Data Structures for AI | Prev. Year BPUT Q&A

MODULE 2

Abstract Data Structures, STACK and QUEUE

Sl.	Questions and Answers	Mark	Year
1	**What is data structure? Give an example each of Linear & Non-linear data structure.**	2	2023

- A data structure is a way to organize, manage, and store data in a computer so it can be accessed and modified efficiently.
- It defines the arrangement of data in memory and supports operations like insertion, deletion, searching, and sorting.
- Data structures are significant for optimizing algorithm performance and to effectively manage data in various applications.
- Examples of **linear** data structure : **arrays**, linked lists
- Examples of **non-linear** data structure : **trees**, graphs.

2	**What is meant by Abstract data type (ADT) ? Differentiate between linear and non-linear data structure.**	6	2021

Abstract Data Structure :

- An Abstract Data Structure (or Abstract Data Type, ADT) is a way of defining a data structure purely in terms of its operations and properties, without specifying how it is implemented.
- It is conceptual (no codes involved). For example, a Stack defines operations like push, pop, and peek but doesn't specify how these operations are implemented.
- Examples of ADT are Stack, Queue, List, Set and Map.

Linear data structure :

- In a linear data structure, data elements are arranged in a sequential or linear order, where each element is connected to its previous and next element.
- The data elements are stored in a contiguous block of memory.
- Linear data structures allow for easy traversal of data elements in a single run, either forward or backward.
- Examples:

  [1] **Array**: A collection of elements identified by index, stored in a contiguous block of memory.

  [2] **Linked List**: A sequence of nodes where each node contains data and a reference (or link) to the next node in the sequence.

  [3] **Stack**: A collection of elements with a Last-In-First-Out (LIFO) access pattern.

  [4] **Queue**: A collection of elements with a First-In-First-Out (FIFO) access pattern.

- In linear data structures like arrays, memory is often allocated in a contiguous block, which leads to inefficient memory usage if the size of the structure changes.
- The time complexity of operations such as insertion, deletion and searching typically depend on the position of the elements within the structure (e.g., O(n) for searching in a linked list).

Non-Linear data structure :

- In a non-linear data structure, data elements are arranged in a hierarchical or interconnected manner, where each element may be connected to multiple elements.
- Non-linear data structures often require more complex algorithms for traversal, such as depth-first search (DFS) or breadth-first search (BFS).
- Examples:

  [1] **Tree**: A hierarchical structure where each node has a value and references to child nodes. A binary tree, allows each node to have at most two children.

  [2] **Graph**: A collection of nodes (vertices) connected by edges. Graphs can be either directed or undirected and may contain cycles.

  [3] **Heap**: A specialized tree-based data structure that satisfies the heap property, where the parent node is always greater (max-heap) or smaller (min-heap) than the child nodes.

- Memory utilization in non-linear data structures is often more efficient than in linear ones, especially for large datasets, as they don't require a contiguous block of memory.
- Operations like insertion, deletion and searching can be more complex, but they have better average-case performance in certain structures (e.g., O(log n) for searching in a balanced binary search tree).

3	**Differentiate between Abstract and Concrete Data Structure.**	**2**	**2022**

Abstract Data Structure	***Concrete Data Structure***
An Abstract Data Structure (or Abstract Data Type, **ADT**) is a way of defining a data structure purely in terms of its operations and properties, without specifying how it is implemented.	A Concrete Data Structure refers to the actual implementation of an abstract data structure. It specifies the details of how data is organized in memory and how operations on that data are performed.
Examples of ADT: Stack, Queue, List, Set, and Map.	Examples of concrete DS: ▸ Array-based implementation of a Stack ▸ Linked list-based implementation of Queue ▸ Hash table implementation of a Map.
It is conceptual (no codes involved). For example, a Stack defines operations like push, pop, and peek but doesn't specify how these operations are implemented.	A Concrete DS is directly concerned with the details of implementation, including memory management, pointers and algorithmic efficiency.

4	**Which data structures are applied when dealing with a recursive function?**	**2**	**2020**

The following data structures implement recursion.

Stack

- Stack uses the concept of "**last in, first out**", which is useful for recursion.
- When a function calls itself recursively, the stack **stores** a **return address** for each activation of the function. This allows the function to return from each activation.

Tree

- Recursive functions are often used to traverse trees (e.g., binary trees) because the structure of a tree naturally lends itself to recursion, with each node being a sub-tree.

Linked List

- Recursive functions are sometimes used to perform operations on linked lists, like reversing the list or finding an element.

5	**Differentiate between stack and queue**	**2**	**2023**

Feature	***Stack***	***Queue***
Order Principle	Last In, First Out (LIFO)	First In, First Out (FIFO)
Insertion Point	Elements are added at the **top** (push)	Elements are added at the **rear** (enqueue)
Deletion Point	Elements are removed from the **top** (pop)	Elements are removed from the **front** (dequeue)
Access	Only the **top** element is accessible	Only the **front** element is accessible
Use Cases	Function calls, expression evaluation, undo mechanisms, balanced parentheses checking	CPU scheduling, BFS traversal, IO buffering, printer management
Real-World Example	Stack of discs where we remove/add from the top	Queue at a ticket counter, where people leave from the front and join at the back

6	**Write the conditions to test "queue is empty" and "queue is full".**	**2**	**2023**

Suppose the access points of Queue are denoted as "**front**" and "**rear**".

And, the length of Queue is denoted as "**capacity**".

Then, **for linear queue**:

- Condition to test Queue is Empty : **front == –1**
- Condition to test Queue is Full : **rear == capacity – 1**

For circular queue:

- Condition to test Queue is Empty : **front == –1**
- Condition to test Queue is Full : **(rear + 1) % capacity == front**

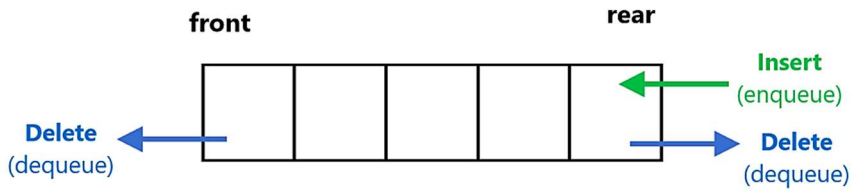
7	**What is deque ? explain its 2 variants.**	**2**	**2023**

- **Deque** = double-ended queue
- This is a linear data structure which allows insertion and deletion of elements from both ends such as front and rear.
- It can be considered as a combination of both stack and queue, providing greater flexibility for accessing elements.

There are two main variants of deque:

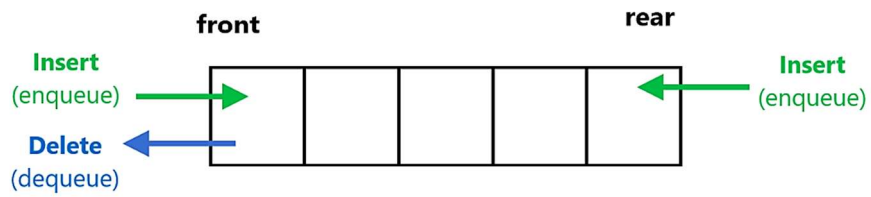
1. Input-Restricted Deque:

- Insertions (***Enqueue***) are allowed only at one end (either front or rear).
- Deletions (***Dequeue***) can be performed at both ends.



