DSC IA2

M2Q1) Define stack. Implement push and pop functions for stack using arrays with stack full and stack empty conditions.

A **stack** is a linear data structure that follows the **Last In First Out (LIFO)** principle. This means the element that is inserted last is the first one to be removed. The basic operations of a stack are:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the top element from the stack.
- **Peek/Top**: View the top element without removing it.
- **isEmpty**: Check if the stack is empty.
- **isFull**: Check if the stack is full (if implemented using arrays of fixed size).

Implementation using Arrays in C:

```
int stack[SIZE];
int top = -1;

void push(int value) {
    if (top == SIZE - 1) {
        printf("Stack Overflow (Stack Full)\n");
    } else {
        top++;
        stack[top] = value;
        printf("%d pushed to stack.\n", value);
    }
}

void pop() {
    if (top == -1) {
        printf("Stack Underflow (Stack Empty)\n");
    } else {
        printf("%d popped from stack.\n", stack[top]);
```

```
        top--;
    }
}
```

M2Q2) Write a function to evaluate a postfix expression and trace the same for the expressions: 1) ab/c - de*+ ac * where a= 6, b = 3, c = 1, d = 2, e = 4 and 2) ab/c-de*+ac*- given a=2, b=2, c=3, d=4, e=2.

Function to Evaluate a Postfix Expression

```
int evaluatePostfix(char expr[], int values[]) {
    for (int i = 0; expr[i] != '\0'; i++) {
        char ch = expr[i];
        if (isalpha(ch)) {
            push(values[ch - 'a']);  // a=0, b=1, ...
        } else {
            int op2 = pop();
            int op1 = pop();
            switch (ch) {
                case '+': push(op1 + op2); break;
                case '-': push(op1 - op2); break;
                case '*': push(op1 * op2); break;
                case '/': push(op1 / op2); break;
            }
        }
    }
    return pop();
}
void push(int val) {
    stack[++top] = val;
}
int pop() {
    return stack[top--];
}
```

Expression 1: `ab/c - de*+ ac *`

(Note: This seems malformed without clear operator precedence. Assuming proper postfix format is: `ab/cde* - +ac*`)

Given:

- `a = 6`, `b = 3`, `c = 1`, `d = 2`, `e = 4`
- values[ ] = {6, 3, 1, 2, 4}

Steps for: `ab/cde*-+ac*`

Step	Symbol	Action	Stack
1	a	Push 6	[6]
2	b	Push 3	[6, 3]
3	/	6 / 3 = 2	[2]
4	c	Push 1	[2, 1]
5	d	Push 2	[2, 1, 2]
6	e	Push 4	[2, 1, 2, 4]
7	*	2 * 4 = 8	[2, 1, 8]
8	-	1 - 8 = -7	[2, -7]
9	+	2 + (-7) = -5	[-5]
10	a	Push 6	[-5, 6]
11	c	Push 1	[-5, 6, 1]
12	*	6 * 1 = 6	[-5, 6]

✅ **Result: -5 + 6 = 1**

Expression 2: `ab/c-de*+ac*-`

Given:

- `a = 2`, `b = 2`, `c = 3`, `d = 4`, `e = 2`
- values[ ] = {2, 2, 3, 4, 2}

Steps:

Step	Symbol	Action	Stack
1	a	Push 2	[2]

Step	Symbol	Action	Stack
2	b	Push 2	[2, 2]
3	/	2 / 2 = 1	[1]
4	c	Push 3	[1, 3]
5	-	1 - 3 = -2	[-2]
6	d	Push 4	[-2, 4]
7	e	Push 2	[-2, 4, 2]
8	*	4 * 2 = 8	[-2, 8]
9	+	-2 + 8 = 6	[6]
10	a	Push 2	[6, 2]
11	c	Push 3	[6, 2, 3]
12	*	2 * 3 = 6	[6, 6]
13	-	6 - 6 = 0	[0]

✅ **Final Result: 0**

M2Q3) List the disadvantage of linear queue and explain how it is solved in circular queue. Give the algorithm to implement a circular queue with the suitable example.

Disadvantage of Linear Queue:

A **linear queue** implemented using arrays suffers from **inefficient memory usage**. Here's why:

- When elements are dequeued from the front, that space **cannot be reused** even if there is space available at the beginning of the array.
- Once the rear reaches the end (`rear == size - 1`), we cannot insert more elements, even if the front has moved ahead.

