JSPM UNIVERSITY PUNE
FACULTY OF SCIENCE AND TECHNOLOGY
SCHOOL OF BASIC AND APPLIED SCIENCES

CERTIFICATE

Class: Year:

This is to certify that the work entered in this Journal is the work of

Mr/Ms.____

of ____________________ division ________ Seat No. __________

has satisfactorily completed the required number of practical for

the ______________________________ course of Semester ______

year 20___ - 20___ in the laboratory as laid down by the University.

Date:

Course Co-ordinator	**Program Co-ordinator**	**I/C Dean, Faculty of Science and Technology**

Q1.Basic Programs - Control Flow Structure (Expression, Conditional, looping, data

Ans. ➔1. Expression and Basic Arithmetic

```
#include <stdio.h>

int main() {
    int a = 10, b = 5;
    int sum = a + b;
    int product = a * b;
    printf("Sum: %d\n", sum);
    printf("Product: %d\n", product);
    return 0;
}
```

Output:

```
Sum: 15
Product: 50
```

2. Conditional Statement (if-else)

```
#include <stdio.h>

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

    if (number % 2 == 0) {
        printf("The number %d is even.\n", number);
    } else {
        printf("The number %d is odd.\n", number);
    }
    return 0;
}
```

Sample Input:

```
7
```

Output:

```
The number 7 is odd.
```

---

3. Looping Structure (for loop)

```
#include <stdio.h>

int main() {
    printf("Numbers from 1 to 5:\n");
    for (int i = 1; i <= 5; i++) {
        printf("%d\n", i);
    }
    return 0;
}
```

Output:

```
Numbers from 1 to 5:
1
2
3
4
5
```

---

4. Looping Structure (while loop)

```
#include <stdio.h>

int main() {
    int count = 1;
    printf("Counting from 1 to 5 using while loop:\n");
    while (count <= 5) {
        printf("%d\n", count);
        count++;
    }
    return 0;
}
```

Output:

```
Counting from 1 to 5 using while loop:
1
2
3
4
5
```

---

5. Looping and Conditional (Multiplication Table using Nested Loop)

```
#include <stdio.h>

int main() {
    printf("Multiplication Table (1 to 5):\n");
```

```
    for (int i = 1; i <= 5; i++) {
        for (int j = 1; j <= 5; j++) {
            printf("%d x %d = %d\t", i, j, i * j);
        }
        printf("\n");
    }
    return 0;
}
```

Output:

```
Multiplication Table (1 to 5):
1 x 1 = 1       1 x 2 = 2       1 x 3 = 3       1 x 4 = 4       1 x 5 = 5
2 x 1 = 2       2 x 2 = 4       2 x 3 = 6       2 x 4 = 8       2 x 5 = 10
3 x 1 = 3       3 x 2 = 6       3 x 3 = 9       3 x 4 = 12      3 x 5 = 15
4 x 1 = 4       4 x 2 = 8       4 x 3 = 12      4 x 4 = 16      4 x 5 = 20
5 x 1 = 5       5 x 2 = 10      5 x 3 = 15      5 x 4 = 20      5 x 5 = 25
```

---

6. Data Manipulation (Array Traversal)

```
#include <stdio.h>

int main() {
    int numbers[] = {10, 20, 30, 40, 50};
    int size = sizeof(numbers) / sizeof(numbers[0]);

    printf("Elements of the array:\n");
    for (int i = 0; i < size; i++) {
        printf("%d\n", numbers[i]);
    }
    return 0;
}
```

Output:

```
Elements of the array:
10
20
30
40
50
```

---

7. Data Manipulation (String Handling)

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

int main() {
    char str1[] = "Hello";
    char str2[] = "World";
    char result[20];
```

```
    strcpy(result, str1);
    strcat(result, " ");
    strcat(result, str2);

    printf("Concatenated String: %s\n", result);
    return 0;
}
```

Output:

```
Concatenated String: Hello World
```

Q2. Implement insertion ,deletion ,and searching in single-dimensional and multidimensional.

Ans ➔ ➔

1. Single-Dimensional Array

1.1 Insertion in Single-Dimensional Array

```
#include <stdio.h>

int main() {
    int arr[100] = {1, 2, 3, 4, 5};
    int n = 5; // Initial size of the array
    int pos = 3, value = 10; // Insert value 10 at position 3

    printf("Original Array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Shift elements to the right
    for (int i = n; i >= pos; i--) {
        arr[i] = arr[i - 1];
    }
    arr[pos - 1] = value; // Insert value
    n++; // Increase size

    printf("Array after insertion:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Original Array:
1 2 3 4 5
Array after insertion:
1 2 10 3 4 5
```

1.2 Deletion in Single-Dimensional Array

```c
#include <stdio.h>

int main() {
    int arr[100] = {1, 2, 3, 4, 5};
    int n = 5; // Initial size of the array
    int pos = 3; // Delete element at position 3

    printf("Original Array:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    // Shift elements to the left
    for (int i = pos - 1; i < n - 1; i++) {
        arr[i] = arr[i + 1];
    }
    n--; // Decrease size

    printf("Array after deletion:\n");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
Original Array:
1 2 3 4 5
Array after deletion:
1 2 4 5
```

1.3 Searching in Single-Dimensional Array

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = 5;
    int key = 3; // Element to search for
    int found = 0;

    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            printf("Element %d found at position %d.\n", key, i + 1);
```

```
                found = 1;
                break;
            }
        }
        if (!found) {
            printf("Element %d not found in the array.\n", key);
        }

        return 0;
    }
```