2. Output-Restricted Deque:

- Insertions (***Enqueue***) can be performed at both ends (front and rear).
- Deletions (***Dequeue***) are allowed only at one end (either front or rear).



Applications of Deque:

- Palindrome Checking
- Sliding Window Problems
- Undo/Redo Mechanisms

Not to be confused with dequeue.
Dequeue = deletion operation of queue.

Basic Operations on Deque:

- Insert at Front : **insertFront()**
- Insert at Rear : **insertRear()**
- Delete from Front : **deleteFront()**
- Delete from Rear : **deleteRear()**
- Check if Deque is Empty : **isEmpty()**
- Check if Deque is Full : **isFull()** – *in the case of a fixed-size deque*

8	**Convert the expression `((A+B)-C)/(D-E))` to equivalent prefix notation.**	**2**	**2023**

First convert the sub-expressions into prefix notation.

A + B => + A B

(A + B) + C => (+ A B) + C => + (+ A B) C => + + A B C

D – E => – D E

Hence, the final expression becomes:

((A + B) – C) / (D – E)) => (+ + A B C) / (– D E)

=> **/ + + A B C – D E** (*Ans*)

9	**Apply the postfix evaluation algorithm to evaluate the expression : 1 5 7 3 - 2 ^ * +**	**6**	**2023**

Step 1 : initialize an empty stack.

Step 2 : Scan the postfix expression from left to right

PUSH operands onto the stack

Apply operators to the operands on the stack

Scanned **1** => PUSH **1** into the stack => Stack : [1]

Scanned **5** => PUSH **5** into the stack => Stack : [1 , 5]

Scanned **7** => PUSH **7** into the stack => Stack : [1 , 5 , 7]

Scanned **3** => PUSH **3** into the stack => Stack : [1 , 5 , 7 , 3]

Scanned **–** => Apply **subtraction** at the top two elements => 7 – 3 = 4

=> POP the top two elements and PUSH the result at the top

=> Stack : [1 , 5 , 4]

Scanned **2** => PUSH **2** into the stack => Stack : [1 , 5 , 4 , 2]

Scanned **^** => Apply **exponent** at the top two elements => 4 ^ 2 = 16

=> POP the top two elements and PUSH the result at the top

=> Stack : [1 , 5 , 16]

Scanned ***** => Apply **multiplication** on the top two elements => 5 ^ 16 = 80

=> POP the top two elements and PUSH the result at the top

=> Stack : [1 , 80]

Scanned **+** => Apply **addition** on the top two elements => 1 + 80 = 81

=> POP the top two elements and PUSH the result at the top

=> Stack : [81]

Therefore, the final result is **81**.

10	**Explain the advantages of circular queue over linear queue.**	**6**	**2023**

A circular queue is an advanced version of a linear queue where the last position is connected back to the first position to make a circle. The main difference is how these queues manage the end of the array when elements are added or removed.

Advantages of Circular Queue Over Linear Queue:

(i) Efficient Use of Space:

- In a linear queue, when elements are dequeued (removed), the front pointer moves forward, but the space left behind cannot be reused. This leads to a condition called "*wastage of space*," and the queue eventually becomes full even if there are empty slots at the beginning.

- A circular queue reuses this empty space by connecting the end of the queue back to the beginning. When rear reaches the end of the array, it can wrap around to fill up the vacant spots at the beginning, making full use of the available space.

(ii) Better performance by avoiding the Need for Shifting Elements:

- In a linear queue, when the front element is dequeued, it might require shifting all remaining elements to maintain continuity (in some implementations). This shifting takes **O(n)** time, making it inefficient.
- A circular queue does not require any shifting because the front and rear pointers simply move in a circular manner, which takes **O(1)** time, making it more efficient for enqueue and dequeue operations.

(iii) Memory Utilization:

- Circular queues allow continuous memory utilization.
- In a linear queue, even if there are empty slots left after dequeuing elements, they cannot be used without shifting, which wastes memory. Circular queues make use of these slots without the need for shifting.

Use-cases where Circular Queues are preferred:

- **Buffer Management**: Circular queues are used in situations where the data needs to be stored temporarily and reused cyclically, such as in circular buffers or ring buffers for buffering streaming data.
- **Scheduling Algorithms**: Circular queues are used in scheduling algorithms like Round Robin in operating systems, where processes are scheduled cyclically.
- **Network and Resource Management**: Managing resources like bandwidth in networks or jobs in print queues can benefit from the space-efficient nature of circular queues.

11	**Write the algorithm to convert an infix expression to postfix expression. using the algorithm, convert the infix expression `(A+B)*C+D/(E+F*G)-H` into a postfix expression.**	**16**	**2023**

To convert an infix expression to a postfix expression, we use the **Shunting Yard Algorithm**, which makes use of a **stack** to handle operators and parentheses.

Step 1 : **Initialize an empty stack** for operators and an empty list for the output.

Step 2 : **Scan** the infix expression from **left** to **right**:

- **Operand**: Add it directly to the output list.
- **Left Parenthesis** `(` : Push it onto the stack.
- **Right Parenthesis** `)` : Pop from the stack to the output list until a left parenthesis is encountered. *Discard the pair of parentheses*.
- **Operator** (+ , – , * , / , ^):

  CASE 1 : STACK IS EMPTY

PUSH the current operator onto the stack

CASE 2 : STACK IS NOT EMPTY AND The operator at TOP has Lower Precedence than the current operator

PUSH the current operator onto the stack

CASE 3 : STACK IS NOT EMPTY AND The operator at TOP has Higher or Equal Precedence than the current operator

POP from the stack and ADD to the output list until this condition persists and finally **PUSH** the current operator into the stack.

Step 3 : After scanning the expression, **POP** all operators remaining in the stack to the output list.

Step 4 : Return the output list as the postfix expression.

Operator Precedence and Associativity:

Operator	Precedence	Associativity
^	Highest	Right to Left
*, /	Medium	Left to Right
+, -	Lowest	Left to Right

The given expression => (A + B) * C + D / (E + F * G) – H

Initialize Stack and Output List => Stack : [ ]
Output : [ ]

Scanned **(** => PUSH **(** into the stack => Stack : [( ]
Output : [ ]

Scanned **A** => ADD **A** into the output => Stack : [( ]
operand Output : [A]

Scanned **+** => PUSH **+** into the stack => Stack : [(, +]
Output : [A]

Scanned **B** => ADD **B** into the output => Stack : [(, +]
operand Output : [A , B]

Scanned **)** => POP until **(** is encountered and, ADD those elements to output
=> Stack : [ ]
Output : [A , B , +]

Scanned ***** => PUSH ***** into the stack => Stack : [*]

Output : [A , B , +]

Scanned **C** => ADD **C** into the output => Stack : [*]

operand Output : [A , B , + , C]

Scanned **+** => POP ***** from stack, ADD ***** to the output and PUSH **+** to stack

as * is higher precedence than +

=> Stack : [+]

Output : [A , B , + , C , *]

Scanned **D** => ADD **D** into the output => Stack : [*]

operand Output : [A , B , + , C, *, D]

Scanned **/** => PUSH **/** into the stack => Stack : [+ , /]

Output : [A , B , + , C , * , D]

Scanned **(** => PUSH **(** into the stack => Stack : [+ , / , ( ]

Output : [A , B , + , C , * , D]