🔁 Circular Queue – The Solution:

A **circular queue** treats the array **as circular** (like a ring).

When `rear` reaches the end and there is free space at the beginning (i.e., `front != 0`), it wraps around and starts using those positions.

> This ensures better utilization of memory.

Circular Queue Algorithm:

```
void insertRear (int item, int q[], int* r, int* count) {
    if (*count == QUEUE_SIZE) {
        printf("Overflow of queue\n");
        return;
    }
    *r = (*r + 1) % QUEUE_SIZE;
    q[*r] = item;
    *count += 1;
}
void deleteFront (int q[], int* f, int* count) {
    if (*count == 0) {
        printf("Underflow of queue\n");
        return;
    }
    *f = (*f + 1) % QUEUE_SIZE;
    *count -= 1;
}
void display (int q[], int f, int count) {
    int i;
    if (count == 0) {
        printf("Queue is empty\n");
        return;
    }
    printf("Contents of queue\n");
    for (i=1; i<=count; i++) {
        printf("%d\n", q[f]);
        f = (f + 1) % QUEUE_SIZE;
    }
}
```

Example Execution:

For `enqueue(10), enqueue(20), enqueue(30), dequeue(), enqueue(40), enqueue(50), enqueue(60)` with SIZE = 5:

Step	Queue Contents	Front	Rear
enqueue(10)	[10, -, -, -, -]	0	0
enqueue(20)	[10, 20, -, -, -]	0	1
enqueue(30)	[10, 20, 30, -, -]	0	2
dequeue()	[10*, 20, 30, -, -]	1	2
enqueue(40)	[10, 20, 30, 40, -]	1	3
enqueue(50)	[10, 20, 30, 40, 50]	1	4
enqueue(60)	Wraps around	Queue Full (if front == 1)	

M2Q4) Convert the infix expression ((a/(b-c+d))*(e-a)*c) to postfix expression. Write a function to evaluate the postfix expression and trace for the given data a=6, b=3, c=1,d=2,e=4.

Convert Infix to Postfix

Given infix expression:

`((a / (b - c + d)) * (e - a) * c)`

Step-by-step conversion using operator precedence:

1. Innermost: `(b - c + d)` → `bc-d+`
2. Then: `a / (bc-d+)` → `abc-d+/`
3. Then: `(e - a)` → `ea-`
4. Multiply results: `abc-d+/ * ea- * c *` →

   ✅ **Final Postfix Expression:** `abc-d+/ea-*c*`

Evaluate Postfix Expression `abc-d+/ea-*c*`

Given values:

`a = 6, b = 3, c = 1, d = 2, e = 4`

So `values[] = {6, 3, 1, 2, 4}`

Step-by-step Trace

Step	Symbol	Stack Operation	Stack
1	a	Push 6	[6]
2	b	Push 3	[6, 3]
3	c	Push 1	[6, 3, 1]
4	-	3 - 1 = 2	[6, 2]
5	d	Push 2	[6, 2, 2]
6	+	2 + 2 = 4	[6, 4]
7	/	6 / 4 = 1	[1]
8	e	Push 4	[1, 4]
9	a	Push 6	[1, 4, 6]
10	-	4 - 6 = -2	[1, -2]
11	*	1 * -2 = -2	[-2]
12	c	Push 1	[-2, 1]
13	*	-2 * 1 = -2	[-2]

✅ **Final Result: -2**

M2Q5) Define Queue. Implement Qinsert and Qdelete function for queues using arrays.

A **Queue** is a linear data structure that follows the **FIFO** (First-In, First-Out) principle.

This means the element inserted first is the one to be removed first — like a line of people.

- **Insertion** happens at the **rear**.
- **Deletion** happens at the **front**.

Array Implementation of Queue:

Qinsert Function (Enqueue):

```
void Qinsert(int value) {
    if (rear == SIZE - 1) {
        printf("Queue Overflow\n");
        return;
```

```
    }

    if (front == -1)  // Inserting first element
        front = 0;

    rear++;
    queue[rear] = value;
    printf("%d inserted into queue\n", value);
}
```

Qdelete Function (Dequeue):

```
void Qdelete() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return;
    }

    printf("%d deleted from queue\n", queue[front]);
    front++;

    if (front > rear)  // Reset queue if empty
        front = rear = -1;
}
```

M2Q6) Write a note on Dequeue and priority queue.