Output:

```
Element 3 found at position 3.
```

---

2. Multi-Dimensional Array

2.1 Insertion in Multi-Dimensional Array

```
#include <stdio.h>

int main() {
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}}; // Initial array with
space for one insertion
    int row = 2, col = 2, value = 9; // Insert 9 at position (2, 2)

    printf("Original Matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    arr[row][col] = value;

    printf("Matrix after insertion:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Original Matrix:
1 2 3
4 5 6
7 8 0
Matrix after insertion:
```

```
1 2 3
4 5 6
7 8 9
```

2.2 Deletion in Multi-Dimensional Array

```
#include <stdio.h>

int main() {
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int row = 1, col = 1; // Delete element at position (1, 1)

    printf("Original Matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    arr[row][col] = 0; // Replace with 0 or a placeholder

    printf("Matrix after deletion:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    return 0;
}
```

Output:

```
Original Matrix:
1 2 3
4 5 6
7 8 9
Matrix after deletion:
1 2 3
4 0 6
7 8 9
```

2.3 Searching in Multi-Dimensional Array

```
#include <stdio.h>

int main() {
    int arr[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    int key = 5; // Element to search for
    int found = 0;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

```
            if (arr[i][j] == key) {
                printf("Element %d found at position (%d, %d).\n", key, i,
j);
                found = 1;
                break;
            }
        }
        if (found) break;
    }
    if (!found) {
        printf("Element %d not found in the matrix.\n", key);
    }

    return 0;
}
```

Output:

```
Element 5 found at position (1, 1).
```

Q3. Demonstrate row-major and column-major order representations of 2D arrays.

Ans➔ ➔

Row-Major and Column-Major Order Demonstration

```
#include <stdio.h>

int main() {
    // Declare a 2D array
    int arr[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    printf("2D Array:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Row-major order representation
    printf("\nRow-Major Order Representation:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", arr[i][j]);
        }
```

```
    }
    printf("\n");

    // Column-major order representation
    printf("\nColumn-Major Order Representation:\n");
    for (int j = 0; j < 3; j++) {
        for (int i = 0; i < 3; i++) {
            printf("%d ", arr[i][j]);
        }
    }
    printf("\n");

    return 0;
}
```

Output :

```
2D Array:
1 2 3
4 5 6
7 8 9

Row-Major Order Representation:
1 2 3 4 5 6 7 8 9

Column-Major Order Representation:
1 4 7 2 5 8 3 6 9
```

Q4. Implement a sparse matrix using a linked list or arrays.

Ans ➔

Sparse Matrix Representation Using Linked List

Each node in the linked list stores:

1. **Row index**
2. **Column index**
3. **Value**
4. **Pointer to the next node**

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

// Define a node structure for the sparse matrix
typedef struct Node {
    int row, col, value;
    struct Node* next;
} Node;
```

```c
// Function to create a new node
Node* createNode(int row, int col, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->row = row;
    newNode->col = col;
    newNode->value = value;
    newNode->next = NULL;
    return newNode;
}

// Function to display the sparse matrix from the linked list
void displaySparseMatrix(Node* head) {
    printf("Row\tCol\tValue\n");
    while (head != NULL) {
        printf("%d\t%d\t%d\n", head->row, head->col, head->value);
        head = head->next;
    }
}

// Function to convert a 2D matrix into a sparse matrix linked list
Node* convertToSparseMatrix(int rows, int cols, int matrix[rows][cols]) {
    Node* head = NULL;
    Node* tail = NULL;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (matrix[i][j] != 0) {
                Node* newNode = createNode(i, j, matrix[i][j]);
                if (head == NULL) {
                    head = newNode;
                    tail = newNode;
                } else {
                    tail->next = newNode;
                    tail = newNode;
                }
            }
        }
    }
    return head;
}

int main() {
    // Define a 2D matrix
    int matrix[4][5] = {
        {0, 0, 3, 0, 4},
        {0, 0, 5, 7, 0},
        {0, 0, 0, 0, 0},
        {0, 2, 6, 0, 0}
    };

    printf("Original Matrix:\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 5; j++) {
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
```

```
    // Convert to sparse matrix representation
    Node* sparseMatrix = convertToSparseMatrix(4, 5, matrix);

    printf("\nSparse Matrix Representation:\n");
    displaySparseMatrix(sparseMatrix);

    return 0;
}
```

Output

Original Matrix:

```
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
```

Sparse Matrix Representation:

```
Row     Col     Value
0       2       3
0       4       4
1       2       5
1       3       7
3       1       2
3       2       6
```

Q5. Implement stack operations (push, pop, peek) using both arrays and linked lists.

Ans ➔

1. Stack Implementation Using Arrays

Code:

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

// Stack structure using array
typedef struct {
    int arr[MAX];
    int top;
} Stack;

// Initialize stack
void initializeStack(Stack* stack) {
    stack->top = -1;
}
```

```
// Push operation
void push(Stack* stack, int value) {
    if (stack->top == MAX - 1) {
        printf("Stack Overflow! Cannot push %d\n", value);
    } else {
        stack->arr[++stack->top] = value;
        printf("Pushed %d onto the stack\n", value);
    }
}

// Pop operation
int pop(Stack* stack) {
    if (stack->top == -1) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    } else {
        return stack->arr[stack->top--];
    }
}

// Peek operation
int peek(Stack* stack) {
    if (stack->top == -1) {
        printf("Stack is empty\n");
        return -1;
    } else {
        return stack->arr[stack->top];
    }
}

