Unit 2
Stacks and Queues

Q1. Write a C program to implement a **stack using arrays**. Perform the following operations:

1. **Push** — Insert an element into the stack.
2. **Pop** — Remove an element from the stack.
3. **Peek** — Display the topmost element of the stack without removing it.
4. **Display** — Show all elements in the stack from top to bottom.

Solution:

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

int stack[MAX], top = -1;

void push(int value) {
   if (top >= MAX - 1)
```

```
        printf("Stack Overflow\n");
    else {
        top++;
        stack[top] = value;
    }
}

void pop() {
    if (top == -1)
        printf("Stack Underflow\n");
    else
        top--;
}

void peek() {
    if (top == -1)
        printf("Stack is Empty\n");
    else
        printf("Top Element: %d\n", stack[top]);
}

void display() {
    int i;
    if (top == -1)
        printf("Stack is Empty\n");
    else {
        printf("Stack Elements: ");
```

```
        for (i = top; i >= 0; i--)
            printf("%d ", stack[i]);
        printf("\n");
    }
}

void main() {
    int choice, value;
    clrscr();

    do {
        printf("\n1.Push\n2.Pop\n3.Peek\n4.Display\n5.Exit\nEnter
choice: ");
        scanf("%d", &choice);

        switch (choice) {
        case 1:
            printf("Enter value to push: ");
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            peek();
            break;
```

```
        case 4:
            display();
            break;
        case 5:
            break;
        default:
            printf("Invalid Choice\n");
        }
    } while (choice != 5);

    getch();
}
```

Q2. Write a C program to implement a **stack using linked lists**. Perform the following operations:

1. **Push** — Insert an element at the top of the stack.

2. **Pop** — Remove an element from the top of the stack.

3. **Peek** — Display the topmost element without removing it.

4. **Display** — Show all elements in the stack from top to bottom.

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

```
// Structure for stack node
struct Node {
    int data;
    struct Node* next;
};

struct Node* top = NULL; // Top pointer

// Push function
void push(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("%d pushed to stack\n", value);
}

// Pop function
void pop() {
struct Node *temp;
    if (top == NULL) {
        printf("Stack Underflow\n");
        return;
    }
 temp = top;
```

```
    printf("%d popped from stack\n", temp->data);
    top = top->next;
    free(temp);
}

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

// Main function
int main() {
int num;
printf("enter the value and press-1 to stop");
scanf("%d",&num);
while(num!=-1){
    push(num);}
```

```
   // push(20);
   // push(30);
    display();
   // pop();
   // display();
       getch();
    return 0;
}
```

Ques 3. Write a program to implement conversion of infix expression into postfix expression.

```
#include <stdio.h>
#include <ctype.h>   // For isalnum() function to check if a character is alphanumeric

char stack[100];     // Stack array to store operators during conversion
int top = -1;        // Stack top pointer, initialized to -1 (empty stack)

// Function to push a character onto the stack
void push(char x) {
    stack[++top] = x;    // Increment top and add the element
}

// Function to pop a character from the stack
char pop() {
    if (top == -1)       // If stack is empty
        return -1;       // Return -1 as an error/empty indicator
```

```
    else
        return stack[top--]; // Return top element and decrement top
}

// Function to return the priority (precedence) of an operator
int priority(char x) {
    if (x == '(')
        return 0;           // '(' has lowest precedence
    if (x == '+' || x == '-')
        return 1;           // '+' and '-' have precedence level 1
    if (x == '*' || x == '/')
        return 2;           // '*' and '/' have precedence level 2
    if (x == '^')
        return 3;           // '^' has highest precedence
    return 0;               // For non-operators, return 0
}

int main() {
    char exp[100];       // Array to store the input infix expression
    char *e, x;          // Pointer to traverse the expression, variable x
for popped char

    printf("Enter the expression : ");
    scanf("%s", exp);     // Read the infix expression (without spaces)
    printf("\n");

