Q.1) Sorting Algorithms Non-Recursive

Bubble Sort

Algorithm

1. Start with the first element in the array.
2. Compare the current element with the next element.
3. If the current element is greater, swap them.
4. Repeat steps 2-3 for all elements until the end of the array.
5. Repeat steps 1-4 for n−1n-1n−1 iterations.
6. End.

Code

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);

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

    return 0;
}
```

Output

```
Developer/dsa/q1 via  v14.2.1-gcc
❯ ./bubble
Enter the number of elements: 5
Enter 5 elements: 1 7 4 9 0
Sorted array: 0 1 4 7 9
```

Insertion Sort

Algorithm

1. Start with the second element as the key.
2. Compare the key with the elements before it.
3. Shift all larger elements one position to the right.
4. Insert the key into its correct position.
5. Repeat steps 1-4 for all elements.
6. End.

Code

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];
```

```c
    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);

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

    return 0;
}
```

Output

```
Developer/dsa/q1 via  v14.2.1-gcc
❯ ./insertion
Enter the number of elements: 6
Enter 6 elements: 9 7 4 5 1 3
Sorted array: 1 3 4 5 7 9
```

Selection Sort

Algorithm

1. Start with the first element as the minimum.
2. Compare the minimum with the rest of the elements.
3. If a smaller element is found, update the minimum.
4. Swap the minimum element with the first element.
5. Repeat steps 1-4 for the remaining unsorted elements.
6. End.

Code

```c
#include <stdio.h>

void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        int temp = arr[minIdx];
        arr[minIdx] = arr[i];
        arr[i] = temp;
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    selectionSort(arr, n);

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

    return 0;
}
```

Output

```
Developer/dsa/q1 via  v14.2.1-gcc
❯ ./selection
Enter the number of elements: 7
Enter 7 elements: 1 5 3 54 8 98 45
Sorted array: 1 3 5 8 45 54 98
```

Q.2) Sorting Algorithms Recursive

Recursive Bubble Sort

Algorithm

1. If the array has only one element, return (base case).
2. Perform one pass of Bubble Sort by comparing adjacent elements and swapping them if needed.
3. Recursively call the algorithm for the first n−1n-1n−1 elements.
4. End.

Code

```c
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    if (n == 1) return; // Base case

    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            int temp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = temp;
        }
    }

    // Recursive call for the remaining array
    bubbleSort(arr, n - 1);
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);

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

    return 0;
}
```

Output

```
Developer/dsa/q2 via  v14.2.1-gcc
❯ ./rbubble
Enter the number of elements: 5
Enter 5 elements: 3 2 78 45 20
Sorted array: 2 3 20 45 78
```

Recursive Insertion Sort

Algorithm

1. If the array size is 1, return (base case).
2. Recursively sort the first n−1n-1n−1 elements.
3. Insert the last element into its correct position by comparing it with the sorted portion.
4. End.

Code

```c
#include <stdio.h>

void insertionSort(int arr[], int n) {
    if (n <= 1) return; // Base case

    // Recursive call for the first n-1 elements
    insertionSort(arr, n - 1);

    // Insert the nth element in its correct position
    int key = arr[n - 1];
    int j = n - 2;

    while (j >= 0 && arr[j] > key) {
        arr[j + 1] = arr[j];
        j--;
    }
    arr[j + 1] = key;
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

Output

```
Developer/dsa/q2 via  v14.2.1-gcc
❯ ./rinsertion
Enter the number of elements: 6
Enter 6 elements: 12 32 25 67 98 76
Sorted array: 12 25 32 67 76 98
```

Recursive Selection Sort

Algorithm

1. If the starting index is the last index, return (base case).
2. Find the index of the smallest element in the unsorted portion of the array.
3. Swap it with the element at the current index.
4. Recursively call the algorithm for the next index.
5. End.

Code

```c
#include <stdio.h>

void selectionSort(int arr[], int n, int index) {
    if (index == n - 1) return; // Base case

    int minIdx = index;
    for (int i = index + 1; i < n; i++) {
        if (arr[i] < arr[minIdx]) {
            minIdx = i;
        }
    }

    // Swap the minimum element with the current index
    int temp = arr[minIdx];
    arr[minIdx] = arr[index];
    arr[index] = temp;

    // Recursive call for the remaining array
    selectionSort(arr, n, index + 1);
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    int arr[n];

    printf("Enter %d elements: ", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    selectionSort(arr, n, 0);

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

    return 0;
}
```

Output

```
Developer/dsa/q2 via  v14.2.1-gcc
❯ ./rselection
Enter the number of elements: 4
Enter 4 elements: 34 98 76 16
Sorted array: 16 34 76 98
```

Q.3) Searching Algorithms

Linear Search

Algorithm

1. **Input**: Take an array `arr[]` of size `n` and a `target` element to search.
2. Start from the first element of the array (`arr[0]`).
3. Compare the current element (`arr[i]`) with the `target`.
4. If `arr[i] == target`, return the index `i` where the element is found.
5. If the target is not found, move to the next element (increment `i`).
6. Repeat steps 3-5 until either the target is found or all elements have been checked.
7. If the target is not found after checking all elements, return `-1` to indicate that the element is not present.

Code

```c
#include <stdio.h>

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

int main() {
    int size, target;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter %d elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the target element to search: ");
    scanf("%d", &target);