// Display stack
void display(Stack* stack) {
    if (stack->top == -1) {
        printf("Stack is empty\n");
    } else {
        printf("Stack elements: ");
        for (int i = 0; i <= stack->top; i++) {
            printf("%d ", stack->arr[i]);
        }
        printf("\n");
    }
}

int main() {
    Stack stack;
    initializeStack(&stack);

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

    printf("Popped: %d\n", pop(&stack));
    display(&stack);

    printf("Peek: %d\n", peek(&stack));
    display(&stack);
```

```
    return 0;
}
```

Output:

```
Pushed 10 onto the stack
Pushed 20 onto the stack
Pushed 30 onto the stack
Stack elements: 10 20 30
Popped: 30
Stack elements: 10 20
Peek: 20
Stack elements: 10 20
```

---

2. Stack Implementation Using Linked Lists

Code:

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

// Node structure for stack
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Push operation
void push(Node** top, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Stack Overflow! Cannot push %d\n", value);
        return;
    }
    newNode->data = value;
    newNode->next = *top;
    *top = newNode;
    printf("Pushed %d onto the stack\n", value);
}

// Pop operation
int pop(Node** top) {
    if (*top == NULL) {
        printf("Stack Underflow! Cannot pop\n");
        return -1;
    }
    Node* temp = *top;
    int value = temp->data;
    *top = (*top)->next;
    free(temp);
    return value;
}

// Peek operation
```

```c
int peek(Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return -1;
    }
    return top->data;
}

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

int main() {
    Node* stack = NULL;

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

    printf("Popped: %d\n", pop(&stack));
    display(stack);

    printf("Peek: %d\n", peek(stack));
    display(stack);

    return 0;
}
```

Output:

```
Pushed 10 onto the stack
Pushed 20 onto the stack
Pushed 30 onto the stack
Stack elements: 30 20 10
Popped: 30
Stack elements: 20 10
Peek: 20
Stack elements: 20 10
```

Q6. Implement queue operations (enqueue, dequeue) using arrays and linked lists

Ans ➔

1. Queue Implementation Using Arrays

Code:

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

typedef struct {
    int arr[MAX];
    int front;
    int rear;
} Queue;

// Initialize the queue
void initializeQueue(Queue* queue) {
    queue->front = -1;
    queue->rear = -1;
}

// Enqueue operation
void enqueue(Queue* queue, int value) {
    if (queue->rear == MAX - 1) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
        return;
    }
    if (queue->front == -1) queue->front = 0; // Initialize front for first
element
    queue->arr[++queue->rear] = value;
    printf("Enqueued %d into the queue\n", value);
}

// Dequeue operation
int dequeue(Queue* queue) {
    if (queue->front == -1 || queue->front > queue->rear) {
        printf("Queue Underflow! Cannot dequeue\n");
        return -1;
    }
    return queue->arr[queue->front++];
}

// Display queue
void display(Queue* queue) {
    if (queue->front == -1 || queue->front > queue->rear) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements: ");
    for (int i = queue->front; i <= queue->rear; i++) {
        printf("%d ", queue->arr[i]);
    }
    printf("\n");
}

int main() {
```

```
    Queue queue;
    initializeQueue(&queue);

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);
    display(&queue);

    printf("Dequeued: %d\n", dequeue(&queue));
    display(&queue);

    enqueue(&queue, 40);
    display(&queue);

    return 0;
}
```

Output:

```
Enqueued 10 into the queue
Enqueued 20 into the queue
Enqueued 30 into the queue
Queue elements: 10 20 30
Dequeued: 10
Queue elements: 20 30
Enqueued 40 into the queue
Queue elements: 20 30 40
```

---

2. Queue Implementation Using Linked List

Code:

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

// Node structure for the queue
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Enqueue operation
void enqueue(Node** front, Node** rear, int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (!newNode) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
        return;
    }
    newNode->data = value;
    newNode->next = NULL;

    if (*rear == NULL) { // First element in the queue
        *front = *rear = newNode;
    } else {
        (*rear)->next = newNode;
```

```
        *rear = newNode;
    }
    printf("Enqueued %d into the queue\n", value);
}

// Dequeue operation
int dequeue(Node** front, Node** rear) {
    if (*front == NULL) {
        printf("Queue Underflow! Cannot dequeue\n");
        return -1;
    }
    Node* temp = *front;
    int value = temp->data;
    *front = (*front)->next;
    if (*front == NULL) *rear = NULL; // Queue is empty now
    free(temp);
    return value;
}

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

int main() {
    Node* front = NULL;
    Node* rear = NULL;

    enqueue(&front, &rear, 10);
    enqueue(&front, &rear, 20);
    enqueue(&front, &rear, 30);
    display(front);

    printf("Dequeued: %d\n", dequeue(&front, &rear));
    display(front);

    enqueue(&front, &rear, 40);
    display(front);

    return 0;
}
```

Output:

```
Enqueued 10 into the queue
Enqueued 20 into the queue
Enqueued 30 into the queue
```

```
Queue elements: 10 20 30
Dequeued: 10
Queue elements: 20 30
Enqueued 40 into the queue
Queue elements: 20 30 40
```

Q7. Implement circular queue, priority queue, and double-ended queue (deque).