Scanned **E** => ADD **E** into the output => Stack : [+ , / , ( ]

operand Output : [A , B , + , C , * , D, E]

Scanned **+** => PUSH **+** into the stack => Stack : [+ , / , (, +]

Output : [A , B , + , C , * , D, E]

Scanned **F** => ADD **F** into the output => Stack : [+ , / , ( ]

operand Output : [A , B , + , C , * , D, E , F]

Scanned ***** => PUSH ***** into the stack => Stack : [+ , / , (, + , *]

Output : [A , B , + , C , * , D, E , F]

Scanned **G** => ADD **G** into the output => Stack : [+ , / , (, + , *]

operand Output : [A , B , + , C , * , D, E , F , G]

Scanned **)** => POP until **(** is encountered and, ADD those elements to output

=> Stack : [+ , /]

Output : [A , B , + , C , * , D, E , F , G , * , +]

Scanned **–** => POP **/** and **+** from stack, ADD **/** and **+** to the output and PUSH **–** to stack, as / is higher precedence than – and, + is equal precedence with –

=> Stack : [–]

Output : [A , B , + , C , * , D, E , F , G , * , + , / , +]

Scanned **H** => ADD **H** into the output => Stack : [–]

operand Output : [A , B , + , C , * , D, E , F , G , * , + , / , + , H]

Now, there is no more element in the given expression.

So, POP all the remaining elements from the stack and ADD them to the output list.

=> Stack : [ ]

Output : [A , B , + , C , * , D, E , F , G , * , + , / , + , H , –]

Therefore, the final Postfix Expression is : **A B + C * D E F G * + / + H –**

12	**What is a Priority Queue ?**	**2**	**2022**

- A priority queue is an abstract data structure where the elements are processed based on their priority.
- Elements are **dequeued** (removed) based on their **priority**, but NOT the order in which they were added.
- Mostly, element with the **highest** priority is removed **first** → max-Priority-queue

  (*if element with lowest priority is dequeued first → min-PQ : this is very rarely used*)
- Applications of Priority queues are: (i) CPU Scheduling, (ii) Data compression with Huffman Coding, (iii) Shortest path algorithm, etc.

13	**Explain the concept of DEQUEUE.**	**2**	**2022**

- DEQUEUE refers to the operation of **removing** an element from the **front** of a **queue**.
- This operation is fundamental to the FIFO (First In, First Out) principle of a queue, where elements are added at the rear and removed from the front.
- The steps of DQUEUE are as follows:
  - If the queue is empty, then a dequeue operation cannot be performed, and typically an "underflow" condition is reported.
  - If the queue is not empty, remove the element pointed to by the FRONT pointer.
  - The FRONT pointer is then moved to the next element in the queue.
  - If the FRONT surpasses the REAR (*meaning, all elements have been removed*), then reset both pointers to indicate that the queue is empty.

Not to be confused with Deque.
Deque = Double Ended Queue.

14	**Write an algorithm for evaluating a postfix expression and evaluate the following postfix expression A B + C D / A D – E A ^ + * where A=2, B=7, C=9, D=3, E=5**	**6**	**2022**

Algorithm:

Step 1: **Initialize** an **empty** stack to store **operands**.

Step 2: **Scan** the postfix expression from left to right:

Step 3: IF the current token is an **operand**:

PUSH the operand onto the stack.

Step 4: IF the current token is an **operator**:

POP the top two **operands** from the stack

Apply the **operator** to these operands.

PUSH the result **back** onto the **stack**.

Step 5: Return the value from the stack as the result.

(*After scanning the entire expression, the stack contains only the result.*)

The given postfix expression: A B + C D / A D – E A ^ + *

The given values: A=2, B=7, C=9, D=3, E=5

So, replacing the operands by their values, the expression becomes:

2 7 + 9 3 / 2 3 – 5 2 ^ + *

Now, let's evaluate the postfix expression using the above algorithm.

Step 1 : initialize an empty stack.

Step 2 : Scan the postfix expression from left to right

PUSH operands onto the stack

Apply operators to the operands on the stack

Scanned **2** => operand => PUSH **2** into the stack => Stack : [2]

Scanned **7** => operand => PUSH **7** into the stack => Stack : [2 , 7]

Scanned **+** => operator

=> Apply **addition** on the top two elements => 2 + 7 = 9

=> POP the top two elements and PUSH the result at the top

=> Stack : [9]

Scanned **9** => operand => PUSH **9** into the stack => Stack : [9 , 9]

Scanned **3** => operand => PUSH **3** into the stack => Stack : [9 , 9 , 3]

Scanned **/** => operator

=> Apply **division** on the top two elements => 9 + 3 = 3

=> POP the top two elements and PUSH the result at the top

=> Stack : [9 , 3]

Scanned **2** => operand => PUSH **2** into the stack => Stack : [9 , 3 , 2]

Scanned **3** => operand => PUSH **3** into the stack => Stack : [9 , 3 , 2 , 3]

Scanned **–** => operator

=> Apply **subtraction** on the top two elements => 2 – 3 = –1

=> POP the top two elements and PUSH the result at the top

=> Stack : [9 , 3 , –1]

Scanned **5** => operand => PUSH **5** into the stack => Stack : [9 , 3 , –1 , 5]

Scanned **2** => operand => PUSH **2** into the stack => Stack : [9 , 3 , –1 , 5 , 2]

Scanned **^** => operator

=> Apply **exponent** on the top two elements => 5 ^ 2 = 25

=> POP the top two elements and PUSH the result at the top

=> Stack : [9 , 3 , –1 , 25]

Scanned **+** => operator

=> Apply **addition** on the top two elements => –1 + 25 = 24

=> POP the top two elements and PUSH the result at the top

=> Stack : [9 , 3 , 24]

Scanned ***** => operator

=> Apply **multiplication** on the top two elements => 3 * 24 = 72

=> POP the top two elements and PUSH the result at the top

=> Stack : [9 , 72]

There is no more operator in the expression. (*This is a special case. Here, we will repeat the operation for the remaining two operands. This is applicable for only + and **).

=> Apply **multiplication** on the top two elements => 9 * 72 = 684

=> Stack : [684]

Hence, the result is **684** .

15	**How can you reverse a string using stack? Give one example and show how you can reverse a given string using stack.**	6	2022

- We can reverse a string using a stack by taking advantage of the Last In, First Out (LIFO) property of stacks.
- The basic idea is to PUSH all the characters of the string onto a stack and then POP them one by one, which gives us the characters in reverse order.

ALGORITHM TO REVERSE A STRING USING A STACK

Step 1 : Initialize an empty stack.

Step 2 : PUSH each character of the string onto the stack.

Step 3 : POP each character from the stack and append it to the result string.

Step 4 : The result string will be the reversed version of the original string.

EXAMPLE:

Let's reverse the string "HELLO" using a stack.

Step 1 : Initialize an empty stack => Stack: [ ]

Step 2 : PUSH each character of the string onto the stack.

Push H	=>	Stack: [H]
Push E	=>	Stack: [H, E]
Push L	=>	Stack: [H, E, L]
Push L	=>	Stack: [H, E, L, L]
Push O	=>	Stack: [H, E, L, L, O]

Step 3 : POP each character from the stack and append it to the result string.