    e = exp;             // Point e to the start of the expression
```

```
    while (*e != '\0') {  // Loop until end of string
        if (isalnum(*e)) {    // If the character is an operand (letter or digit)
            printf("%c ", *e);  // Directly print it (operands appear in same order in postfix)
        }
        else if (*e == '(') {   // If character is an opening bracket
            push(*e);           // Push it onto the stack
        }
        else if (*e == ')') {   // If character is a closing bracket
            while ((x = pop()) != '(')    // Pop and print until '(' is found
                printf("%c ", x);
        }
        else { // Character is an operator (+, -, *, /, ^)
            // Pop from stack while:
            // 1. Operator at top has higher precedence
            // 2. OR same precedence and operator is NOT '^' //(because '^' is right associative)


            while (priority(stack[top]) > priority(*e) ||
                   (priority(stack[top]) == priority(*e) && *e != '^')) {
                printf("%c ", pop()); // Print the popped operator
            }
            push(*e); // Push current operator to stack
        }
```

```
        e++; // Move to next character
    }

    // Pop and print remaining operators from stack
    while (top != -1) {
printf("%c ", pop());
    }

    getch();
   return 0;
}
```

Queue

Ques 4. Create a simple Queue and do the following operations using arrays.

(i) Enqueue(Insert) (ii) Dequeue(Delete) (iii) Display

```
// Queue implementation in C

#include <stdio.h>
#define MAX 5


int items[MAX], front = -1, rear = -1;
```

```
void enQueue(int value) {
  if (rear == MAX - 1)
    printf("\nQueue is Full!!");
  else {
    if (front == -1)
      front = 0;
    rear++;
    items[rear] = value;
    printf("\nInserted -> %d", value);
  }
}

void deQueue() {
  if (front == -1)
    printf("\nQueue is Empty!!");
  else {
    printf("\nDeleted : %d", items[front]);
    front++;
    if (front > rear)
      front = rear = -1;
  }
```

```
}

// Function to print the queue
void display() {
  if (rear == -1)
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements are:\n");
    for (i = front; i <= rear; i++)
      printf("%d  ", items[i]);
  }
  printf("\n");
}

int main() {
  //deQueue is not possible on empty queue
  deQueue();

  //enQueue 5 elements
  enQueue(1);
  enQueue(2);
```

```
  enQueue(3);
  enQueue(4);
  enQueue(5);

  // 6th element can't be added to because the queue is full
  enQueue(6);

  display();

  //deQueue removes element entered first i.e. 1
  deQueue();

  //Now we have just 4 elements
  display();
  getch();
  return 0;
}
```

Ques 5. Create a simple Queue and do the following operations using linked lists

(i) Enqueue(Insert) (ii) Dequeue(Delete) (iii) Display

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

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

struct Node* front = NULL;
struct Node* rear = NULL;

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

```
    if (rear == NULL) {
                        front = rear = newNode;
    } else {
                        rear->next = newNode;
                        rear = newNode;
    }

    printf("%d inserted into queue\n", value);
}

void dequeue() {
struct Node *temp;
    if (front == NULL) {
                        printf("Queue is empty\n");
                        return;
    }

 temp = front;
    front = front->next;

    if (front == NULL)
                        rear = NULL;
```

```
    printf("%d removed from queue\n", temp->data);
    free(temp);
}

void display() {
    struct Node* temp = front;

    if (front == NULL) {
                    printf("Queue is empty\n");
                    return;
    }

    printf("Queue: ");
    while (temp != NULL) {
                    printf("%d -> ", temp->data);
                    temp = temp->next;
    }
    printf("NULL\n");
}


void peek() {
```

```
    if (front == NULL)
                printf("Queue is empty\n");
    else
                printf("Front element is: %d\n", front->data);
}

int main() {
clrscr();
    enqueue(10);
    enqueue(20);
    enqueue(30);

    display();

    dequeue();
    display();

    peek();
getch();
    return 0;
}
```

Ques 6. Create a Circular Queue and do the following operations using Array
(i) Enqueue (Insert) (ii) Dequeue (Delete) (iii) Display

```
#include<stdio.h>
#define MAX 5        // Maximum size of the circular queue

int queue[MAX];      // Array to store elements of the queue
int front = -1, rear = -1;  // Front and Rear pointers, -1 means queue is
                            empty