Ans➔

1. Circular Queue

Code:

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

typedef struct {
    int arr[MAX];
    int front;
    int rear;
} CircularQueue;

// Initialize Circular Queue
void initializeCircularQueue(CircularQueue* queue) {
    queue->front = -1;
    queue->rear = -1;
}

// Enqueue operation
void enqueue(CircularQueue* queue, int value) {
    if ((queue->rear + 1) % MAX == queue->front) {
        printf("Circular Queue Overflow! Cannot enqueue %d\n", value);
        return;
    }
    if (queue->front == -1) queue->front = 0;
    queue->rear = (queue->rear + 1) % MAX;
    queue->arr[queue->rear] = value;
    printf("Enqueued %d into the circular queue\n", value);
}

// Dequeue operation
int dequeue(CircularQueue* queue) {
    if (queue->front == -1) {
        printf("Circular Queue Underflow! Cannot dequeue\n");
        return -1;
    }
    int value = queue->arr[queue->front];
    if (queue->front == queue->rear) {
        queue->front = -1; // Queue becomes empty
        queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % MAX;
    }
```

```c
    return value;
}

// Display circular queue
void displayCircularQueue(CircularQueue* queue) {
    if (queue->front == -1) {
        printf("Circular Queue is empty\n");
        return;
    }
    printf("Circular Queue elements: ");
    int i = queue->front;
    while (1) {
        printf("%d ", queue->arr[i]);
        if (i == queue->rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    CircularQueue queue;
    initializeCircularQueue(&queue);

    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 30);
    enqueue(&queue, 40);
    displayCircularQueue(&queue);

    printf("Dequeued: %d\n", dequeue(&queue));
    displayCircularQueue(&queue);

    enqueue(&queue, 50);
    displayCircularQueue(&queue);

    enqueue(&queue, 60); // Overflow scenario
    return 0;
}
```

Output:

```
Enqueued 10 into the circular queue
Enqueued 20 into the circular queue
Enqueued 30 into the circular queue
Enqueued 40 into the circular queue
Circular Queue elements: 10 20 30 40
Dequeued: 10
Circular Queue elements: 20 30 40
Enqueued 50 into the circular queue
Circular Queue elements: 20 30 40 50
Circular Queue Overflow! Cannot enqueue 60
```

---

2. Priority Queue

A **priority queue** stores elements based on their priority values, where smaller numbers represent higher priority.

Code:

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

typedef struct {
    int arr[MAX];
    int size;
} PriorityQueue;

// Initialize Priority Queue
void initializePriorityQueue(PriorityQueue* queue) {
    queue->size = 0;
}

// Enqueue operation
void enqueue(PriorityQueue* queue, int value) {
    if (queue->size == MAX) {
        printf("Priority Queue Overflow! Cannot enqueue %d\n", value);
        return;
    }
    int i = queue->size - 1;
    while (i >= 0 && queue->arr[i] > value) {
        queue->arr[i + 1] = queue->arr[i];
        i--;
    }
    queue->arr[i + 1] = value;
    queue->size++;
    printf("Enqueued %d into the priority queue\n", value);
}

// Dequeue operation
int dequeue(PriorityQueue* queue) {
    if (queue->size == 0) {
        printf("Priority Queue Underflow! Cannot dequeue\n");
        return -1;
    }
    return queue->arr[--queue->size];
}

// Display priority queue
void displayPriorityQueue(PriorityQueue* queue) {
    if (queue->size == 0) {
        printf("Priority Queue is empty\n");
        return;
    }
    printf("Priority Queue elements: ");
    for (int i = 0; i < queue->size; i++) {
        printf("%d ", queue->arr[i]);
    }
    printf("\n");
}

int main() {
```

```
    PriorityQueue queue;
    initializePriorityQueue(&queue);

    enqueue(&queue, 30);
    enqueue(&queue, 10);
    enqueue(&queue, 20);
    enqueue(&queue, 40);
    displayPriorityQueue(&queue);

    printf("Dequeued: %d\n", dequeue(&queue));
    displayPriorityQueue(&queue);

    return 0;
}
```

Output:

```
Enqueued 30 into the priority queue
Enqueued 10 into the priority queue
Enqueued 20 into the priority queue
Enqueued 40 into the priority queue
Priority Queue elements: 10 20 30 40
Dequeued: 40
Priority Queue elements: 10 20 30
```

---

3. Double-Ended Queue (Deque)

A **deque** allows insertion and deletion from both ends.

Code:

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

typedef struct {
    int arr[MAX];
    int front;
    int rear;
} Deque;

// Initialize Deque
void initializeDeque(Deque* deque) {
    deque->front = -1;
    deque->rear = -1;
}

// Insert at front
void insertFront(Deque* deque, int value) {
    if ((deque->front == 0 && deque->rear == MAX - 1) || (deque->front ==
deque->rear + 1)) {
        printf("Deque Overflow! Cannot insert %d at front\n", value);
        return;
    }
    if (deque->front == -1) {
```

```
        deque->front = deque->rear = 0;
    } else if (deque->front == 0) {
        deque->front = MAX - 1;
    } else {
        deque->front--;
    }
    deque->arr[deque->front] = value;
    printf("Inserted %d at the front of the deque\n", value);
}

// Insert at rear
void insertRear(Deque* deque, int value) {
    if ((deque->front == 0 && deque->rear == MAX - 1) || (deque->front ==
deque->rear + 1)) {
        printf("Deque Overflow! Cannot insert %d at rear\n", value);
        return;
    }
    if (deque->front == -1) {
        deque->front = deque->rear = 0;
    } else if (deque->rear == MAX - 1) {
        deque->rear = 0;
    } else {
        deque->rear++;
    }
    deque->arr[deque->rear] = value;
    printf("Inserted %d at the rear of the deque\n", value);
}