Pop O	=>	Result: "O"	Stack: [H, E, L, L]
Pop L	=>	Result: "OL"	Stack: [H, E, L]
Pop L	=>	Result: "OLL"	Stack: [H, E]
Pop E	=>	Result: "OLLE"	Stack: [H]
Pop H	=>	Result: "OLLEH"	Stack: [ ]

Final Result: The reversed string is "OLLEH".

16	**What is a stack ? What are the basic operations associated with the stack ? Convert the following arithmetic infix expression into postfix by using stack: A * (B + C) + (B / D) * A + Z * U.**	**6**	**2022**

- A stack is a linear data structure which follows the Last In, First Out (LIFO) principle
- The last element added to the stack is the first one to be removed.
- A stack can be visualized as a collection of elements placed one over the other, where the addition (PUSH) and removal (POP) of elements happen only at the TOP.

BASIC **OPERATIONS** ASSOCIATED WITH STACK

- **PUSH**: Adds an element to the top of the stack.
- **POP**: Removes and returns the element from the top of the stack.
- **TOP**: Returns the element at the top of the stack without removing it.
- **PEEK**: Same as **TOP**
- **isEmpty**: Checks whether the stack is empty.
- **isFull**: Checks whether the stack is full (in array-based implementations).

IN-FIX TO **POST-FIX** CONVERSION USING STACK

Step 1 : **Initialize an empty stack** for operators and an empty list for the output.

Step 2 : **Scan** the infix expression from **left** to **right**:

- **Operand**: Add it directly to the output list.

- **Left Parenthesis** (: Push it onto the stack.
- **Right Parenthesis**) : Pop from the stack to the output list until a left parenthesis is encountered. *Discard the pair of parentheses*.
- **Operator** (+ , – , * , / , ^):

  CASE 1 : STACK IS EMPTY

  PUSH the current operator onto the stack

  CASE 2 : STACK IS NOT EMPTY AND The operator at TOP has Lower Precedence than the current operator

  PUSH the current operator onto the stack

  CASE 3 : STACK IS NOT EMPTY AND The operator at TOP has Higher or Equal Precedence than the current operator

  POP from the stack and ADD to the output list until this condition persists and finally **PUSH** the current operator into the stack.

Step 3 : After scanning the expression, **POP** all operators remaining in the stack to the output list.

Step 4 : Return the output list as the postfix expression.

Operator Precedence and Associativity:

Operator	Precedence	Associativity
^	Highest	Right to Left
*, /	Medium	Left to Right
+, -	Lowest	Left to Right

Algorithm is NOT asked in this Question. So, you may skip the Algorithm if time does not permit.

The given expression => A * (B + C) + (B / D) * A + Z * U

Initialize Stack and Output List => Stack : [ ]
Output : [ ]

Scanned **A** => Add **A** to the output => Stack : [ ]
operand Output : [A]

Scanned ***** => PUSH ***** into the stack => Stack : [*]
Output : [A]

Scanned **(** => PUSH **(** into the stack => Stack : [* , ( ]
Output : [A]

Scanned **B** => Add **B** to the output => Stack : [* , ( ]
operand Output : [A , B]

Scanned **+** => PUSH **+** into the stack => Stack : [* , (, +]
Output : [A , B]

Scanned **C** => Add **C** to the output => Stack : [* , (, +]
operand Output : [A , B , C]

Scanned **)** => POP until **(** is encountered and, ADD those elements to output
=> Stack : [*]
Output : [A , B , C , +]

Scanned **+** => POP ***** from stack, ADD ***** to the output and PUSH **+** to stack
as * is higher precedence than +
=> Stack : [+]
Output : [A , B , C , + , *]

Scanned **(** => PUSH **(** into the stack => Stack : [+ , ( ]
Output : [A , B , C , + , *]

Scanned **B** => Add **B** to the output => Stack : [+ , ( ]
Output : [A , B , C , + , * , B]

Scanned **/** => PUSH **/** into the stack => Stack : [+ , (, /]
Output : [A , B , C , + , * , B]

Scanned **D** => Add **D** to the output => Stack : [+ , (, /]
Output : [A , B , C , + , * , B , D]

Scanned **)** => POP until **(** is encountered and, ADD those elements to output
=> Stack : [+]
Output : [A , B , C , + , * , B , D , /]

Scanned ***** => PUSH ***** into the stack => Stack : [+ , *]
Output : [A , B , C , + , * , B , D , /]

Scanned **A** => Add **A** to the output => Stack : [+ , *]
operand Output : [A , B , C , + , * , B , D , / , A]

Scanned **+** => POP ***** and **+** from stack and ADD ***** and **+** to the output

As both * and + are higher or same precedence than +

and then PUSH the current **+** to stack

=> Stack : [+]

Output : [A , B , C , + , * , B , D , / , A , * , +]

Scanned **Z** => Add **Z** to the output => Stack : [+]

operand Output : [A , B , C , + , * , B , D , / , A , * , +, Z]

Scanned ***** => PUSH ***** into the stack => Stack : [+ , *]

Output : [A , B , C , + , * , B , D , / , A , * , +, Z]

Scanned **U** => Add **U** to the output => Stack : [+ , *]

operand Output : [A , B , C , + , * , B , D , / , A , * , + , Z , U]

Now, the infix expression has no more elements.

So, POP the remaining elements from the stack and add to the Output.

Stack : [ ]

Output : [A , B , C , + , * , B , D , / , A , * , + , Z , U , * , +]

Hence, the final Post-Fix Expression is : **A B C + * B D / A * + Z U * +**

17	**Suppose a circular queue of capacity n elements is implemented with an array of n elements. assume that the insertion and deletion operation are carried out using REAR & FRONT as array index variables, respectively. Initially, REAR = FRONT = -1. Write the conditions to detect queue full and queue empty ?**	**2**	**2021**

- Queue is Empty: FRONT == -1
- Queue is Full: (REAR + 1) % n == FRONT

*Here, "**(REAR + 1) % n**" represents the next position after REAR.*

If this position is equal to FRONT, it means there is no blank space for insertions.

Which implies the queue is full.

18	**Evaluate the following postfix expression with single digit operands using a stack: 8 2 3 ^ / 2 3 * + 5 1 * -**	**2**	**2021**

Algorithm:

Step 1: **Initialize** an **empty** stack to store **operands**.

Step 2: **Scan** the postfix expression from left to right:

Step 3: IF the current token is an **operand**:

PUSH the operand onto the stack.

Step 4: IF the current token is an **operator**:

POP the top two **operands** from the stack

Apply the **operator** to these operands.

PUSH the result **back** onto the **stack**.

Step 5: Return the value from the stack as the result.

(*After scanning the entire expression, the stack contains only the result.*)

For 2 marks, you may skip the Algorithm.

The given postfix expression: 8 2 3 ^ / 2 3 * + 5 1 * -

Now, let's evaluate the postfix expression using the above algorithm.

Step 1 : initialize an empty stack.