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

    return 0;
}
```

Output

```
Developer/dsa/q3 via  v14.2.1-gcc
❯ ./linearsearch
Enter the size of the array: 6
Enter 6 elements:
23 87 12 48 90 84
Enter the target element to search: 48
Element found at index 3

Developer/dsa/q3 via  v14.2.1-gcc took 24s
```

```
❯ ./linearsearch
Enter the size of the array: 5
Enter 5 elements:
87 4 32 98 17
Enter the target element to search: 15
Element not found
```

Binary Search

Algorithm

1. **Input**: Take a sorted array `arr[]`, the size `n`, and the `target` element to search.
2. Initialize two pointers: `left = 0` and `right = n - 1` (indices of the first and last elements of the array).
3. While `left <= right`:
    - Calculate the middle index `mid = (left + right) / 2`.
    - Compare `arr[mid]` with the `target`.
        - If `arr[mid] == target`, return `mid`.
        - If `arr[mid] > target`, search the left half by updating `right = mid - 1`.
        - If `arr[mid] < target`, search the right half by updating `left = mid + 1`.
4. If `left > right`, the target is not in the array, so return `-1`.

Code

```c
#include <stdio.h>

int binarySearch(int arr[], int size, int target) {
    int left = 0, right = size - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (arr[mid] == target) {
            return mid;  // Element found at mid
        }
        if (arr[mid] > target) {
            right = mid - 1;  // Search in left half
        } else {
            left = mid + 1;  // Search in right half
        }
    }
    return -1;  // Element not found
}

int main() {
    int size, target;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter %d sorted elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the target element to search: ");
    scanf("%d", &target);

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

    return 0;
}
```

Output

```
Developer/dsa/q3 via 🅒 v14.2.1-gcc
❯ ./binarysearch
Enter the size of the array: 4
Enter 4 sorted elements:
23 69 80 76
Enter the target element to search: 34
Element not found

Developer/dsa/q3 via 🅒 v14.2.1-gcc took 15s
❯ ./binarysearch
Enter the size of the array: 5
Enter 5 sorted elements:
37 65 45 61 83
Enter the target element to search: 45
Element found at index 2
```

Recursive Binary Search

Algorithm

1. **Input**: Take a sorted array `arr[]`, the left index `left`, the right index `right`, and the `target` element to search.

2. Calculate the middle index `mid = (left + right) / 2`.
3. Compare `arr[mid]` with the `target`.
    - If `arr[mid] == target`, return `mid`.
    - If `arr[mid] > target`, recursively search the left half of the array (i.e., search between `left` and `mid-1`).
    - If `arr[mid] < target`, recursively search the right half of the array (i.e., search between `mid+1` and `right`).
4. If `left` becomes greater than `right`, the target is not in the array, so return `-1`.

Code

```c
#include <stdio.h>

int binarySearch(int arr[], int left, int right, int target) {
    if (left > right) {
        return -1;  // Element not found
    }

    int mid = left + (right - left) / 2;

    if (arr[mid] == target) {
        return mid;  // Element found at mid
    }
    if (arr[mid] > target) {
        return binarySearch(arr, left, mid - 1, target);  // Search in
left half
    }
    return binarySearch(arr, mid + 1, right, target);  // Search in right
half
}

int main() {
    int size, target;

    printf("Enter the size of the array: ");
    scanf("%d", &size);

    int arr[size];
    printf("Enter %d sorted elements:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    printf("Enter the target element to search: ");
    scanf("%d", &target);

    int result = binarySearch(arr, 0, size - 1, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

Output

```
Developer/dsa/q3 via Ⓒ v14.2.1-gcc
❯ ./rbinarysearch
Enter the size of the array: 5
Enter 5 sorted elements:
23 43 56 78 90
Enter the target element to search: 43
Element found at index 1

Developer/dsa/q3 via Ⓒ v14.2.1-gcc took 11s
❯ ./rbinarysearch
Enter the size of the array: 4
Enter 4 sorted elements:
57 78 90 98
Enter the target element to search: 76
Element not found
```

Q.4) Implementation of Stack Using Array

Algorithm

1. **Initialization:**
   - Define a `Stack` structure with an integer array `arr[]` of size `MAX` and an integer `top` to track the top of the stack.
   - Initialize `top = -1` to indicate an empty stack.
2. **Push Operation (Continuous until -1):**
   - Check if the stack is full (`top == MAX - 1`):
     - If full, print "Stack overflow! Unable to push more elements" and exit the push loop.
   - Otherwise, enter a loop where:
     - Prompt the user to enter a value to push.
     - If the user enters `-1`, exit the loop (stop pushing).
     - If the value is not `-1`, increment `top` and insert the value at `stack->arr[top]`.
     1. Print a message indicating the value has been pushed to the stack.
3. **Pop Operation:**

- Check if the stack is empty (`top == -1`):
    - If empty, print "Stack underflow! Unable to pop" and return `-1`.
- Otherwise, print and return the element at `stack->arr[top]` and decrement `top`.

4. **Peek Operation:**
    - Check if the stack is empty (`top == -1`):
        - If empty, print "Stack is empty" and return `-1`.
    - Otherwise, return the element at `stack->arr[top]` (the top element of the stack).
5. **Display Operation:**
    - Check if the stack is empty (`top == -1`):
        - If empty, print "Stack is empty".
    - Otherwise, print all elements from `stack->arr[0]` to `stack->arr[top]`.
6. **Exit Operation:**
    - Exit the program.