Dequeue (Double-Ended Queue):

A **Dequeue (Double-Ended Queue)** is a linear data structure where **insertion and deletion can be performed from both ends** — **front** and **rear**.

Types of Dequeues:

- **Input-Restricted Dequeue:** Insertion only at one end, deletion at both.
- **Output-Restricted Dequeue:** Deletion only at one end, insertion at both.

Operations in Dequeue:

- `insertFront()`: Insert at front
- `insertRear()`: Insert at rear
- `deleteFront()`: Delete from front
- `deleteRear()`: Delete from rear

Applications:

- Job scheduling systems
- Palindrome checking
- Sliding window algorithms

Priority Queue:

A **Priority Queue** is an abstract data type where **each element is assigned a priority**, and **deletion is based on priority, not just order**.

- **Higher-priority** elements are dequeued **before** lower-priority ones.
- If two elements have the **same priority**, they follow **FIFO order**.

Types of Priority Queues:

- **Ascending Priority Queue**: Lower priority values removed first.
- **Descending Priority Queue**: Higher priority values removed first.

Implementation:

- Arrays
- Linked Lists
- Heaps (for efficient operations)

Applications:

- CPU scheduling
- Dijkstra's shortest path algorithm
- Task scheduling in operating systems

M2Q7) What is the advantage of circular queue over ordinary queue? Write a C program to simulate the working of circular queue of integers using array. Provide the following operations: Insert, Delete, Display.

Problem with Ordinary (Linear) Queue:

In a **linear queue**, once the `rear` reaches the end of the array (`SIZE - 1`), **no new insertions** can be made — even if space is freed up at the front due to deletions.

This leads to **inefficient memory usage**.

Advantage of Circular Queue:

A **circular queue** solves this by making the array behave like a circle:

- When `rear == SIZE - 1`, it wraps around to `0`, **if space is available** (i.e., `front != 0`).
- This allows **reuse of freed-up spaces** at the beginning.

> ✅ Efficient space utilization and better performance.

C Program: Circular Queue Implementation Using Array

```
#include <stdio.h>
#define SIZE 5

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

// Insert element in circular queue
void insert(int value) {
    if ((front == 0 && rear == SIZE - 1) || (rear + 1) % SIZE == front) {
        printf("Queue is Full\n");
        return;
    }

    if (front == -1)  // First insertion
        front = rear = 0;
    else
        rear = (rear + 1) % SIZE;

    queue[rear] = value;
    printf("%d inserted\n", value);
}
```

```
// Delete element from circular queue
void delete() {
    if (front == -1) {
        printf("Queue is Empty\n");
        return;
    }

    printf("%d deleted\n", queue[front]);

    if (front == rear)  // Last element
        front = rear = -1;
    else
        front = (front + 1) % SIZE;
}

// Display elements of circular queue
void display() {
    if (front == -1) {
        printf("Queue is Empty\n");
        return;
    }

    printf("Queue: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear)
            break;
        i = (i + 1) % SIZE;
    }
    printf("\n");
}

// Main function with menu
int main() {
    int choice, value;

    while (1) {
```

```
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                insert(value);
                break;
            case 2:
                delete();
                break;
            case 3:
                display();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice\n");
        }
    }
}
```

M2Q8) Write an algorithm and function to convert a valid infix expression to postfix expression.

Algorithm: Infix to Postfix Conversion

We use a **stack** to hold operators and parentheses during the conversion.

🔁 Steps:

1. **Initialize** an empty stack and an empty postfix string.
2. **Scan** the infix expression from left to right:
    - If the character is an **operand**, add it to postfix.
    - If the character is an **opening parenthesis** `(`, push it onto the stack.

- If the character is a **closing parenthesis** `)`, pop and add all operators to postfix until `(` is encountered. Discard the `(`.
- If the character is an **operator (+, -, *, /, ^)**:
  - While the **stack is not empty**, and **precedence of top of stack is greater than or equal to current operator**, pop and add to postfix.
  - Push the current operator onto the stack.

3. After the entire expression is scanned, **pop all remaining operators** from the stack to postfix.
4. **Return** the postfix expression.