// Delete from front
int deleteFront(Deque* deque) {
    if (deque->front == -1) {
        printf("Deque Underflow! Cannot delete from front\n");
        return -1;
    }
    int value = deque->arr[deque->front];
    if (deque->front == deque->rear) {
        deque->front = deque->rear = -1;
    } else if (deque->front == MAX - 1) {
        deque->front = 0;
    } else {
        deque->front++;
    }
    return value;
}

// Delete from rear
int deleteRear(Deque* deque) {
    if (deque->rear == -1) {
        printf("Deque Underflow! Cannot delete from rear\n");
        return -1;
    }
    int value = deque->arr[deque->rear];
    if (deque->front == deque->rear) {
        deque->front = deque->rear = -1;
    } else if (deque->rear == 0) {
        deque->rear = MAX - 1;
    } else {
```

```
        deque->rear--;
    }
    return value;
}

// Display deque
void displayDeque(Deque* deque) {
    if (deque->front == -1) {
        printf("Deque is empty\n");
        return;
    }
    printf("Deque elements: ");
    int i = deque->front;
    while (1) {
        printf("%d ", deque->arr[i]);
        if (i == deque->rear) break;
        i = (i + 1) % MAX;
    }
    printf("\n");
}

int main() {
    Deque deque;
    initializeDeque(&deque);

    insertRear(&deque, 10);
    insertRear(&deque, 20);
    insertFront(&deque, 5);
    displayDeque(&deque);

    printf("Deleted from front: %d\n", deleteFront(&deque));
    printf("Deleted from rear: %d\n", deleteRear(&deque));
    displayDeque(&deque);

    return 0;
}
```

Output:

```
Inserted 10 at the rear of the deque
Inserted 20 at the rear of the deque
Inserted 5 at the front of the deque
```

Q8. Implement singly linked list and perform operations such as insertion, deletion, and traversal.

Ans ➔

Code:

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

```c
// Node structure
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning
void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning\n", data);
}

// Function to insert a node at the end
void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        printf("Inserted %d at the end\n", data);
        return;
    }
    Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Inserted %d at the end\n", data);
}

// Function to delete a node by value
void deleteByValue(Node** head, int value) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete %d\n", value);
        return;
    }

    // If the value is in the head node
    if ((*head)->data == value) {
        Node* temp = *head;
        *head = (*head)->next;
        free(temp);
        printf("Deleted %d from the list\n", value);
        return;
    }

    // Traverse the list to find the node to delete
    Node* temp = *head;
```

```
    while (temp->next != NULL && temp->next->data != value) {
        temp = temp->next;
    }

    if (temp->next == NULL) {
        printf("Value %d not found in the list\n", value);
    } else {
        Node* nodeToDelete = temp->next;
        temp->next = nodeToDelete->next;
        free(nodeToDelete);
        printf("Deleted %d from the list\n", value);
    }
}

// Function to traverse and display the list
void traverse(Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    printf("Linked list elements: ");
    Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    Node* head = NULL;

    // Perform operations
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);
    traverse(head);

    deleteByValue(&head, 20);
    traverse(head);

    deleteByValue(&head, 50); // Attempt to delete non-existent value
    traverse(head);

    deleteByValue(&head, 10);
    traverse(head);

    return 0;
}
```

Output

```
Inserted 10 at the beginning
```

```
Inserted 20 at the beginning
Inserted 30 at the end
Inserted 40 at the end
Linked list elements: 20 -> 10 -> 30 -> 40 -> NULL
Deleted 20 from the list
Linked list elements: 10 -> 30 -> 40 -> NULL
Value 50 not found in the list
Linked list elements: 10 -> 30 -> 40 -> NULL
Deleted 10 from the list
Linked list elements: 30 -> 40 -> NULL
```

Q9. Extend to doubly and circularly linked lists with all operations.

Ans ➔

1. Doubly Linked List

In a **Doubly Linked List**, each node contains a `next` pointer (to the next node) and a `prev` pointer (to the previous node).

Code (Doubly Linked List)

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

// Node structure for doubly linked list
typedef struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;
    return newNode;
}

// Insert at the beginning
void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        newNode->next = *head;
        (*head)->prev = newNode;
        *head = newNode;
    }
    printf("Inserted %d at the beginning\n", data);
}

// Insert at the end
void insertAtEnd(Node** head, int data) {
```

```c
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
    } else {
        Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Inserted %d at the end\n", data);
}

// Delete a node by value
void deleteByValue(Node** head, int value) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete %d\n", value);
        return;
    }

    Node* temp = *head;
    // If the value is at the head
    if (temp->data == value) {
        *head = temp->next;
        if (*head != NULL) (*head)->prev = NULL;
        free(temp);
        printf("Deleted %d from the list\n", value);
        return;
    }

    // Traverse the list
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }

    if (temp == NULL) {
        printf("Value %d not found in the list\n", value);
        return;
    }

    // Delete the node
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    }
    free(temp);
    printf("Deleted %d from the list\n", value);
}

// Traverse the list
void traverse(Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
```

```
    }
    Node* temp = head;
    printf("Doubly Linked List elements: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    Node* head = NULL;

    // Perform operations
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);
    traverse(head);

    deleteByValue(&head, 20);
    traverse(head);

    deleteByValue(&head, 50); // Non-existent value
    traverse(head);

    deleteByValue(&head, 10);
    traverse(head);

    return 0;
}
```

Output (Doubly Linked List)

```
Inserted 10 at the beginning
Inserted 20 at the beginning
Inserted 30 at the end
Inserted 40 at the end
Doubly Linked List elements: 20 <-> 10 <-> 30 <-> 40 <-> NULL
Deleted 20 from the list
Doubly Linked List elements: 10 <-> 30 <-> 40 <-> NULL
Value 50 not found in the list
Doubly Linked List elements: 10 <-> 30 <-> 40 <-> NULL
Deleted 10 from the list
Doubly Linked List elements: 30 <-> 40 <-> NULL
```

---

2. Circular Linked List

In a **Circular Linked List**, the last node points back to the head, forming a circular structure.