Code

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

#define MAX 10  // Define the maximum size of the stack

// Structure to represent a stack
struct Stack {
    int arr[MAX];
    int top;
};

// Function to initialize the stack
void initStack(struct Stack* stack) {
    stack->top = -1;  // Stack is empty initially
}

// Function to check if the stack is full
int isFull(struct Stack* stack) {
    return stack->top == MAX - 1;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack* stack) {
    int value;
    while (1) {
        if (isFull(stack)) {
            printf("Stack overflow! Unable to push more elements.\n");
            return;  // Exit the loop if the stack is full
        }

        printf("Enter value to push (or -1 to stop pushing): ");
        scanf("%d", &value);
        if (value == -1) {
            return;  // Exit the loop if user inputs -1
        }

        stack->arr[++stack->top] = value;  // Increment top and insert
element
        printf("%d pushed to stack\n", value);
    }
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow! Unable to pop\n");
        return -1;  // Indicating stack is empty
    } else {
        return stack->arr[stack->top--];  // Return the element and
decrement top
    }
}

// Function to get the top element of the stack without popping
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty\n");
        return -1;  // Indicating stack is empty
    } else {
        return stack->arr[stack->top];  // Return the top element
    }
}

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

```
// Main function to test stack operations
int main() {
    struct Stack stack;
    initStack(&stack);  // Initialize the stack

    int choice, value;

    while (1) {
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                push(&stack);  // Push elements until user decides to
stop
                break;
            case 2:
                value = pop(&stack);
                if (value != -1) {
                    printf("Popped element: %d\n", value);
                }
                break;
            case 3:
                value = peek(&stack);
                if (value != -1) {
                    printf("Top element is: %d\n", value);
                }
                break;
            case 4:
                display(&stack);
                break;
            case 5:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Developer/dsa/q4 via  v14.2.1-gcc
❯ ./stack

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push (or -1 to stop pushing): 23
23 pushed to stack
Enter value to push (or -1 to stop pushing): 32
32 pushed to stack
Enter value to push (or -1 to stop pushing): 67
67 pushed to stack
Enter value to push (or -1 to stop pushing): 89
89 pushed to stack
Enter value to push (or -1 to stop pushing): 90
90 pushed to stack
Enter value to push (or -1 to stop pushing): -1

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 23 32 67 89 90

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element is: 90

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
```

```
Popped element: 90

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 23 32 67 89

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting program.
```

Q.5) Implementation of Queue Using Array

Algorithm

1. **Initialization:**
   - Create an array `queue[]` of size `MAX`.
   - Initialize `front = -1` and `rear = -1`.
2. **Enqueue Operation:**
   - Check if the queue is full (`rear == MAX - 1`):
     - If full, print "Queue Overflow!"
     - Else, if the queue is empty (`front == -1`), set `front = 0`.
     - Increment `rear` and insert `value` at `queue[rear]`.
3. **Dequeue Operation:**
   - Check if the queue is empty (`front == -1` or `front > rear`):
     - If empty, print "Queue Underflow!"
     - Else, print and remove the element at `queue[front]` and increment `front`.
4. **Peek Operation:**
   - If the queue is empty (`front == -1` or `front > rear`), print "Queue is empty".
   - Otherwise, print the element at `queue[front]`.
5. **Display Operation:**
   - If the queue is empty, print "Queue is empty".
   - Else, print all elements from `queue[front]` to `queue[rear]`.
6. **Exit Operation:**
   - Exit the program.

Code

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

#define MAX 5 // Define the maximum size of the queue

// Queue structure with front, rear, and the array to store elements
int queue[MAX];
int front = -1, rear = -1;

// Function to check if the queue is full
int isFull() {
    return (rear == MAX - 1);
}

// Function to check if the queue is empty
int isEmpty() {
    return (front == -1 || front > rear);
}

// Function to add an element to the queue (Enqueue operation)
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
        if (front == -1) { // If queue is empty, set front to 0
            front = 0;
        }
        rear++;
        queue[rear] = value; // Insert element at rear
        printf("Enqueued: %d\n", value);
    }
}

// Function to remove an element from the queue (Dequeue operation)
void dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow! No elements to dequeue.\n");
    } else {
        printf("Dequeued: %d\n", queue[front]);
        front++; // Increment front to remove the element
    }
}

// Function to peek at the front element of the queue
void peek() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Front element: %d\n", queue[front]);
    }
}

// Function to display all elements in the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, value;

    while (1) {
        // Display menu for the user
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");

        // Read user's choice
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Enqueue (Push)
                do {
                    printf("Enter value to enqueue (or -1 to stop): ");
                    scanf("%d", &value);
                    if (value != -1) {
                        enqueue(value);
                    }
                } while (value != -1);
                break;

            case 2: // Dequeue
                dequeue();
                break;

            case 3: // Peek
```

```c
                peek();
                break;

            case 4: // Display
                display();
                break;

            case 5: // Exit
                printf("Exiting program.\n");
                exit(0);
                break;

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Developer/dsa/q5 via C v14.2.1-gcc
❯ ./queue

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to enqueue (or -1 to stop): 34
Enqueued: 34
Enter value to enqueue (or -1 to stop): 32
Enqueued: 32
Enter value to enqueue (or -1 to stop): 87
Enqueued: 87
Enter value to enqueue (or -1 to stop): 59
Enqueued: 59
Enter value to enqueue (or -1 to stop): -1