Step 2 : Scan the postfix expression from left to right

PUSH operands onto the stack

Apply operators to the operands on the stack

Scanned **8** => operand => PUSH **8** into the stack => Stack : [8]

Scanned **2** => operand => PUSH **2** into the stack => Stack : [8 , 2]

Scanned **3** => operand => PUSH **3** into the stack => Stack : [8 , 2 , 3]

Scanned **^** => operator

=> Apply **exponent** on the top two elements => 2 ^ 3 = 8

=> POP the top two elements and PUSH the result at the top

=> Stack : [8 , 8]

Scanned **/** => operator

=> Apply **division** on the top two elements => 8 / 8 = 1

=> POP the top two elements and PUSH the result at the top

=> Stack : [1]

Scanned **2** => operand => PUSH **2** into the stack => Stack : [1 , 2]

Scanned **3** => operand => PUSH **3** into the stack => Stack : [1 , 2 , 3]

Scanned ***** => operator

=> Apply **multiplication** on the top two elements => 2 * 3 = 6

=> POP the top two elements and PUSH the result at the top

=> Stack : [1 , 6]

Scanned **+** => operator

=> Apply **addition** on the top two elements => 1 + 6 = 7

=> POP the top two elements and PUSH the result at the top

=> Stack : [7]

Scanned **5** => operand => PUSH **5** into the stack => Stack : [7 , 5]

Scanned **1** => operand => PUSH **1** into the stack => Stack : [7 , 1 , 5]

Scanned ***** => operator

=> Apply **multiplication** on the top two elements => 1 * 5 = 5

=> POP the top two elements and PUSH the result at the top

=> Stack : [7 , 5]

Scanned **–** => operator

=> Apply **subtraction** on the top two elements => 7 – 5 = 2

=> POP the top two elements and PUSH the result at the top

=> Stack : [2]

Therefore, the answer is **2** .

19	**Assume that the operators +, –, * are left associative and ^ is right associative. The order of precedence (from highest to lowest) is ^, *, +, –. Find the postfix expression for the infix expression: a + b * c – d ^ e ^ f**	**2**	**2021**

We can use Shunting Yard Algorithm to convert the expression from prefix to postfix.

Step 1 : **Initialize an empty stack** for operators and an empty list for the output.

Step 2 : **Scan** the infix expression from **left** to **right**:

- **Operand**: Add it directly to the output list.
- **Left Parenthesis** **(** : Push it onto the stack.
- **Right Parenthesis** **)** : Pop from the stack to the output list until a left parenthesis is encountered. *Discard the pair of parentheses*.
- **Operator** (+ , – , * , / , ^):

  CASE 1 : STACK IS EMPTY

  PUSH the current operator onto the stack

  CASE 2 : STACK IS NOT EMPTY AND The operator at TOP has Lower Precedence than the current operator

  PUSH the current operator onto the stack

  CASE 3 : STACK IS NOT EMPTY AND The operator at TOP has Higher or Equal Precedence than the current operator

  POP from the stack and ADD to the output list until this condition persists and finally **PUSH** the current operator into the stack.

For 2 marks, you may skip the Algorithm.

Step 3 : After scanning completes, **POP** all operators remaining in the stack to the output.

Step 4 : Return the output list as the postfix expression.

The given order of precedence (from highest to lowest) is : ^ , * , + , –.

The given expression => a + b * c – d ^ e ^ f

Initialize Stack and Output List => Stack : [ ]
Output : [ ]

Scanned **a** => Add **a** to the output => Stack : [ ]
operand Output : [a]

Scanned **+** => PUSH **+** into the stack => Stack : [+]
Output : [a]

Scanned **b** => Add **b** to the output => Stack : [+]
operand Output : [a , b]

Scanned ***** => PUSH ***** into the stack => Stack : [+ , *]
Output : [a , b]

Scanned **c** => Add **c** to the output => Stack : [+ , *]
operand Output : [a , b , c]

Scanned **–** => POP ***** and **+** from stack, ADD ***** and **+** to the output and PUSH **–** to stack, as both * and + are higher precedence than –.
=> Stack : [–]
Output : [a , b , c , * , +]

Scanned **d** => Add **d** to the output => Stack : [–]
operand Output : [a , b , c , * , + , d]

Scanned **^** => PUSH **^** into the stack as – is lower precedence than ^.
=> Stack : [– , ^]
Output : [a , b , c , * , + , d , e]

Scanned **e** => Add **e** to the output => Stack : [– , ^]
operand Output : [a , b , c , * , + , d , e]

Scanned **^** => PUSH **^** into the stack as the top and current elements are same.
=> Stack : [– , ^ , ^]
Output : [a , b , c , * , + , d , e]

Scanned **f** => Add **f** to the output => Stack : [– , ^ , ^]
operand Output : [a , b , c , * , + , d , e , f]

There is no more elements in the expression. So, POP the operators from stack and add to the output to get the final expression.

Hence, the final expression is : **a b c * + d e f ^ ^ –**

20	**How many queues are needed to implement a stack? consider the situation where no other data structure like arrays, linked-list is available to you. Justify your answer with a suitable example.**	**6**	**2021**

- To implement a stack using queues, we need at least **two queues** if no other data structure (like arrays or linked lists) is available.

JUSTIFICATION:

- Let's say we want to implement a stack using two queues, named **queue1** and **queue2**.
- We will use the two queues in such a way that the last element pushed onto the stack will be the first one to be popped.
- The operations can be implemented as follows

1. **Push Operation**:
   - Add the new element to **queue1**.
2. **Pop Operation**:
   - IF **queue1** is empty : return an error (underflow).
   - ELSE : move all elements from **queue1** to **queue2**, except the last element. The last element of **queue1** will be popped.
   - **Swap** the names of queue1 and queue2 so that,
     - queue1 is always used for new additions, and
     - queue2 becomes empty.

EXAMPLE:

Consider we have an empty stack, and we perform the following sequence of operations: **Push(1)**, **Push(2)**, **Push(3)**, and then **Pop()**.

Step-by-Step Implementation:

1. **Initial State**:
   - queue1: [ ]
   - queue2: [ ]
2. **Push(1)**:
   - Add 1 to queue1.
   - queue1: [1]
   - queue2: [ ]
3. **Push(2)**:
   - Add 2 to queue1.

- queue1: [1, 2]
- queue2: [ ]

4. **Push(3)**:
   - Add 3 to queue1.
   - queue1: [1, 2, 3]
   - queue2: [ ]
5. **Pop()**:
   - Move elements from queue1 to queue2 except the last element.
     - Move 1 from queue1 to queue2.

       queue1: [2, 3] queue2: [1]
     - Move 2 from queue1 to queue2.

       queue1: [3] queue2: [1, 2]
   - The last element left in queue1 (3) is popped.
   - Swap queue1 and queue2.
   - queue1: [1, 2] // will be used for new addition
   - queue2: [ ] // queue2 becomes empty
6. **Result of Pop**:
   - The popped value is 3, which follows the **LIFO** behaviour of a stack.

COMPLEXITY:

- *The **push** operation is always efficient (O(1)).*
- *The **pop** operation involves moving elements between the two queues, resulting in an **O(n)** time complexity, where n is the number of elements in the queue.*

Hence, we have implemented STACK using two QUEUEs.

21	**A circular queue has a size of 5 and has 3 elements 10, 20 and 40, where F=2 and R=4. After inserting 50 and 60, what is the value of F and R. Trying to insert 30 at this stage what happens? delete 2 elements from the queue and insert 70, 80 and 90. Show the sequence of steps with necessary diagrams with the value of F and R.**	6	2021

Initial State of Circular Queue:

- Size of Queue: 5
- Elements in Queue: [__ , __ , 10 , 20 , 40]
  - F = 2 (pointing to 10)
  - R = 4 (pointing to 40)

1. **Insert 50**:
   - **Current State**: F = 2, R = 4
   - Since R = 4 and the next position is (R + 1) % 5 = 0, we move R to 0.

