest/largest element)

Space Complexity:

- **O(log n)** (recursive stack)

i) Selection Sort Example

Array: `[29, 10, 14, 37, 13]`

- Pass 1: Find minimum in entire array → `10`. Swap with `29`.
  → `[10, 29, 14, 37, 13]`
- Pass 2: Find minimum in remaining → `13`. Swap with `29`.
  → `[10, 13, 14, 37, 29]`
- Pass 3: Minimum in `[14, 37, 29]` is `14`. No swap.
  → `[10, 13, 14, 37, 29]`
- Pass 4: Minimum in `[37, 29]` is `29`. Swap with `37`.
  → `[10, 13, 14, 29, 37]`

✅ Sorted array: `[10, 13, 14, 29, 37]`

---

ii) Bubble Sort Example

Array: `[29, 10, 14, 37, 13]`

- Pass 1: Compare adjacent elements
  `[10, 29, 14, 13, 37]`
- Pass 2:
  `[10, 14, 13, 29, 37]`
- Pass 3:
  `[10, 13, 14, 29, 37]`
- Pass 4: No swap → stop.

✅ Sorted array: `[10, 13, 14, 29, 37]`

---

iii) Insertion Sort Example

Array: `[29, 10, 14, 37, 13]`

- Pass 1: Take `10`, insert in sorted part `[29]` → `[10, 29, 14, 37, 13]`
- Pass 2: Take `14`, insert in `[10, 29]` → `[10, 14, 29, 37, 13]`
- Pass 3: Take `37`, insert in `[10, 14, 29]` → `[10, 14, 29, 37, 13]`
- Pass 4: Take `13`, insert in `[10, 14, 29, 37]` → `[10, 13, 14, 29, 37]`

✅ Sorted array: `[10, 13, 14, 29, 37]`

---

iv) Linear Search Example

Array: `[5, 10, 15, 20, 25, 30]`, Search Key = `20`

Steps:

- Compare `5` with `20` → Not found
- Compare `10` with `20` → Not found
- Compare `15` with `20` → Not found
- Compare `20` with `20` → Found at index 3

✅ Result: Element found at position 4 (0-based index = 3)

---

v) Binary Search Example

Array (sorted): `[5, 10, 15, 20, 25, 30]`, Search Key = `20`

Steps:

- `low=0, high=5 → mid=2 → arr[2]=15` < `20` → search right
- `low=3, high=5 → mid=4 → arr[4]=25` > `20` → search left
- `low=3, high=3 → mid=3 → arr[3]=20` = Key found

✅ Result: Element found at index 3

---

vi) Quick Sort Example

Array: `[29, 10, 14, 37, 13]`

- Choose last element (13) as pivot.
  Partition: `[10]` `[13]` `[29, 14, 37]`
- Left side `[10]` → already sorted.
- Right side `[29, 14, 37]`, pivot = `37`.
  Partition: `[29, 14]` `[37]`
- Sort `[29, 14]`, pivot = `14`.
  Partition: `[14]` `[29]`
- Combine: `[10, 13, 14, 29, 37]`

✅ Sorted array: `[10, 13, 14, 29, 37]`

Q.3) Comparison of Selection Sort, Bubble Sort, Insertion Sort and Quick Sort.

Answer:

The following table shows the comparison of Selection Sort, Bubble Sort, Insertion Sort and Quick Sort:

Selection Sort	Bubble Sort	Insertion Sort	Quick Sort
Selects the minimum element and places it at correct position.	Repeatedly swaps adjacent elements if in wrong order.	Inserts each element into its correct position in a sorted sub-array.	Uses divide and conquer; partitions array around a pivot.
Best case: **$O(n^2)$**	Best case: **O(n)** (optimized)	Best case: **O(n)**	Best case: **O(n log n)**
Average case: **$O(n^2)$**	Average case: **$O(n^2)$**	Average case: **$O(n^2)$**	Average case: **O(n log n)**
Worst case: **$O(n^2)$**	Worst case: **$O(n^2)$**	Worst case: **$O(n^2)$**	Worst case: **$O(n^2)$**
Space: **O(1)**	Space: **O(1)**	Space: **O(1)**	Space: **O(log n)** (recursion)
Not stable	Stable	Stable	Not stable