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
```

```
Enter your choice: 4
Queue elements: 34 32 87 59

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element: 34

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued: 34

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 32 87 59

Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting program.
```

Q.6) Implementation of Circular Queue Using Array

Algorithm

1. **Initialization:**
   - Define a queue array `queue[MAX]` to store elements.
   - Define `front` and `rear` to keep track of the front and rear positions of the queue. Initialize both as `-1` (indicating an empty queue).
2. **Function:** `isFull()`

- Return `1` if the queue is full (i.e., `(front == (rear + 1) % MAX)`), else return `0`.

3. **function:** `isempty()`
    - return `1` if the queue is empty (i.e., `front == -1`), else return `0`.
4. **Function:** `enqueue(value)`
    - Check if the queue is full by calling `isFull()`.
        - If full, print "Queue Overflow! Cannot enqueue" and return.
    - If the queue is empty (`front == -1`), initialize `front = 0`.
    - Increment `rear` using circular increment: `rear = (rear + 1) % MAX`.
    - Insert the value into the queue at `queue[rear]`.
    - Print "Enqueued: value".
5. **Function:** `dequeue()`
    - Check if the queue is empty by calling `isEmpty()`.
        - If empty, print "Queue Underflow! No elements to dequeue" and return.
    - Print and remove the element at the `front` position.
    - If the queue has only one element (`front == rear`), reset `front` and `rear` to `-1` (empty queue).
    - Otherwise, increment `front` using circular increment: `front = (front + 1) % MAX`.
6. **Function:** `peek()`
    - Check if the queue is empty by calling `isEmpty()`.
        - If empty, print "Queue is empty".
    - Otherwise, print the element at the `front` of the queue.
7. **Function:** `display()`
    - Check if the queue is empty by calling `isEmpty()`.
        - If empty, print "Queue is empty".
    - Otherwise, display all elements in the queue:
        - Start from `front` and print each element until reaching `rear`.
        - Use circular increment `(i + 1) % MAX` to traverse the queue.
8. **Main Program Loop:**
    - Display a menu with the following options:
        - 1. Enqueue
        - 2. Dequeue
        - 3. Peek
        - 4. Display
        - 5. Exit
    - Based on user input, perform the corresponding operation:
        - **Enqueue:** Prompt the user to input a value and enqueue it. Ask if they want to enqueue more values.
        - **Dequeue:** Remove and display the front element.
        - **Peek:** Display the front element.

- **Display:** Show all elements in the queue.
- **Exit:** Exit the program.

Code

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

#define MAX 5 // Define the maximum size of the queue

// Queue structure with front, rear, and the array to store elements
int queue[MAX];
int front = -1, rear = -1;

// Function to check if the queue is full
int isFull() {
    return (front == (rear + 1) % MAX);
}

// Function to check if the queue is empty
int isEmpty() {
    return (front == -1);
}

// Function to add an element to the queue (Enqueue operation)
void enqueue(int value) {
    if (isFull()) {
        printf("Queue Overflow! Cannot enqueue %d\n", value);
    } else {
        if (front == -1) { // If queue is empty, initialize front and
rear to 0
            front = 0;
        }
        rear = (rear + 1) % MAX; // Circular increment
        queue[rear] = value;
        printf("Enqueued: %d\n", value);
    }
}

// Function to remove an element from the queue (Dequeue operation)
void dequeue() {
    if (isEmpty()) {
        printf("Queue Underflow! No elements to dequeue.\n");
    } else {
        printf("Dequeued: %d\n", queue[front]);
        if (front == rear) { // Only one element left in the queue
            front = rear = -1; // Reset the queue
        } else {
```

```c
            front = (front + 1) % MAX; // Circular increment
        }
    }
}

// Function to peek at the front element of the queue
void peek() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Front element: %d\n", queue[front]);
    }
}

// Function to display all elements in the queue
void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        int i = front;
        while (i != rear) {
            printf("%d ", queue[i]);
            i = (i + 1) % MAX; // Circular increment
        }
        printf("%d\n", queue[rear]); // Print the last element
    }
}

int main() {
    int choice, value;
    char continueEnqueueing;

    while (1) {
        // Display menu for the user
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");

        // Read user's choice
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Enqueue (Push)
                do {
                    printf("Enter value to enqueue: ");
```

```
          scanf("%d", &value);
          enqueue(value);

          // Ask user if they want to enqueue another element
          printf("Do you want to enqueue another element?
(y/n): ");
          getchar(); // To clear the newline character left by
scanf
          scanf("%c", &continueEnqueueing);
        } while (continueEnqueueing == 'y' || continueEnqueueing
== 'Y');
        break;

      case 2: // Dequeue
        dequeue();
        break;

      case 3: // Peek
        peek();
        break;

      case 4: // Display
        display();
        break;

      case 5: // Exit
        printf("Exiting program.\n");
        exit(0);
        break;

      default:
        printf("Invalid choice! Please try again.\n");
    }
  }

  return 0;
}
```