Operator Precedence and Associativity

Operator	Precedence	Associativity
`^`	3	Right to Left
`*` `/`	2	Left to Right
`+` `-`	1	Left to Right

💻 C Function: Infix to Postfix

```
void infix_postfix(char infix[], char postfix[])
{
   int top;     /* points to top of the stack */
   int j;       /* Index for postfix expression */
   int i;       /* Index to access infix expression */
   char s[30];  /* Acts as storage for stack elements */
   char symbol; /* Holds scanned char from infix expression */

   top =-1;        /* Stack is empty */
   s[++top] = '#'; /* Initialize stack to # */
   j=0;            /* Output is empty. So, j = 0 */

   for( i = 0; i < strlen(infix); i++)
   {
      symbol = infix[i];  /* Scan the next symbol */
```

```
        /* if stack precedence greater, remove symbol
        from:stack and place into postfix*/

        while ( F(s[top]) > G(symbol) )
        {
            postfix[j] = s[top--];          /* Pop from stack and place */
            j++;                            /* into postfix */
        }
        if ( F(s[top]) != G(symbol) )
            s[++top] = symbol;              /* push the input symbol */
        else
            top--;                          /* discard *(* from stack */
    }
    /* Pop remaining symbols and place */
    /* them in postfix expression */
    while ( s[top] != '#')
    {
        postfix [j++] = s[top--];
    }
        postfix[j] = \0'; /* Attach NULL char at the end to frame.a string */
}
```

M3Q1) What is a linked list? Explain the different types of linked lists with a neat diagram.

A **linked list** is a linear data structure where each element (called a *node*) contains two parts:

1. **Data** – the value stored in the node.
2. **Link** (or pointer) – a reference to the next node in the sequence.

Unlike arrays, linked lists do not require contiguous memory locations. Each node is dynamically allocated and linked using pointers, making insertions and deletions more efficient.

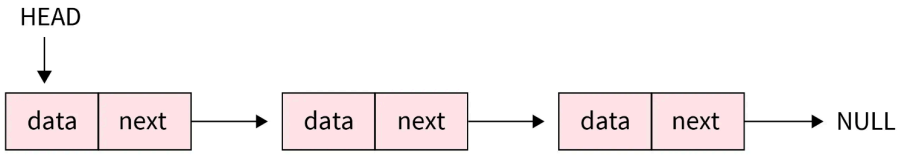
Types of Linked Lists:

1. Singly Linked List:

- Each node contains data and a pointer to the **next** node.
- The last node's next pointer is `NULL`, indicating the end of the list.

Structure:

```
struct Node {
    int data;
    struct Node* next;
};
```



SLL

2. Doubly Linked List:

- Each node contains three parts: a pointer to the **previous** node, the data, and a pointer to the **next** node.
- Allows traversal in both forward and backward directions.

Structure:

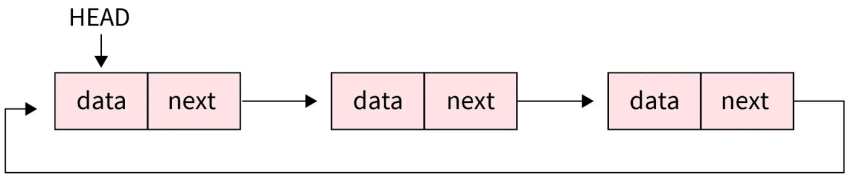
```
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
```

Diagram:

DLL

3. Circular Linked List:

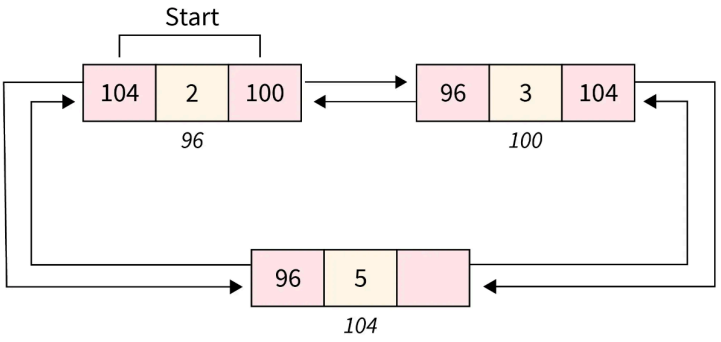
- Similar to a singly linked list, but the last node points back to the **first node**, forming a circle.
- Useful for applications that require circular traversal.