Q.4) Comparison of Linear Search and Binary Search

Answer:

The following table shows the comparison between Linear Search and Binary Search:

Linear Search	Binary Search
Searches each element of the list one by one until the target is found.	Repeatedly divides the sorted list into two halves to search for the target.
Can be applied on **unsorted or sorted** data.	Can be applied only on **sorted** data.
Best case: **O(1)** (if element found at first position).	Best case: **O(1)** (if element found at middle).
Average/Worst case: **O(n)**	Average/Worst case: **O(log n)**
Simple to implement.	More efficient but requires extra logic.
Suitable for **small datasets.**	Suitable for **large datasets.**

Q.1) Algorithm for push(), pop(), peek(), isempty() on Stack with Example Diagram

Answer:

A **stack** is a linear data structure which follows **LIFO (Last In, First Out)** principle.

The basic operations on stack are **push, pop, peek, isempty**.

---

1) Algorithm for PUSH(x)

vbnet Copy Edit

```
Step 1: If TOP = MAX-1 then
            Print "Overflow"
            Exit
Step 2: TOP = TOP + 1
Step 3: STACK[TOP] = x
Step 4: Exit
```

2) Algorithm for POP()

vbnet Copy Edit

```
Step 1: If TOP = -1 then
            Print "Underflow"
            Exit
Step 2: ITEM = STACK[TOP]
Step 3: TOP = TOP - 1
Step 4: Return ITEM
Step 5: Exit
```

3) Algorithm for PEEK()

vbnet Copy Edit

```
Step 1: If TOP = -1 then
            Print "Stack is Empty"
            Exit
Step 2: Return STACK[TOP]
Step 3: Exit
```

4) Algorithm for ISEMPTY()

vbnet Copy Edit

```
Step 1: If TOP = -1 then
            Return TRUE
        Else
            Return FALSE
Step 2: Exit
```

Q.2) Time Complexities of Operations Performed on Stack Data Structure

Answer:

The common operations on a stack and their time complexities are:

Operation	Description	Time Complexity
PUSH(x)	Insert an element on top of the stack	**O(1)**
POP()	Remove the top element from the stack	**O(1)**
PEEK()/TOP()	Return the top element without removing it	**O(1)**
ISEMPTY()	Check if the stack is empty	**O(1)**
ISFULL() (for array implementation)	Check if the stack is full	**O(1)**

Q.3) Explain Stack as ADT using Array

Answer:

A **Stack** is an **Abstract Data Type (ADT)** which stores elements in a **linear order** but follows the rule of **LIFO (Last In, First Out)**.
In stack, insertion and deletion are done only from one end called **TOP**.

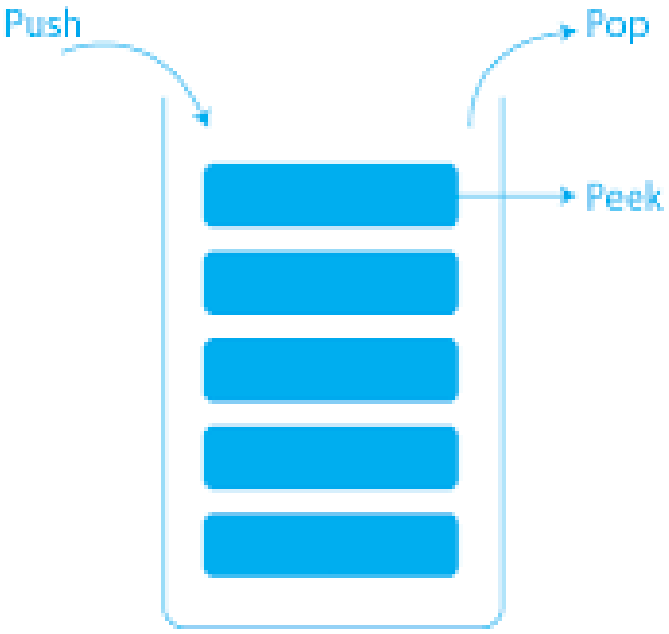
When implemented using **array**, a stack requires:

- An array `STACK[MAX]` to hold elements
- An integer variable `TOP` to indicate the index of the top element (initially `-1`)

Basic Operations on Stack (Array Implementation):

1. **PUSH(x):**
   - Add element `x` at the top of stack.
   - If `TOP = MAX-1`, stack is **overflow**.
   - Else, increment `TOP` and assign `STACK[TOP] = x`.
2. **POP():**
   - Remove and return element at `TOP`.
   - If `TOP = -1`, stack is **underflow**.
   - Else, return `STACK[TOP]` and decrement `TOP`.
3. **PEEK():**
   - Return element at `STACK[TOP]` without deleting it.
4. **ISEMPTY():**
   - Returns **true** if `TOP = -1`, else **false**.
5. **ISFULL():**
   - Returns **true** if `TOP = MAX-1`, else **false**.

Diagram:-



Q.4) Memory Allocation to Different Function Calls in Recursion and Recursion in Stack with Example and Diagram

Answer:

🔹 Recursion and Memory Allocation

- **Recursion** is a programming technique where a function calls itself.
- During recursion, each **function call** is assigned a separate **activation record (stack frame)** in memory.
- These activation records are stored in the **runtime stack.**
- Each record stores:
  - Function parameters
  - Local variables
  - Return address (where to return after function finishes)

When a recursive function is called:

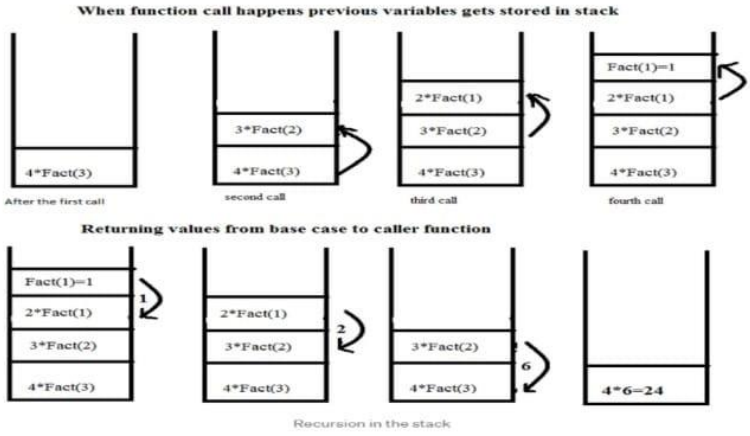
1. A new stack frame is created and pushed onto the stack.
2. When the function completes, its frame is popped out.
3. This continues until the base condition is reached.

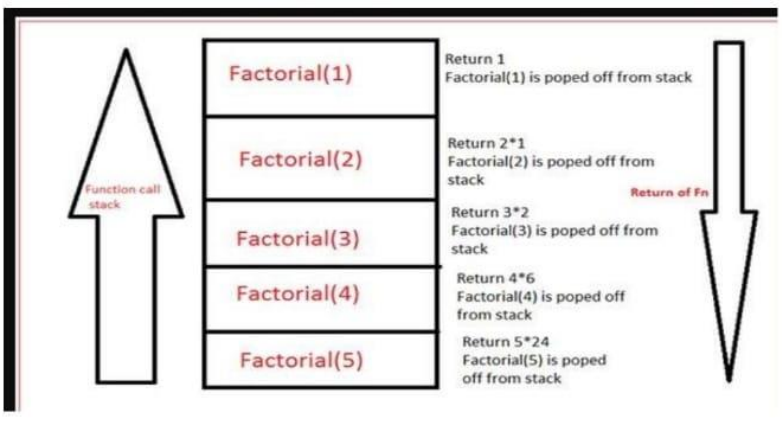
🔹 Recursion in Stack (Example: Factorial)

Function:

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

Call: `fact(3)`





Q.5) Polish Notations with Example

Answer:

In arithmetic expressions, operators can be written in different positions relative to operands. These are called **Polish Notations.**