- **New State**:
  - Queue: [50, __ , 10, 20, 40]
  - F = 2, R = 0 (pointing to 50)

2. **Insert 60**:
   - **Current State**: F = 2, R = 0
   - Since R = 0 and the next position is (R + 1) % 5 = 1, we move R to 1.
   - **New State**:
     - Queue: [50, 60, 10, 20, 40]
     - F = 2, R = 1 (pointing to 60)
3. **Insert 30**:
   - **Current State**: F = 2, R = 1
   - Since R = 1 and the next position is (R + 1) % 5 = 2, which is equal to F, it indicates that the **queue is full**.
   - **Result**: **Insertion of 30 is not possible** because the queue is full.

4. **Delete 2 Elements**:
   - **Current State**: F = 2, R = 1
   - **First Deletion**:
     - Delete the element at F = 2 (which is 10).
     - Move F to (F + 1) % 5 = 3.
     - **New State**:
       - Queue: [50, 60, _, 20, 40]
       - F = 3, R = 1
   - **Second Deletion**:
     - Delete the element at F = 3 (which is 20).
     - Move F to (F + 1) % 5 = 4.
     - **New State**:
       - Queue: [50, 60, _, _, 40]
       - F = 4, R = 1
5. **Insert 70**:
   - **Current State**: F = 4, R = 1
   - Since R = 1 and the next position is (R + 1) % 5 = 2, we move R to 2.
   - **New State**:
     - Queue: [50, 60, 70, _, 40]
     - F = 4, R = 2
6. **Insert 80**:
   - **Current State**: F = 4, R = 2
   - Since R = 2 and the next position is (R + 1) % 5 = 3, we move R to 3.

- **New State**:
  - Queue: [50, 60, 70, 80, 40]
  - F = 4, R = 3

7. **Insert 90**:
   - **Current State**: F = 4, R = 3
   - Since R = 3 and the next position is (R + 1) % 5 = 4, which is equal to F, it indicates that the **queue is full**.
   - **Result**: **Insertion of 90 is not possible** because the queue is full.

Final State of Circular Queue:

- **Queue**: [50, 60, 70, 80, 40]
- **Front (F)**: 4 (pointing to 40)
- **Rear (R)**: 3 (pointing to 80)

Summary of Operations:

- After inserting 50 and 60, the queue becomes **full** and cannot accommodate further elements.
- The **delete** operation removed two elements (10 and 20).
- The subsequent insertions of 70 and 80 were successful, but the insertion of 90 failed as the queue is already **full**.

22	**Write an ADT to implement stack of size N using an array. The elements in the stack are to be integers. The operations to be supported are PUSH, POP and DISPLAY. Take into account the exceptions of the stack overflow and stack underflow.**	**6**	**2021**

We will form an Abstract Data Type (ADT) to implement a **stack of size N** using an **array.**

- The stack stores **integer elements**
- The supported operations are **PUSH**, **POP**, and **DISPLAY**.
- We'll handle the **exceptions** for **stack overflow** and **stack underflow**.

Stack ADT:

1. Data Members:

- **int stack[N]** : An array to store the elements of the stack
  where, N is the maximum size of the stack.
- **int top** : A variable to keep the index of the top element in the stack.
  Initially, top = –1

2. Operations:

1. **PUSH(x)**: Adds an element x to the top of the stack.
   - **Input**: An integer element x.
   - **Algorithm**:

1. IF top == N – 1 : the stack is full.

   Raise an exception (**stack overflow**).

2. ELSE :

   increment top by 1

   assign stack[top] = x.

2. **POP()**: Removes and returns the element at the top of the stack.
   - **Output**: The element at the top of the stack.
   - **Algorithm**:
     1. IF top == –1 : the stack is empty.

        Raise an exception (**stack underflow**).

     2. ELSE :

        decrement top by 1

        return stack[top]

3. **DISPLAY()**: Displays all elements in the stack.
   - **Output**: Prints the elements from the bottom to the top of the stack.
   - **Algorithm**:
     1. IF top == –1, the stack is empty.

        Print "Stack is empty."

     2. ELSE :

        print the elements of the stack from index 0 to top.

PYTHON CODE:

```
class Stack:
    def __init__(self, N):
        self.stack = [0] * N  # Initialize stack array of size N
        self.top = -1  # Initialize top to -1
        self.size = N  # Size of the stack

    def push(self, x):
        # Check for stack overflow
        if self.top == self.size - 1:
            print("Stack Overflow! Cannot push", x)
            return

        # Increment top and add the element
        self.top += 1
        self.stack[self.top] = x
        print(f"Pushed {x} onto stack")
```

```
    def pop(self):
        # Check for stack underflow
        if self.top == -1:
            print("Stack Underflow! Cannot pop")
            return None

        # Return the element and decrement top
        popped_element = self.stack[self.top]
        self.top -= 1
        print(f"Popped {popped_element} from stack")
        return popped_element

    def display(self):
        # Check if stack is empty
        if self.top == -1:
            print("Stack is empty")
        else:
            # Print elements from bottom to top
            print("Stack elements:", end=" ")
            for i in range(self.top + 1):
                print(self.stack[i], end=" ")
            print()
# Example
stack_size = 5
s = Stack(stack_size)

s.push(10)    # Pushed 10 onto stack
s.push(20)    # Pushed 20 onto stack
s.push(30)    # Pushed 30 onto stack
s.display()   # Stack elements: 10 20 30
s.pop()       # Popped 30 from stack
s.display()   # Stack elements: 10 20
s.push(40)    # Pushed 40 onto stack
s.push(50)    # Pushed 50 onto stack
s.push(60)    # Pushed 60 onto stack
s.push(70)    # Stack Overflow! Cannot push 70
s.display()   # Stack elements: 10 20 40 50 60

# end of program
```

23	**explain the array implementation of ADT in detail. explain the insertion and deletion operations performed on a circular queue with necessary algorithms.**	**16**	**2021**

Array Implementation of ADT (Abstract Data Type)

- The **array implementation of an ADT** involves using a fixed-size array to store elements and manage them with specific operations that correspond to the properties of the ADT.
- For a **circular queue** using an array, we use two pointers: **FRONT** and **REAR**.

- This data structure efficiently uses space by wrapping around the array, avoiding the need for shifting elements when elements are removed.

Characteristics of Array-Based Circular Queue:

- **Size Limit**: The array is of a fixed size N.
- **Index Pointers**:
  - FRONT: Points to the first element in the queue.
  - REAR: Points to the last element in the queue.
- **Wrap Around**: When REAR reaches the end of the array, it wraps around to the beginning, enabling efficient use of space.

Circular Queue Operations:

0. Initialization:

- FRONT = -1, REAR = -1
- This indicates that the queue is empty.

1. Insertion Operation (Enqueue) in a Circular Queue:

The **enqueue operation** adds an element to the **rear** of the queue.

Algorithm for Enqueue (Insertion):

Check if the Queue is Full:

- The queue is **full** if (REAR + 1) % N == FRONT.
- If true, print "Queue is full" and exit.

Check if the Queue is Initially Empty:

- If FRONT == -1 (i.e., the queue is empty), set FRONT = 0.

Update REAR and Insert the Element:

- Set REAR = (REAR + 1) % N.
- Insert the element at queue[REAR].