Output

```
Developer/dsa/q6 via  v14.2.1-gcc
❯ ./circularqueue

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
```

```
5. Exit
Enter your choice: 1
Enter value to enqueue: 32
Enqueued: 32
Do you want to enqueue another element? (y/n): y
Enter value to enqueue: 54
Enqueued: 54
Do you want to enqueue another element? (y/n): y
Enter value to enqueue: 89
Enqueued: 89
Do you want to enqueue another element? (y/n): n

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 32 54 89

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element: 32

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued: 32

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue elements: 54 89

Circular Queue Operations:
1. Enqueue
```

```
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting program.
```

Q.7) Implementation of Stack Using Linked List

Algorithm

1. **Initialization:**
   - Define a structure `Node` with fields `data` (to store the value) and `next` (to point to the next node).
   - Initialize the stack as an empty linked list (`top = NULL`).
2. **Function:** `createNode(data)`
   - Allocate memory for a new node.
   - Set the `data` of the new node to the given value and set `next` to `NULL`.
   - Return the created node.
3. **Function:** `isEmpty(top)`
   - Return `1` if the stack is empty (`top == NULL`), else return `0`.
4. **Function:** `push(top, data)`
   - Create a new node using `createNode(data)`.
   - Set the `next` of the new node to the current `top` of the stack.
   - Update the `top` to point to the new node.
   - Print "Pushed: value".
5. **Function:** `pop(top)`
   - Check if the stack is empty by calling `isEmpty(top)`.
     - If empty, print "Stack Underflow! No elements to pop".
   - If the stack is not empty:
     - Temporarily store the current `top` in `temp`.
     - Update `top` to point to the next node of the current top.
     - Print and return the `data` of the node in `temp`.
     - Free the memory of `temp`.
6. **Function:** `peek(top)`
   - Check if the stack is empty by calling `isEmpty(top)`.
     - If empty, print "Stack is empty".
   - If the stack is not empty, print the `data` of the `top` node.
7. **Function:** `display(top)`
   - Check if the stack is empty by calling `isEmpty(top)`.
     - If empty, print "Stack is empty".

- If the stack is not empty:
  - Traverse the stack starting from the `top`.
  - Print each node's `data` until reaching the end (`next == NULL`).

8. **Main Program Loop:**
   - Display a menu with the following options:
     - 1. Push
     - 2. Pop
     - 3. Peek
     - 4. Display
     - 5. Exit
   - Based on user input, perform the corresponding operation:
     - **Push:** Prompt the user to input a value and push it onto the stack. Continue pushing values until the user enters `-1`.
     - **Pop:** Remove and display the top element of the stack.
     - **Peek:** Display the top element of the stack.
     - **Display:** Show all elements in the stack.
     - **Exit:** Exit the program.

Code

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

// Structure for a node in the stack
struct Node {
    int data;
    struct Node* next;
};

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

// Function to check if the stack is empty
int isEmpty(struct Node* top) {
    return top == NULL;
}

// Function to push an element onto the stack
void push(struct Node** top, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *top;
    *top = newNode;
    printf("Pushed: %d\n", data);
}

// Function to pop an element from the stack
void pop(struct Node** top) {
    if (isEmpty(*top)) {
        printf("Stack Underflow! No elements to pop.\n");
    } else {
        struct Node* temp = *top;
        *top = (*top)->next;
        printf("Popped: %d\n", temp->data);
        free(temp);
    }
}

// Function to peek at the top element of the stack
void peek(struct Node* top) {
    if (isEmpty(top)) {
        printf("Stack is empty.\n");
    } else {
        printf("Top element: %d\n", top->data);
    }
}

// Function to display all elements in the stack
void display(struct Node* top) {
    if (isEmpty(top)) {
        printf("Stack is empty.\n");
    } else {
        struct Node* temp = top;
        printf("Stack elements: ");
        while (temp != NULL) {
            printf("%d ", temp->data);
            temp = temp->next;
        }
        printf("\n");
    }
}

int main() {
    struct Node* stack = NULL;
    int choice, value;

    while (1) {
        // Display menu for the user
        printf("\nStack Operations:\n");
        printf("1. Push\n");
        printf("2. Pop\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");

        // Read user's choice
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1: // Push continuously
                printf("Enter value to push (-1 to stop): ");
                while (1) {
                    scanf("%d", &value);
                    if (value == -1) {
                        break;
                    }
                    push(&stack, value);
                    printf("Enter next value to push (-1 to stop): ");
                }
                break;

            case 2: // Pop
                pop(&stack);
                break;

            case 3: // Peek
                peek(stack);
                break;

            case 4: // Display
                display(stack);
                break;

            case 5: // Exit
                printf("Exiting program.\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Developer/dsa/q7 via  v14.2.1-gcc
❯ ./llstack

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to push (-1 to stop): 4
Pushed: 4
Enter next value to push (-1 to stop): 67
Pushed: 67
Enter next value to push (-1 to stop): 89
Pushed: 89
Enter next value to push (-1 to stop): 70
Pushed: 70
Enter next value to push (-1 to stop): -1