🔹 **1) Infix Notation**

- Operator is written **between operands.**
- This is the standard form we use in mathematics.
- Example:

```
css                                                    Copy  Edit

(A + B) * C
```

🔹 **2) Prefix Notation (Polish Notation)**

- Operator is written **before operands.**
- Also called **Polish Notation.**
- No need of brackets, order is determined by operator position.
- Example:

```
css                                                    Copy  Edit

* + A B C
```

(Equivalent to (A + B) * C)

🔹 **3) Postfix Notation (Reverse Polish Notation)**

- Operator is written **after operands.**
- Also called **Reverse Polish Notation (RPN).**
- No brackets needed, evaluation is done using stack.
- Example:

```
css                                                    Copy  Edit

A B + C *
```

(Equivalent to (A + B) * C)

🔹 **Example Conversion**

For the expression:

```
css                                                    Copy  Edit

(A + B) * (C - D)
```

- **Infix:** `(A + B) * (C - D)`
- **Prefix (Polish):** `* + A B - C D`
- **Postfix (Reverse Polish):** `A B + C D - *`

Q.6) Algorithm for Evaluation of Postfix Expression using Stack

Answer:

A **postfix expression** (Reverse Polish Notation) is evaluated using a **stack.**
We scan the expression from **left to right:**

- If the symbol is **operand**, push it onto the stack.
- If the symbol is **operator**, pop the required number of operands from stack, perform the operation, and push the result back.
- Final result is at the top of stack.

🔹 **Algorithm:**

vbnet Copy Edit

```
Step 1: Initialize an empty stack
Step 2: Scan postfix expression from left to right
Step 3: If symbol is operand
           Push it onto stack
Step 4: If symbol is operator
           Pop two operands from stack
           Apply operator: result = operand2 operator operand1
           Push result back onto stack
Step 5: Repeat steps 2-4 until expression ends
Step 6: Final result = Pop element from stack
Step 7: Exit
```

Q.7) Demonstrate Steps of Evaluation of Postfix Expression using Stack with Example and Diagram

Answer:

- **Example Expression:**

Copy Edit

```
6 2 3 + - 3 8 2 / + * 2 ^
```

- **Step-by-Step Evaluation**

Step	Symbol Read	Action Performed	Stack Status
1	6	Push 6	[6]
2	2	Push 2	[6, 2]
3	3	Push 3	[6, 2, 3]
4	+	2 + 3 = 5 → Push	[6, 5]
5	-	6 - 5 = 1 → Push	[1]
6	3	Push 3	[1, 3]
7	8	Push 8	[1, 3, 8]
8	2	Push 2	[1, 3, 8, 2]
9	/	8 / 2 = 4 → Push	[1, 3, 4]
10	+	3 + 4 = 7 → Push	[1, 7]
11	*	1 * 7 = 7 → Push	[7]
12	2	Push 2	[7, 2]
13	^	7 ^ 2 = 49 → Push	[49]

Final Result = 49

5, 6, 2, +, *, 12, 4, /, -,
(1) (2) (3) (4) (5) (6) (7) (8) (9)

Symbol Scanned		STACK		
(1)	5	5		
(2)	6	5,	6	
(3)	2	5,	6,	2
(4)	+	5,	8	
(5)	*	40		
(6)	12	40,	12	
(7)	4	40,	12,	4
(8)	/	40,	3	
(9)	-	37		

Q.8) Define Queue. Describe Array Representation of Queue with Diagram

Answer:

🔹 Definition of Queue

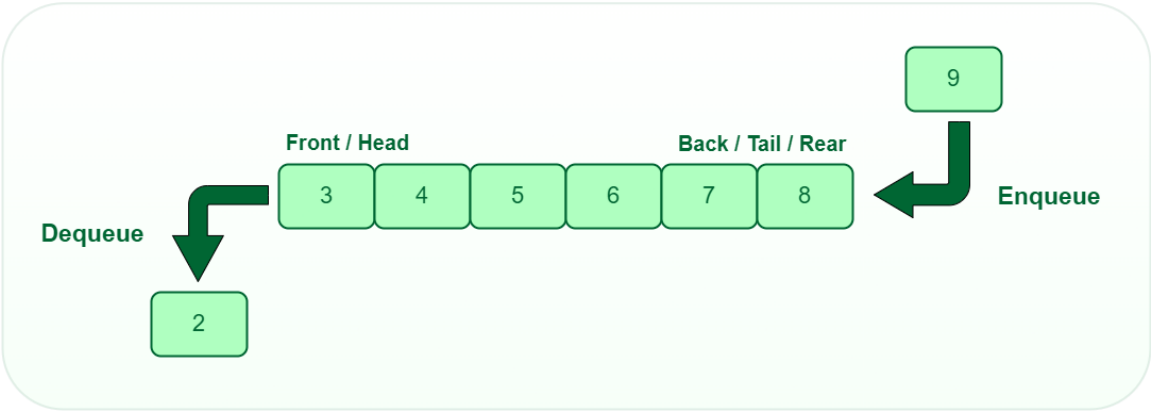
- A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.
- The element inserted first is deleted first, just like a real-life queue.
- Insertion is done at the **rear end**, and deletion is done from the **front end**.

🔹 Array Representation of Queue

- A queue can be implemented using an **array.**
- Two variables are maintained:
  - `FRONT` → points to the first element of the queue
  - `REAR` → points to the last element of the queue
- Initially: `FRONT = -1`, `REAR = -1` (queue empty).

🔹 Operations on Queue (Array Implementation)

1. **ENQUEUE(x) (Insertion):**
   - If `REAR = MAX-1`, then queue is **overflow.**
   - Else if queue empty (`FRONT = -1`), set `FRONT = 0` and `REAR = 0`.
   - Else increment `REAR` and insert element at `QUEUE[REAR]`.
2. **DEQUEUE() (Deletion):**
   - If `FRONT = -1` or `FRONT > REAR`, then queue is **underflow.**
   - Else return element at `QUEUE[FRONT]` and increment `FRONT`.
3. **ISEMPTY():**
   - True if `FRONT = -1` or `FRONT > REAR`.
4. **ISFULL():**
   - True if `REAR = MAX-1`.



To understand

🔹 Operations on Queue (Array Implementation) – Simple Explanation

1. **ENQUEUE(x) → Insert into queue**
   - If the queue is already full (`REAR = MAX-1`), no more elements can be added → **Overflow.**
   - If the queue is empty (`FRONT = -1`), then set both `FRONT = 0` and `REAR = 0`, and put the element there.
   - Otherwise, move `REAR` one step forward and place the element at that position.

👉 Think of it like standing in a line: new person always goes at the **rear end.**

---

2. **DEQUEUE() → Remove from queue**
   - If the queue is empty (`FRONT = -1` or `FRONT > REAR`), then nothing to remove → **Underflow.**
   - Otherwise, remove the element from the `FRONT` position and move `FRONT` one step forward.

👉 Just like in a line: the **first person** in the queue goes out first.

---

3. **ISEMPTY() → Check if queue has no elements**
   - Queue is empty if `FRONT = -1` (never used) or if `FRONT` has crossed `REAR` (all elements removed).

👉 Means: "There is **no one in the line.**"

---

4. **ISFULL() → Check if queue is full**
   - Queue is full if `REAR = MAX-1` (last index of the array is occupied).

👉 Means: "The **line has reached maximum capacity**, can't add more people."

Q.9) Classify Types of Queues

Answer:

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle.
Based on functionality, queues are classified into the following types:

🔹 **1) Simple Queue (Linear Queue)**

- Also called **basic queue**.
- Insertion happens at **REAR**, deletion happens at **FRONT**.
- Example: Ticket counter line.

🔹 **2) Circular Queue**

- The last position is connected back to the first position, forming a circle.
- Solves the problem of unused space in simple queue.
- Example: CPU scheduling in Operating System.

🔹 **3) Double Ended Queue (Deque)**

- Elements can be inserted and deleted from **both ends** (front and rear).
- Types:
  - **Input-restricted deque** → insertion only at rear, deletion from both ends.
  - **Output-restricted deque** → deletion only at front, insertion from both ends.