Pseudocode:

```
ENQUEUE(queue, element)
  if ( (REAR + 1) % N == FRONT )
    print "Queue is full"
    return

  if (FRONT == -1)
    FRONT = 0

  REAR = (REAR + 1) % N
  queue[REAR] = element
  print element + " inserted"
```

Example:

Suppose the queue is [__ , __ , __ , __ , __] (size N = 5), and FRONT = -1, REAR = -1.

- **Insert 10**:
  - Queue was empty and now 1st element is inserted, so set FRONT = 0.
  - Set REAR = (REAR + 1) % 5 = 0.
  - Insert 10 at queue[0].
  - Queue: [10 , __ , __ , __ , __] FRONT = 0, REAR = 0
- **Insert 20**:
  - Set REAR = (REAR + 1) % 5 = 1.
  - Insert 20 at queue[1].
  - Queue: [10, 20, __ , __ , __] FRONT = 0, REAR = 1

2. Deletion Operation (Dequeue) in a Circular Queue:

The **dequeue operation** removes an element from the **front** of the queue.

Algorithm for Dequeue (Deletion):

1. **Check if the Queue is Empty**:
   - The queue is **empty** if FRONT == -1.
   - So, IF (FRONT == -1) : then print "Queue is empty" and exit.
2. **Retrieve the Element**:
   - Retrieve the element at queue[FRONT].
3. **Update FRONT and REAR**:
   - If FRONT == REAR :

     set FRONT = -1 and REAR = -1 (*the queue is now empty*).
   - ELSE:

     set FRONT = (FRONT + 1) % N.

Pseudocode:

```
DEQUEUE(queue)
  if (FRONT == -1)
    print "Queue is empty"
    return

  element = queue[FRONT]
  if (FRONT == REAR)
    FRONT = -1
    REAR = -1
  else
    FRONT = (FRONT + 1) % N

  print element + " removed"
```

Example:

Suppose the queue is [10, 20, _, _, _] (size N = 5), and FRONT = 0, REAR = 1.

- **Delete Element**:
  - Retrieve queue[FRONT] = 10.
  - Since FRONT != REAR, set FRONT = (FRONT + 1) % 5 = 1.
  - Queue: [10, 20, __ , __ , __]

    consider 10 logically removed => FRONT = 1, REAR = 1.
- **Delete Element**:
  - Retrieve queue[FRONT] = 20.
  - Since FRONT == REAR

    set FRONT = -1 and REAR = -1 (queue is now empty).
  - Queue: [10, 20, __ , __ , __]

    both 10 and 20 logically removed => FRONT = -1, REAR = -1.

Important Points for Circular Queue Implementation Using an Array:

1. **Wrap-Around**:
   - The key feature of a circular queue is that when either FRONT or REAR reaches the end of the array, it wraps around to the beginning using modulo arithmetic ((index + 1) % N).
2. **Full and Empty Conditions**:
   - **Full**: (REAR + 1) % N == FRONT
   - **Empty**: FRONT == -1
3. **Efficiency**:
   - The circular nature helps to utilize space efficiently without needing to shift elements after deletions.
   - The **time complexity** for both **enqueue** and **dequeue** operations is **O(1)**.

Visualization Example:

Let's visualize the queue after the sequence of operations:

- **Initial Queue**: [__ , __ , __ , __ , __]
  - FRONT = -1, REAR = -1
- **Insert 10, 20**:
  - Queue: [10, 20, __ , __ , __]
  - FRONT = 0, REAR = 1
- **Insert 30, 40, 50** (Queue is now full):
  - Queue: [10, 20, 30, 40, 50]
  - FRONT = 0, REAR = 4
- **Delete 10, 20**:
  - Queue: [10, 20, 30, 40, 50] (logical removal of 10 and 20)

- FRONT = 2, REAR = 4

- **Insert 60, 70** (Wrap Around):
  - Queue: [60, 70, 30, 40, 50]
  - FRONT = 2, REAR = 1 (after wrapping around)

This shows how **wrap-around** and **index pointers** help in managing a circular queue efficiently.

24	**how many minimum number of queues needed to implement the priority queue?**	**2**	**2020**

- In a **priority queue**, elements are assigned different priorities, and the **dequeue** operation always removes the element with the highest priority first.
- If we use **one queue** for each priority level, the following process can be applied:
  - **Enqueue Operation**: Insert elements into the queue corresponding to their priority.
  - **Dequeue Operation**: Always dequeue from the queue with the **highest priority** (non-empty).
- Hence, to implement a **priority queue**, we need at least **one queue per priority level**.
- If there is only **one priority level**, then a **single queue** is sufficient.

Therefore, the answer is : **ONE**

25	**How many minimum number of queues needed to implement the priority queue?**	**2**	**2020**

- A **minimum of two stacks** is required to implement a queue.
- **Stack1** is used for **enqueue** operations, and **Stack2** is used for **dequeue** operations.
- By transferring elements from **Stack1** to **Stack2**, the order of elements is reversed.
- This allows the implementation of the **FIFO** behaviour of a queue.

Algorithm for Queue Implementation Using Two Stacks:

1. **Enqueue Operation**
   - Push the element k onto **Stack1**.
2. **Dequeue Operation**
   - IF **Stack2** is empty:

     WHILE **Stack1** is not empty:

     POP the top element from **Stack1**

     PUSH it onto **Stack2**.
   - IF **Stack2** is still empty : the queue is empty (underflow condition).
   - ELSE: POP the top element from **Stack2**.

26	**List out the applications of stack.** **Write the algorithms for PUSH and POP operations done using stack.**	**6**	**2020**

Stacks are widely used in computer science and real-world applications due to their **Last In, First Out (LIFO)** property. Here are some of the common applications of stacks:

1. **Function Call Management**:
   - Stacks are used in programming languages to manage **function calls**. The **call stack** keeps track of active functions, parameters, and local variables during program execution.
2. **Expression Evaluation and Conversion**:
   - Stacks are used to convert **infix expressions to postfix** or **prefix** and to **evaluate postfix and prefix expressions**.
3. **Undo Mechanisms in Software**:
   - Applications like word processors or photo editors use stacks to implement the **undo** feature. Each action is pushed onto a stack, and an "undo" operation pops the last action.
4. **Parentheses Matching**:
   - Stacks are used to check if **parentheses, brackets, or braces** are balanced in an expression.
5. **Backtracking Algorithms**:
   - Stacks are used in **backtracking algorithms**, such as solving **mazes** or **graph traversal** (Depth-First Search).
6. **Recursion**:
   - When a program uses **recursion**, the system uses an implicit stack to manage function calls, allowing for **nested function** executions.
7. **Browser History Management**:
   - In web browsers, the **back** and **forward** functionality can be implemented using two stacks, where one stack stores visited pages, and the other stores the pages navigated back from.

PUSH and POP Operations (Stack using an Array) :

Data Members:

- stack[ ] : An array to store the stack elements.
- top : A variable to keep the index of the top element of the stack.
- N : Maximum size of the stack.

Algorithm for PUSH Operation:

PUSH(x): Adds an element x to the top of the stack.

1. **Check for Stack Overflow**:
   - If top == N – 1 :

     print "Stack Overflow" and return.