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 70 89 67 4

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 70

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 2
Popped: 70

Stack Operations:
1. Push
```

```
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 3
Top element: 89

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 4
Stack elements: 89 67 4

Stack Operations:
1. Push
2. Pop
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting program.
```

Q.8) Implementation of Queue Using Linked List

Algorithm

1. **Initialization:**
    - Define a structure `Node` with fields `data` (to store the value) and `next` (to point to the next node).
    - Initialize two pointers `front` and `rear` to `NULL`, representing an empty queue.
2. **Function:** `createNode(data)`
    - Allocate memory for a new node.
    - Set the `data` field of the new node to the given value and set `next` to `NULL`.
    - Return the created node.
3. **Function:** `isEmpty(front)`
    - Return `1` if the queue is empty (`front == NULL`), else return `0`.
4. **Function:** `enqueue(front, rear, data)`
    - Create a new node using `createNode(data)`.
    - If the queue is empty (`front == NULL`):
        - Set both `front` and `rear` to point to the new node.
    - If the queue is not empty:
        - Set the `next` pointer of `rear` to the new node.

        - Update `rear` to point to the new node.
    - Print "Enqueued: data".
5. **Function:** `dequeue(front)`
    - Check if the queue is empty by calling `isEmpty(front)`.
        - If empty, print "Queue Underflow! The queue is empty".
    - If the queue is not empty:
        - Temporarily store the current `front` in `temp`.
        - Update `front` to point to the next node of the current `front`.
        - Print and return the `data` of the node in `temp`.
        - Free the memory of `temp`.
6. **Function:** `peek(front)`
    - Check if the queue is empty by calling `isEmpty(front)`.
        - If empty, print "Queue is empty".
    - If the queue is not empty, print the `data` of the `front` node.
7. **Function:** `display(front)`
    - Check if the queue is empty by calling `isEmpty(front)`.
        - If empty, print "Queue is empty".
    - If the queue is not empty:
        - Traverse the queue starting from the `front`.
        - Print each node's `data` until reaching the end (`next == NULL`).
8. **Main Program Loop:**
    - Display a menu with the following options:
        - 1. Enqueue
        - 2. Dequeue
        - 3. Peek
        - 4. Display
        - 5. Exit
    - Based on user input, perform the corresponding operation:
        - **Enqueue:** Prompt the user to input a value and enqueue it to the queue. Continue enqueuing values until the user enters `-1`.
        - **Dequeue:** Remove and display the front element of the queue.
        - **Peek:** Display the front element of the queue.
        - **Display:** Show all elements in the queue.
        - **Exit:** Exit the program.

Code

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

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

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

// Function to check if the queue is empty
int isEmpty(struct Node* front) {
    return (front == NULL);
}

// Enqueue operation
void enqueue(struct Node** front, struct Node** rear, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(*front)) {
        *front = *rear = newNode;
    } else {
        (*rear)->next = newNode;
        *rear = newNode;
    }
    printf("Enqueued: %d\n", data);
}

// Dequeue operation
void dequeue(struct Node** front) {
    if (isEmpty(*front)) {
        printf("Queue Underflow! The queue is empty.\n");
        return;
    }
    struct Node* temp = *front;
    *front = (*front)->next;
    printf("Dequeued: %d\n", temp->data);
    free(temp);
}

// Peek operation (view front element without removing it)
void peek(struct Node* front) {
    if (isEmpty(front)) {
        printf("Queue is empty.\n");
    } else {
        printf("Front element: %d\n", front->data);
    }
```

```
}

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

int main() {
    struct Node* front = NULL;
    struct Node* rear = NULL;
    int choice, data;

    while (1) {
        // Display menu to user
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                // Enqueue operation (Continuous input until -1 is
entered)
                while (1) {
                    printf("Enter value to enqueue (Enter -1 to stop):
");
                    scanf("%d", &data);
                    if (data == -1) {
                        break; // Exit loop when -1 is entered
                    }
                    enqueue(&front, &rear, data);
                }
                break;

            case 2:
                // Dequeue operation
                dequeue(&front);
                break;

            case 3:
                // Peek operation
                peek(front);
                break;

            case 4:
                // Display the queue
                display(front);
                break;

            case 5:
                // Exit
                printf("Exiting the program.\n");
                exit(0);

            default:
                printf("Invalid choice! Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Developer/dsa/q8 via  v14.2.1-gcc
❯ ./llqueue

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 1
Enter value to enqueue (Enter -1 to stop): 68
Enqueued: 68
Enter value to enqueue (Enter -1 to stop): 90
Enqueued: 90
Enter value to enqueue (Enter -1 to stop): 83
Enqueued: 83
Enter value to enqueue (Enter -1 to stop): 23
Enqueued: 23
Enter value to enqueue (Enter -1 to stop): 76
Enqueued: 76
```

```
Enter value to enqueue (Enter -1 to stop): -1

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue: 68 90 83 23 76

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element: 68

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 2
Dequeued: 68

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 4
Queue: 90 83 23 76

Queue Operations Menu:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Exit
Enter your choice: 3
Front element: 90

Queue Operations Menu:
1. Enqueue
2. Dequeue
```

```
3. Peek
4. Display
5. Exit
Enter your choice: 5
Exiting the program.
```

Q.9) Implementation of Circular Queue Using Linked List

Algorithm

1. **Initialization:**

- Define a structure `Node` with the following fields:
  - `data`: to store the value of the node.
  - `next`: to point to the next node in the queue.
- Define a structure `CircularQueue` with the following fields:
  - `front`: a pointer to the first element in the queue.
  - `rear`: a pointer to the last element in the queue.
- Initialize both `front` and `rear` to `NULL` to represent an empty queue.

2. **Function: createNode(data)**

- Allocate memory for a new node.
- Set the `data` field of the new node to the given value.
- Set the `next` field of the new node to `NULL`.
- Return the created node.