Code (Circular Linked List)

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

```c
// Node structure for circular linked list
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Insert at the beginning
void insertAtBeginning(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        newNode->next = *head;  // Points to itself
    } else {
        Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
        *head = newNode;
    }
    printf("Inserted %d at the beginning\n", data);
}

// Insert at the end
void insertAtEnd(Node** head, int data) {
    Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        newNode->next = *head;  // Points to itself
    } else {
        Node* temp = *head;
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = *head;
    }
    printf("Inserted %d at the end\n", data);
}

// Delete a node by value
void deleteByValue(Node** head, int value) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete %d\n", value);
        return;
    }
```

```c
    Node* temp = *head;
    Node* prev = NULL;

    // If the head node is to be deleted
    if (temp->data == value) {
        prev = *head;
        while (prev->next != *head) {
            prev = prev->next;
        }
        if (*head == (*head)->next) {
            free(*head);
            *head = NULL;
        } else {
            prev->next = temp->next;
            *head = temp->next;
            free(temp);
        }
        printf("Deleted %d from the list\n", value);
        return;
    }

    // Traverse the list to find the node to delete
    prev = temp;
    temp = temp->next;
    while (temp != *head && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == *head) {
        printf("Value %d not found in the list\n", value);
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Deleted %d from the list\n", value);
}

// Traverse the list
void traverse(Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    Node* temp = head;
    printf("Circular Linked List elements: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(head)\n");
}

int main() {
    Node* head = NULL;
```

```
    // Perform operations
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);
    insertAtEnd(&head, 40);
    traverse(head);

    deleteByValue(&head, 20);
    traverse(head);

    deleteByValue(&head, 50); // Non-existent value
    traverse(head);

    deleteByValue(&head, 10);
    traverse(head);

    return 0;
}
```

Output (Circular Linked List)

```
Inserted 10 at the beginning
Inserted 20 at the beginning
Inserted 30 at the end
Inserted 40 at the end
Circular Linked List elements: 20 -> 10 -> 30 -> 40 -> (head)
Deleted 20 from the list
Circular Linked List elements: 10 -> 30 -> 40 -> (head)
Value 50 not found in the list
Circular Linked List elements: 10 -> 30 -> 40 -> (head)
Deleted 10 from the list
Circular Linked List elements: 30 -> 40 -> (head)
```

Q10. Implement Linear Search and Binary Search algorithms.

Ans➔

Code:

```
#include <stdio.h>

// Function for Linear Search
int linearSearch(int arr[], int size, int target) {
    for (int i = 0; i < size; i++) {
        if (arr[i] == target) {
            return i; // Target found, return index
        }
    }
    return -1; // Target not found
}

// Function for Binary Search (assuming array is sorted)
int binarySearch(int arr[], int size, int target) {
```

```
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if target is at mid
        if (arr[mid] == target) {
            return mid; // Target found at mid
        }

        // If target is smaller, search in the left half
        if (arr[mid] > target) {
            right = mid - 1;
        }
        // If target is larger, search in the right half
        else {
            left = mid + 1;
        }
    }
    return -1; // Target not found
}

int main() {
    int arr[] = {3, 5, 7, 9, 11, 13, 15, 17, 19};
    int size = sizeof(arr) / sizeof(arr[0]);
    int target = 13;

    // Perform Linear Search
    int linearResult = linearSearch(arr, size, target);
    if (linearResult != -1) {
        printf("Linear Search: Element %d found at index %d\n", target,
linearResult);
    } else {
        printf("Linear Search: Element %d not found\n", target);
    }

    // Perform Binary Search
    int binaryResult = binarySearch(arr, size, target);
    if (binaryResult != -1) {
        printf("Binary Search: Element %d found at index %d\n", target,
binaryResult);
    } else {
        printf("Binary Search: Element %d not found\n", target);
    }

    return 0;
}
```

Output

For the given sorted array `{3, 5, 7, 9, 11, 13, 15, 17, 19}` and target value `13`, the output will be:

```
Linear Search: Element 13 found at index 5
Binary Search: Element 13 found at index 5
```

Q11 . Implement and compare Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, and Merge Sort.