     // This indicates that the stack is full, and no more elements can be added.
2. **Increment the Top Index**:
   - Set top = top + 1
3. **Insert the Element**:

- Set stack[top] = x

Pseudocode for PUSH:

```
PUSH(stack, top, N, x)
  if (top == N - 1)
    print "Stack Overflow"
    return

  top = top + 1
  stack[top] = x
  print x + " pushed onto stack"
```

Algorithm for POP Operation:

POP(): Removes and returns the element at the top of the stack.

1. **Check for Stack Underflow**:
   - If top == -1 :

     print "Stack Underflow" and return.

     // This indicates that the stack is empty and there is no element to remove.
2. **Retrieve the Element**:
   - Set element = stack[top]
3. **Decrement the Top Index**:
   - Set top = top - 1
4. **Return the Element**:
   - Return element

Pseudocode for POP:

```
POP(stack, top)
  if (top == -1)
    print "Stack Underflow"
    return

  element = stack[top]
  top = top - 1
  print element + " popped from stack"
  return element
```

Example Implementation:

Suppose the stack is implemented as an array of size N = 5, and initially top = -1 (indicating an empty stack).

1. **PUSH(10)**:
   - top = top + 1 = 0

- stack[top] = 10
- **Stack State**: [10, __ , __ , __ , __], top = 0

2. **PUSH(20)**:
   - top = top + 1 = 1
   - stack[top] = 20
   - **Stack State**: [10, 20, __ , __ , __], top = 1
3. **POP()**:
   - Retrieve element = stack[top] = 20
   - top = top - 1 = 0
   - **Stack State**: [10, __ , __ , __ , __], top = 0
   - **Returned Element**: 20
4. **POP()**:
   - Retrieve element = stack[top] = 10
   - top = top - 1 = -1
   - **Stack State**: [__ , __ , __ , __ , __], top = -1
   - **Returned Element**: 10
5. **POP()**:
   - Since top == -1, **Stack Underflow** occurs.

Summary :

- **PUSH Operation**: Adds an element to the **top** of the stack and increments top. If the stack is full, it results in **Stack Overflow**.
- **POP Operation**: Removes and returns the element from the **top** of the stack and decrements top. If the stack is empty, it results in **Stack Underflow**.

These two operations satisfy the **LIFO** property of **STACK**.

27	**Enlist the advantages of circular queue over ordinary queue.** **Design a function CQINSERT for static implementation of circular queue.**	**6**	**2020**

Please note : CQUINSERT = Circular Queue INSERT

Advantages of Circular Queue over Ordinary Queue:

1. **Efficient Space Utilization**:
   - In an **ordinary queue** (linear queue), when elements are dequeued, the front moves forward, leading to unused space at the beginning of the array. As a result, even if there is empty space, a new element cannot be added unless the entire array is shifted.
   - A **circular queue** solves this problem by allowing the REAR to wrap around to the beginning when it reaches the end of the array. This ensures efficient use of the entire array.

2. **No Need for Shifting Elements**:
    - In an ordinary queue, if the FRONT moves beyond the first position, new elements can only be added by shifting the elements to the left, which is time-consuming.
    - In a circular queue, **shifting is not needed** because the REAR and FRONT pointers automatically wrap around.
3. **Consistent Time Complexity**:
    - Enqueue (insert) and Dequeue (delete) operations in a **circular queue** take **O(1)** time.
    - In an ordinary queue, the **enqueue** operation may take **O(n)** time in case shifting is required to make space for a new element.
4. **Better for Fixed Size Buffers**:
    - Circular queues are often used in situations where memory is limited and a **fixed size buffer** is needed, such as **stream processing** or **circular buffers** in operating systems and communication systems.

CQINSERT Function for Static Implementation of Circular Queue:

- CQUINSERT = Circular Queue INSERT
- CQINSERT function is used for inserting an element into a circular queue
- We have implemented this using an **array**.
- It assumes the use of two pointers FRONT and REAR to manage the queue.

Data Members:

- queue[ ] : An array of size N to store the elements of the queue.
- FRONT : Index of the front element of the queue. Initially, FRONT = -1
- REAR : Index of the rear element of the queue. Initially, REAR = -1
- N : Maximum size of the queue.

Algorithm for CQINSERT (Enqueue Operation):

1. **Check if the Queue is Full**:
    - If (REAR + 1) % N == FRONT :

        print "Queue is full" and return.
2. **Check if the Queue is Initially Empty**:
    - If FRONT == -1

        set FRONT = 0 // the first element is being inserted
3. **Update REAR and Insert the Element**:
    - Set REAR = (REAR + 1) % N.
    - Insert the element at queue[REAR].

Pseudocode for CQINSERT:

```
CQINSERT(queue, N, FRONT, REAR, element)
  if ( (REAR + 1) % N == FRONT )
    print "Queue is full"
    return

  if (FRONT == -1)
    FRONT = 0

  REAR = (REAR + 1) % N
  queue[REAR] = element
  print element + " inserted into queue"
```

Example:

Consider a queue with N = 5 (size is 5), and the queue is initially empty.

So, FRONT = -1, REAR = -1

- **Insert 10**:
  - FRONT = 0 (since the queue was empty).
  - REAR = (REAR + 1) % 5 = 0.
  - queue[REAR] = 10.
  - **Queue State**: [10, __ , __ , __ , __], FRONT = 0, REAR = 0.
- **Insert 20**:
  - REAR = (REAR + 1) % 5 = 1.
  - queue[REAR] = 20.
  - **Queue State**: [10, 20, __ , __ , __], FRONT = 0, REAR = 1.
- **Insert 30**:
  - REAR = (REAR + 1) % 5 = 2.
  - queue[REAR] = 30.
  - **Queue State**: [10, 20, 30, __ , __], FRONT = 0, REAR = 2.

Hence, the **CQINSERT** function handles the enqueue operation for a circular queue implemented using an array.

28	**Convert the following infix expression to prefix notation.** **E : (A + B * C * (M * N ^ P + T) - G + H).**	**6**	**2020**

First convert the sub-expressions into prefix notation.

Here, we need to check the operator with highest precedence.

So, the 1st sub-expression : N ^ P becomes ^ N P

=> E : (A + B * C * (M * ^ N P + T) - G + H)

Next, M * ^ N P becomes * ^ M N P

=> E : (A + B * C * (* ^ M N P + T) - G + H)

Next, * ^ M N P + T becomes + * ^ M N P T

=> E : (A + B * C * (+ * ^ M N P T) - G + H)

Here, the parenthesis are not required, as the expression inside it represents a single unit.

=> E : (A + B * C * + * ^ M N P T - G + H)

Next, B * C becomes * B C

=> E : (A + * B C * + * ^ M N P T - G + H)

Next, * B C * + * ^ M N P T becomes * * B C + * ^ M N P T

=> E : (A + * * B C + * ^ M N P T - G + H)

Next, A + * * B C + * ^ M N P T becomes + A * * B C + * ^ M N P T

=> E : (+ A * * B C + * ^ M N P T - G + H)

Next, + A * * B C + * ^ M N P T - G becomes - + A * * B C + * ^ M N P T G

=> E : (- + A * * B C + * ^ M N P T G + H)

Next, A + * * B C + * ^ M N P T - G + H becomes + - + A * * B C + * ^ M N P T G H

=> E : (+ - + A * * B C + * ^ M N P T G H)

Hence, the final expression becomes: **+ - + A * * B C + * ^ M N P T G H** (*Ans*)