3. **Function: initQueue(queue)**

- Set `front` and `rear` of the queue to `NULL`, representing an empty queue.

4. **Function: isEmpty(queue)**

- If `front == NULL`, return `1` (indicating the queue is empty).
- Otherwise, return `0` (indicating the queue is not empty).

5. **Function: enqueue(queue, data)**

- Create a new node using `createNode(data)`.
- If the queue is empty (`front == NULL`):
  - Set both `front` and `rear` to point to the new node.
  - Set the `next` pointer of the new node to point to `front`, maintaining the circular link.
- If the queue is not empty:
  - Set the `next` pointer of `rear` to the new node.

    - Update `rear` to point to the new node.
    - Set the `next` pointer of `rear` to `front` to maintain the circular link.
- Print "Enqueued: data".

6. **Function: dequeue(queue)**

- Check if the queue is empty by calling `isEmpty(queue)`.
    - If the queue is empty, print "Queue Underflow: Queue is empty".
- If the queue is not empty:
    - Store the current `front` in a temporary variable `temp`.
    - If `front == rear` (only one element in the queue), set both `front` and `rear` to `NULL`.
    - Otherwise, update `front` to point to the next node, and adjust the `next` pointer of `rear` to point to the new `front`.
    - Print and return the data of the node in `temp`.
    - Free the memory of `temp`.

7. **Function: peek(queue)**

- Check if the queue is empty by calling `isEmpty(queue)`.
    - If the queue is empty, print "Queue is empty".
- If the queue is not empty, print the data of the `front` node.

8. **Function: size(queue)**

- If the queue is empty, return `0`.
- If the queue is not empty:
    - Initialize a count variable to 0.
    - Traverse the queue starting from `front`, and increment the count for each node.
    - Stop when the traversal reaches `front` again (due to the circular link).
    - Return the count as the size of the queue.

9. **Function: display(queue)**

- Check if the queue is empty by calling `isEmpty(queue)`.
    - If the queue is empty, print "Queue is empty".
- If the queue is not empty:
    - Traverse the queue starting from `front`.
    - Print the data of each node until the traversal reaches `front` again (due to the circular link).

10. **Main Program Loop:**

- Display a menu with the following options:

    - Enqueue
    - Dequeue
    - Peek
    - Display
    - Size
    - Exit
- Based on the user's input, perform the corresponding operation:
    - **Enqueue:** Prompt the user to input a value and enqueue it to the queue. Continue enqueuing values until the user enters `-1`.
    - **Dequeue:** Remove and display the front element of the queue.
    - **Peek:** Display the front element of the queue.
    - **Display:** Show all elements in the queue.
    - **Size:** Show the size of the queue.
    - **Exit:** Exit the program.

Code

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

// Define a node structure for the linked list
struct Node {
    int data;
    struct Node* next;
};

// Define a circular queue structure
struct CircularQueue {
    struct Node* front;
    struct Node* rear;
};

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

// Function to initialize the circular queue
void initQueue(struct CircularQueue* queue) {
    queue->front = queue->rear = NULL;
}
```

```c
// Function to check if the queue is empty
int isEmpty(struct CircularQueue* queue) {
    return queue->front == NULL;
}

// Function to enqueue an element into the circular queue
void enqueue(struct CircularQueue* queue, int data) {
    struct Node* newNode = createNode(data);
    if (isEmpty(queue)) {
        queue->front = queue->rear = newNode;
        newNode->next = queue->front; // Circular link
    } else {
        queue->rear->next = newNode;
        queue->rear = newNode;
        queue->rear->next = queue->front; // Circular link
    }
    printf("Enqueued: %d\n", data);
}

// Function to dequeue an element from the circular queue
void dequeue(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue Underflow: Queue is empty\n");
        return;
    }

    struct Node* temp = queue->front;
    if (queue->front == queue->rear) {
        queue->front = queue->rear = NULL;
    } else {
        queue->front = queue->front->next;
        queue->rear->next = queue->front;
    }

    printf("Dequeued: %d\n", temp->data);
    free(temp);
}

// Function to peek the front element of the queue
void peek(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
    } else {
        printf("Front element: %d\n", queue->front->data);
    }
}

// Function to check if the queue is empty
int size(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        return 0;
    }

    int count = 0;
    struct Node* temp = queue->front;
    do {
        count++;
        temp = temp->next;
    } while (temp != queue->front);
    return count;
}

// Function to display the elements of the circular queue
void display(struct CircularQueue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return;
    }

    struct Node* temp = queue->front;
    printf("Queue elements: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != queue->front);
    printf("\n");
}

// Main function to drive the circular queue operations
int main() {
    struct CircularQueue queue;
    initQueue(&queue);

    int choice, data;

    while (1) {
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Peek\n");
        printf("4. Display\n");
        printf("5. Size\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue (-1 to stop): ");
                while (1) {
                    scanf("%d", &data);
                    if (data == -1) {
                        break;
                    }
                    enqueue(&queue, data);
                }
                break;

            case 2:
                dequeue(&queue);
                break;

            case 3:
                peek(&queue);
                break;

            case 4:
                display(&queue);
                break;

            case 5:
                printf("Queue size: %d\n", size(&queue));
                break;

            case 6:
                printf("Exiting...\n");
                exit(0);

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
Developer/dsa/q9 via 🅒 v14.2.1-gcc
❯ ./llcircularqueue