Ans ➔ Code :

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

// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// **Selection Sort** Implementation
void selectionSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        int minIndex = i;
        for (int j = i + 1; j < size; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        // Swap the found minimum element with the first element
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

// **Bubble Sort** Implementation
void bubbleSort(int arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap the elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

// **Insertion Sort** Implementation
void insertionSort(int arr[], int size) {
    for (int i = 1; i < size; i++) {
        int key = arr[i];
        int j = i - 1;
```

```
        // Move elements of arr[0..i-1] that are greater than key to one
position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

// **Quick Sort** Implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partitioning index
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // Swap arr[i + 1] and arr[high] (pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        int pi = i + 1;

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// **Merge Sort** Implementation
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int L[n1], R[n2];

    // Copy data to temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++) L[i] = arr[left + i];
    for (int j = 0; j < n2; j++) R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    // Merge the temp arrays back into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        } else {
            arr[k] = R[j];
```

```c
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[], if any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[], if any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}

int main() {
    int arr1[] = {64, 34, 25, 12, 22, 11, 90};
    int size = sizeof(arr1) / sizeof(arr1[0]);

    int arr2[size];
    int arr3[size];
    int arr4[size];
    int arr5[size];
    int arr6[size];

    // Make copies of the original array for different sorting algorithms
    for (int i = 0; i < size; i++) {
        arr2[i] = arr1[i];
        arr3[i] = arr1[i];
        arr4[i] = arr1[i];
        arr5[i] = arr1[i];
        arr6[i] = arr1[i];
    }

    // Measure time for Selection Sort
    clock_t start = clock();
    selectionSort(arr2, size);
    clock_t end = clock();
    printf("Selection Sort: ");
    printArray(arr2, size);
    printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);
```

```
    // Measure time for Bubble Sort
    start = clock();
    bubbleSort(arr3, size);
    end = clock();
    printf("Bubble Sort: ");
    printArray(arr3, size);
    printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    // Measure time for Insertion Sort
    start = clock();
    insertionSort(arr4, size);
    end = clock();
    printf("Insertion Sort: ");
    printArray(arr4, size);
    printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    // Measure time for Quick Sort
    start = clock();
    quickSort(arr5, 0, size - 1);
    end = clock();
    printf("Quick Sort: ");
    printArray(arr5, size);
    printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    // Measure time for Merge Sort
    start = clock();
    mergeSort(arr6, 0, size - 1);
    end = clock();
    printf("Merge Sort: ");
    printArray(arr6, size);
    printf("Time: %.6f seconds\n", (double)(end - start) / CLOCKS_PER_SEC);

    return 0;
}
```

Output :

```
Selection Sort: 11 12 22 25 34 64 90
Time: 0.000003 seconds
Bubble Sort: 11 12 22 25 34 64 90
Time: 0.000004 seconds
Insertion Sort: 11 12 22 25 34 64 90
Time: 0.000003 seconds
Quick Sort: 11 12 22 25 34 64 90
Time: 0.000002 seconds
Merge Sort: 11 12 22 25 34 64 90
Time: 0.000003 seconds
```

Q12. Implement the creation of a binary tree and perform traversals (Inorder,Preorder,Postorder).

Ans➔

Binary Tree Traversal Implementation

In a binary tree, each node has at most two children, and the traversals visit each node in a particular order:

- **Inorder Traversal**: Left -> Root -> Right
- **Preorder Traversal**: Root -> Left -> Right
- **Postorder Traversal**: Left -> Right -> Root

Code:

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

// Structure for a binary tree node
struct Node {
    int data;
    struct Node *left, *right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// **Inorder Traversal** (Left -> Root -> Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);  // Traverse left subtree
        printf("%d ", root->data);     // Visit root
        inorderTraversal(root->right); // Traverse right subtree
    }
}

// **Preorder Traversal** (Root -> Left -> Right)
void preorderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);     // Visit root
        preorderTraversal(root->left);  // Traverse left subtree
        preorderTraversal(root->right); // Traverse right subtree
    }
}

// **Postorder Traversal** (Left -> Right -> Root)
void postorderTraversal(struct Node* root) {
    if (root != NULL) {
```

```
        postorderTraversal(root->left);  // Traverse left subtree
        postorderTraversal(root->right); // Traverse right subtree
        printf("%d ", root->data);       // Visit root
    }
}

int main() {
    // Create the binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    // Print the traversals
    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

    printf("Preorder Traversal: ");
    preorderTraversal(root);
    printf("\n");

    printf("Postorder Traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output

For the binary tree:

```
        1
       / \
      2   3
     / \ / \
    4  5 6  7
```

The output will be:

yaml
कोड कॉपी करणे