🔹 **4) Priority Queue**

- Each element has a **priority**.
- Element with higher priority is served before lower priority, even if it comes later.
- Example: Emergency patients in hospital queue.

🔹 **5) Circular Deque (Hybrid)** ***(sometimes considered separately)***

- Combines the properties of **circular queue** and **deque**.

Q.10) Algorithm to Insert and Delete an Element in a Queue and Explain with Example

Answer:

A **Queue** is a linear data structure following **FIFO (First In, First Out)**.
Insertion is done at **REAR** (ENQUEUE) and deletion at **FRONT** (DEQUEUE).

🔹 1) Algorithm for ENQUEUE (Insertion)

vbnet Copy Edit

```
Step 1: If REAR = MAX-1 then
           Print "Queue Overflow"
           Exit
Step 2: If FRONT = -1 then
           FRONT = 0
Step 3: REAR = REAR + 1
Step 4: QUEUE[REAR] = x
Step 5: Exit
```

Explanation:

- If the queue is full → cannot insert → overflow.
- If queue is empty → initialize FRONT.
- Increment REAR and insert element at QUEUE[REAR].

🔹 2) Algorithm for DEQUEUE (Deletion)

vbnet Copy Edit

```
Step 1: If FRONT = -1 or FRONT > REAR then
           Print "Queue Underflow"
           Exit
Step 2: ITEM = QUEUE[FRONT]
Step 3: FRONT = FRONT + 1
Step 4: Return ITEM
Step 5: Exit
```

Explanation:

- If queue is empty → cannot delete → underflow.
- Otherwise, remove element at FRONT and increment FRONT.

Initial Queue (empty):

makefile

```
QUEUE: [ -  -  - ]
FRONT = -1, REAR = -1
```

ENQUEUE(10) →

makefile

```
QUEUE: [10 - -], FRONT=0, REAR=0
```

ENQUEUE(20) →

makefile

```
QUEUE: [10 20 -], FRONT=0, REAR=1
```

DEQUEUE() → removes 10

makefile

```
QUEUE: [- 20 -], FRONT=1, REAR=1
```

Final Queue: `[20]` ✅

Q.11) Algorithm to Convert Infix Expression to Postfix Expression

Answer:

Postfix Expression (Reverse Polish Notation):

- Operator comes **after operands**.
- Example: Infix → `(A + B) * C`, Postfix → `A B + C *`

We use a stack to hold operators during conversion.

🔹 Algorithm (Infix → Postfix using Stack)

vbnet Copy Edit

```
Step 1: Initialize an empty stack for operators
Step 2: Scan the infix expression from left to right
Step 3: For each symbol:
        a) If symbol is an operand → Add it to postfix expression
        b) If symbol is '(' → Push it onto the stack
        c) If symbol is ')' → Pop from stack and add to postfix until '(' is found, then pop '('
        d) If symbol is an operator (+, -, *, /, ^):
             While stack is not empty and precedence of stack top >= precedence of symbol:
                 Pop from stack and add to postfix
             Push the symbol onto stack
Step 4: After scanning the expression, pop all remaining operators from stack and add to postfix
Step 5: Exit
```

Q.13) Advantage of Circular Queue over Linear Queue

Answer:

A **Circular Queue** is an improvement over a **Linear Queue**, where the last position is connected back to the first to form a circle.

- **Advantages:**

1. **Efficient Use of Space:**
   - In a linear queue, after some dequeues, the front moves forward and free space at the beginning cannot be reused.
   - Circular queue **reuses empty spaces**, so memory is utilized efficiently.
2. **No "False Overflow":**
   - Linear queue may show **overflow** even when there is free space in the array.
   - Circular queue prevents this by wrapping around.
3. **Better for Continuous Processes:**
   - Ideal for **CPU scheduling, buffer management**, and **real-time systems** where queue runs continuously.
4. **Fixed Size Handling:**
   - Works well with **fixed-size arrays**, avoiding shifting elements after every dequeue.