CLL

4. Circular Doubly Linked List:

- Like a doubly linked list, but the `next` pointer of the last node points to the first node, and the `prev` pointer of the first node points to the last node.



CDLL

M3Q2) Create SLL of integers and write C functions to perform the following: 1) Create a node list with data 10, 20 and 30. 2) Insert a node with value 15 in between 10 and 20. 3) Delete the node whose data is 20 Display the resulting SLL.

Structure Definition for a Singly Linked List Node:

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

struct Node {
    int data;
    struct Node* next;
};
```

Create a node list with data 10, 20, and 30

```
struct Node* createList() {
    struct Node *head, *second, *third;

    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    head→data = 10;
    head→next = second;

    second→data = 20;
    second→next = third;

    third→data = 30;
    third→next = NULL;

    return head;
}
```

Insert a node with value 15 between 10 and 20

```
void insertAfter(struct Node* head, int afterData, int newData) {
    struct Node* temp = head;
    while (temp != NULL && temp->data != afterData)
        temp = temp->next;

    if (temp != NULL) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = newData;
        newNode->next = temp->next;
        temp->next = newNode;
    }
}
```

Delete the node whose data is 20

```
void deleteNode(struct Node** head_ref, int key) {
    struct Node *temp = *head_ref, *prev = NULL;

    // If head node holds the key
    if (temp != NULL && temp->data == key) {
        *head_ref = temp->next;
        free(temp);
        return;
    }

    // Search for the key to delete
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return; // Key not found

    prev->next = temp->next;
```

```
    free(temp);
}
```

Display the resulting SLL

```
void display(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

M3Q3) Write operations for SSL: 1) Insert front(), 2) Insert end(), 3) Del_front(), 4) Del_End()

Insert at Front – `insert_front()`

```
void insert_front(struct Node** head_ref, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *head_ref;
    *head_ref = newNode;
}
```

Insert at End – `insert_end()`

```
void insert_end(struct Node** head_ref, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (*head_ref == NULL) {
        *head_ref = newNode;
        return;
```

```
    }

    struct Node* temp = *head_ref;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
}
```

Delete from Front – `del_front()`

```
void del_front(struct Node** head_ref) {
    if (*head_ref == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head_ref;
    *head_ref = temp->next;
    free(temp);
}
```

Delete from End – `del_end()`

```
void del_end(struct Node** head_ref) {
    if (*head_ref == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node* temp = *head_ref;

    // Only one node
    if (temp->next == NULL) {
        free(temp);
        *head_ref = NULL;
        return;
```

```
    }

    struct Node* prev = NULL;
    while (temp->next != NULL) {
        prev = temp;
        temp = temp->next;
    }

    prev->next = NULL;
    free(temp);
}
```

M3Q4) Write the program to show the stack implementation using linked list

A stack is a **LIFO** (Last In, First Out) structure. We use the **head** of the linked list as the **top** of the stack for efficient `push` and `pop` operations.

C Program: Stack Using Linked List

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

// Define node structure
struct Node {
    int data;
    struct Node* next;
};

// Push operation (insert at front)
void push(struct Node** top_ref, int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *top_ref;
    *top_ref = newNode;
    printf("%d pushed to stack.\n", value);
}
```

```
// Pop operation (delete from front)
void pop(struct Node** top_ref) {
    if (*top_ref == NULL) {
        printf("Stack Underflow! Cannot pop.\n");
        return;
    }

    struct Node* temp = *top_ref;
    *top_ref = temp->next;
    printf("%d popped from stack.\n", temp->data);
    free(temp);
}

// Peek operation (top element)
void peek(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Top element is: %d\n", top->data);
}

// Display stack contents
void display(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    printf("Stack contents (top to bottom): ");
    while (top != NULL) {
        printf("%d ", top->data);
        top = top->next;
    }
    printf("\n");
}

// Main function to demonstrate stack operations
int main() {
```

```
    struct Node* stack = NULL;

    push(&stack, 10);
    push(&stack, 20);
    push(&stack, 30);

    display(stack);
    peek(stack);

    pop(&stack);
    display(stack);

    return 0;
}
```

********** EOF **********