// -------------------------- ENQUEUE FUNCTION --------------------------
// Function to add an element to the circular queue
void enqueue (int value)
{
   // Condition for Queue Overflow:
   // Case 1: front is just next to rear (front == rear + 1) in circular manner
   // Case 2: front is 0 and rear is at the last index (MAX - 1)
   if ((front == rear + 1) || (front == 0 && rear == MAX - 1)) {
      printf ("Overflow condition\n");
   }
   else
   {
      // If queue is empty, set front to 0 (first insertion)
      if (front == -1)
```

```
        front = 0;

    // Move rear circularly (modulo MAX ensures wrap-around)
    rear = (rear + 1) % MAX;

    // Insert the value at the new rear position
    queue[rear] = value;

    // Inform user about insertion
    printf ("%d was enqueued to circular queue\n", value);
  }
}

// -------------------------- DEQUEUE FUNCTION --------------------------
// Function to remove an element from the circular queue
int dequeue ()
{
  int variable; // Temporary variable to hold the removed value

  // If queue is empty
  if (front == -1)
  {
    printf ("Underflow condition\n");
    return -1; // Return -1 to indicate failure
  }
```

```
    else
    {
        // Store the value at front
        variable = queue[front];

        // If this was the last element (queue will become empty)
        if (front == rear)
        {
            front = rear = -1; // Reset both to -1
        }
        else
        {
            // Move front circularly
            front = (front + 1) % MAX;
        }

        // Inform user about deletion
        printf ("%d was dequeued from circular queue\n", variable);

        return 1; // Success indicator
    }
}

// ------------------------- PRINT FUNCTION -------------------------
// Function to display the elements of the circular queue
```

```
void print ()
{
   int i;

   // If queue is empty
   if (front == -1)
      printf ("Nothing to delete\n");
   else
   {
      printf ("\nThe queue looks like: \n");

      // Loop from front to rear circularly
      for (i = front; i != rear; i = (i + 1) % MAX)
      {
         printf ("%d ", queue[i]);
      }

      // Print the last element at rear
      printf ("%d \n\n", queue[i]);
   }
}


// -------------------------- MAIN FUNCTION -------------------------
int main ()
```

```
{
   clrscr();
   // Try to dequeue from empty queue -> Underflow
   dequeue ();

   // Enqueue 5 elements into queue
   enqueue (15);
   enqueue (20);
   enqueue (25);
   enqueue (30);
   enqueue (35);

   // Display the queue
   print ();

   // Dequeue two elements
   dequeue ();
   dequeue ();

   // Display queue after deletion
   print ();

   // Enqueue more elements (wrap-around happens here)
   enqueue (40);
   enqueue (45);
```

```
    enqueue (50);   // This should work and fill queue
    enqueue (55);   // Overflow condition here

    // Display queue after more insertions
    print ();

    getch();
    return 0;
}
```

Ques 7. Write a program to implement Recursion.

Program: Factorial using Recursion

```
#include <stdio.h>

// Recursive function for factorial
int factorial(int n) {
    if (n == 0 || n == 1)  // Base condition
        return 1;
    else
        return n * factorial(n - 1);  // Recursive
              call
}
```

```
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);

    printf("Factorial of %d is %d\n", num,
                factorial(num));
    return 0;
}
```

Explanation of How Stack Works in Recursion(Not to write in notebook)

When a function is called recursively, **each call is pushed onto the stack** until the base case is reached. Then the stack starts **unwinding (popping)**.

👉 Example: Input = `5`

Step-by-step Stack Behavior

1. **Call:** `factorial(5)`
   → pushed to stack

Needs result of `5 * factorial(4)`

2. **Call:** `factorial(4)`
   → pushed to stack
   Needs result of `4 * factorial(3)`

3. **Call:** `factorial(3)`
   → pushed to stack
   Needs result of `3 * factorial(2)`

4. **Call:** `factorial(2)`
   → pushed to stack
   Needs result of `2 * factorial(1)`

5. **Call:** `factorial(1)`
   → Base case → returns `1`
   Stack starts popping.

Stack Unwinding (Return Phase)

- `factorial(1)` returns `1` → back to `factorial(2)`

- `factorial(2)` = `2 * 1 = 2` → back to `factorial(3)`
- `factorial(3)` = `3 * 2 = 6` → back to `factorial(4)`
- `factorial(4)` = `4 * 6 = 24` → back to `factorial(5)`
- `factorial(5)` = `5 * 24 = 120`

Final Answer: `120`