```
Inorder Traversal: 4 2 5 1 6 3 7
Preorder Traversal: 1 2 4 5 3 6 7
Postorder Traversal: 4 5 2 6 7 3 1
```

Q13. Implement a binary search tree (BST) with insertion, deletion, and searching operations.

Ans➔

Binary Search Tree (BST) Operations

1. **Insertion**: Insert a new node while maintaining the BST property (left child < parent < right child).
2. **Search**: Search for a value in the tree.
3. **Deletion**: Delete a node and restructure the tree while maintaining the BST property.

Code:

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

// Structure for a binary tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// **Insertion** into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return newNode(data);  // If tree is empty, create a new node
    }

    if (data < root->data) {
        root->left = insert(root->left, data);  // Insert in the left subtree
    } else if (data > root->data) {
        root->right = insert(root->right, data);  // Insert in the right
subtree
    }

    return root;  // Return the unchanged node pointer
}

// **Search** for a node in the BST
struct Node* search(struct Node* root, int data) {
```

```
    if (root == NULL || root->data == data) {
        return root;  // Return NULL if not found, or the node if found
    }

    if (data < root->data) {
        return search(root->left, data);  // Search in the left subtree
    }

    return search(root->right, data);  // Search in the right subtree
}

// **Find the minimum node** in the BST (used for deletion)
struct Node* findMin(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;  // Keep going left until the minimum node is
found
    }
    return root;
}

// **Delete a node from the BST**
struct Node* delete(struct Node* root, int data) {
    if (root == NULL) {
        return root;  // If tree is empty, return NULL
    }

    if (data < root->data) {
        root->left = delete(root->left, data);  // Search for the node to
delete in the left subtree
    } else if (data > root->data) {
        root->right = delete(root->right, data);  // Search for the node to
delete in the right subtree
    } else {
        // Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Node with two children: Get the inorder successor (smallest in the
right subtree)
        struct Node* temp = findMin(root->right);

        // Copy the inorder successor's content to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = delete(root->right, temp->data);
    }

    return root;
}
```

```
// **Inorder Traversal** (Left -> Root -> Right)
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);  // Traverse left subtree
        printf("%d ", root->data);     // Visit root
        inorderTraversal(root->right); // Traverse right subtree
    }
}

int main() {
    struct Node* root = NULL;

    // Inserting nodes into the BST
    root = insert(root, 50);
    root = insert(root, 30);
    root = insert(root, 20);
    root = insert(root, 40);
    root = insert(root, 70);
    root = insert(root, 60);
    root = insert(root, 80);

    // Print the inorder traversal of the BST
    printf("Inorder Traversal (BST): ");
    inorderTraversal(root);
    printf("\n");

    // Search for a node
    int searchValue = 40;
    struct Node* result = search(root, searchValue);
    if (result != NULL) {
        printf("Node with value %d found in the BST.\n", searchValue);
    } else {
        printf("Node with value %d not found in the BST.\n", searchValue);
    }

    // Delete a node
    int deleteValue = 20;
    root = delete(root, deleteValue);
    printf("Inorder Traversal after deleting %d: ", deleteValue);
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

Output Example

For the BST:

```
        50
       /  \
      30   70
     /  \  /  \
```

```
    20  40 60  80
```

The output will be:

```
Inorder Traversal (BST): 20 30 40 50 60 70 80
Node with value 40 found in the BST.
Inorder Traversal after deleting 20: 30 40 50 60 70 80
```

Q14. Implement a heap data structure (min-heap or max-heap) and demonstrate heap insertion and deletion.

Ans➔

Code:

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

#define MAX_SIZE 100

// Structure for a Min-Heap
struct MinHeap {
    int arr[MAX_SIZE];
    int size;
};

// Function to initialize the Min-Heap
void initHeap(struct MinHeap* heap) {
    heap->size = 0;
}

// Function to swap two elements in the heap
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to get the index of the left child of a node
int leftChild(int i) {
    return 2 * i + 1;
}

// Function to get the index of the right child of a node
int rightChild(int i) {
    return 2 * i + 2;
}

// Function to get the index of the parent of a node
int parent(int i) {
    return (i - 1) / 2;
}
```

```c
// Function to heapify the Min-Heap (bubble down)
void minHeapify(struct MinHeap* heap, int i) {
    int left = leftChild(i);
    int right = rightChild(i);
    int smallest = i;

    // Check if left child is smaller than the current smallest
    if (left < heap->size && heap->arr[left] < heap->arr[smallest]) {
        smallest = left;
    }

    // Check if right child is smaller than the current smallest
    if (right < heap->size && heap->arr[right] < heap->arr[smallest]) {
        smallest = right;
    }

    // If the smallest is not the current node, swap and continue heapifying
    if (smallest != i) {
        swap(&heap->arr[i], &heap->arr[smallest]);
        minHeapify(heap, smallest);
    }
}

// Function to insert an element into the Min-Heap
void insert(struct MinHeap* heap, int value) {
    if (heap->size == MAX_SIZE) {
        printf("Heap is full\n");
        return;
    }

    // Insert the new value at the end of the heap
    heap->arr[heap->size] = value;
    int i = heap->size;
    heap->size++;

    // Bubble up to maintain the heap property
    while (i != 0 && heap->arr[parent(i)] > heap->arr[i]) {
        swap(&heap->arr[i], &heap->arr[parent(i)]);
        i = parent(i);
    }
}

// Function to delete the root (minimum element) from the Min-Heap
int deleteMin(struct MinHeap* heap) {
    if (heap->size <= 0) {
        printf("Heap is empty\n");
        return -1;
    }

    if (heap->size == 1) {
        heap->size--;
        return heap->arr[0];
    }

    // Get the minimum element (root)
    int root = heap->arr[0];
```

```
    // Replace the root with the last element
    heap->arr[0] = heap->arr[heap->size - 1];
    heap->size--;

    // Heapify the root element to maintain the heap property
    minHeapify(heap, 0);

    return root;
}

// Function to print the elements of the heap
void printHeap(struct MinHeap* heap) {
    for (int i = 0; i < heap->size; i++) {
        printf("%d ", heap->arr[i]);
    }
    printf("\n");
}

int main() {
    struct MinHeap heap;
    initHeap(&heap);

    // Inserting elements into the Min-Heap
    insert(&heap, 10);
    insert(&heap, 15);
    insert(&heap, 20);
    insert(&heap, 17);
    insert(&heap, 8);
    insert(&heap, 5);

    printf("Min-Heap after insertion: ");
    printHeap(&heap);

    // Deleting the root (minimum) element
    printf("Deleted min element: %d\n", deleteMin(&heap));

    printf("Min-Heap after deletion: ");
    printHeap(&heap);

    return 0;
}
```

Output Example

For the operations:

- Insert: 10, 15, 20, 17, 8, 5
- Delete root (minimum element)

The output will be:

```
Min-Heap after insertion: 5 8 20 17 15 10
Deleted min element: 5
Min-Heap after deletion: 8 10 20 17 15
```