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
```

```
Enter your choice: 1
Enter value to enqueue (-1 to stop): 34
Enqueued: 34
47
Enqueued: 47
89
Enqueued: 89
65
Enqueued: 65
-1

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 4
Queue elements: 34 47 89 65

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 2
Dequeued: 34

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 3
Front element: 47

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 4
Queue elements: 47 89 65
```

```
Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 5
Queue size: 3

Circular Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Size
6. Exit
Enter your choice: 6
Exiting...
```

Q.10) Implementation of Tree Structure, Binary Tree, Tree Traversal, Binary Search Tree, Insertion and Deletion in BST

Binary Tree And Traversal

Algorithm

1. **Node Structure:**
   - Define a structure with fields `data`, `left`, and `right`.
2. **Create Node:**
   - Allocate memory, assign `data`, and set `left` and `right` to `NULL`.
3. **Insert Node:**
   - Prompt user for input.
   - If `-1`, return `NULL`.
   - Create a node, recursively set `left` and `right` using user input.
4. **Inorder Traversal (Left, Root, Right):**
   - If node is `NULL`, return.
   - Traverse left, print root, traverse right.
5. **Preorder Traversal (Root, Left, Right):**
   - If node is `NULL`, return.
   - Print root, traverse left, traverse right.
6. **Postorder Traversal (Left, Right, Root):**
   - If node is `NULL`, return.

- Traverse left, traverse right, print root.

7. **Output:**
   - Call all three traversal functions on the root node.

Code

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

// Define the structure of a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

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

// Function to insert nodes in a binary tree (user-driven)
struct Node* insertNode() {
    int data;
    printf("Enter data (-1 for no node): ");
    scanf("%d", &data);

    if (data == -1) {
        return NULL;
    }

    struct Node* newNode = createNode(data);

    printf("Enter left child of %d:\n", data);
    newNode->left = insertNode();

    printf("Enter right child of %d:\n", data);
    newNode->right = insertNode();

    return newNode;
}

// **Binary Tree Traversals**
// Inorder Traversal (Left, Root, Right)
```

```c
void inorderTraversal(struct Node* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

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

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

// Main function to drive the binary tree operations
int main() {
    struct Node* root = NULL;

    printf("Create the binary tree:\n");
    root = insertNode();

    printf("\nTree Traversals:\n");

    printf("Inorder Traversal: ");
    inorderTraversal(root);
    printf("\n");

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

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

    return 0;
}
```

Output

```
dsa/q10 on  main [?] via  v14.2.1-gcc
❯ ./binarytree
Create the binary tree:
Enter data (-1 for no node): 67
Enter left child of 67:
Enter data (-1 for no node): 90
Enter left child of 90:
Enter data (-1 for no node): 34
Enter left child of 34:
Enter data (-1 for no node): -1
Enter right child of 34:
Enter data (-1 for no node): -1
Enter right child of 90:
Enter data (-1 for no node): 32
Enter left child of 32:
Enter data (-1 for no node): -1
Enter right child of 32:
Enter data (-1 for no node): -1
Enter right child of 67:
Enter data (-1 for no node): 45
Enter left child of 45:
Enter data (-1 for no node): -1
Enter right child of 45:
Enter data (-1 for no node): -1

Tree Traversals:
Inorder Traversal: 34 90 32 67 45
Preorder Traversal: 67 90 34 32 45
Postorder Traversal: 34 32 90 45 67
```

Binary Search Tree

Algorithm

Algorithm for BST with BFS and DFS Traversals

1. **Node Creation:**
   - Define a `Node` structure with `data`, `left`, and `right` fields.
2. **Insertion in BST:**
   - If the tree is empty, create the root node with the input data.
   - Otherwise:
     - If `data < root->data`, recursively insert into the left subtree.
     - If `data > root->data`, recursively insert into the right subtree.
3. **Deletion in BST:**

- Locate the node to delete:
  - If the node has no children, remove it directly.
  - If the node has one child, replace it with its child.
  - If the node has two children:
    - Find the minimum value node in the right subtree.
    - Replace the node's data with the minimum value.
    - Recursively delete the minimum value node.

4. **Depth-First Search (DFS) Traversals:**
   - **Preorder (Root-Left-Right):**
     - Print the node's data.
     - Recursively visit the left and right subtrees.
   - **Inorder (Left-Root-Right):**
     - Recursively visit the left subtree.
     - Print the node's data.
     - Recursively visit the right subtree.
   - **Postorder (Left-Right-Root):**
     - Recursively visit the left and right subtrees.
     - Print the node's data.
5. **Breadth-First Search (BFS) Traversal:**
   - Use a queue to store nodes at each level.
   - Print and dequeue nodes level by level.
   - Enqueue the left and right children of each node.
6. **Continuous Insertion:**
   - Accept input values for insertion until the user enters `-1`.
7. **Menu-Driven Execution:**
   - Display options for insertion, deletion, and all traversal techniques.
   - Perform the corresponding operation based on the user’s choice.
   - Repeat until the user chooses to exit.

Code

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

// Define structure for a tree node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Create a new node
```

```c
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Insert a node into the BST
struct Node* insert(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }
    if (data < root->data) {
        root->left = insert(root->left, data);
    } else if (data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}

// Find the minimum value node in the BST
struct Node* findMin(struct Node* root) {
    while (root && root->left != NULL) {
        root = root->left;
    }
    return root;
}

// Delete a node from the BST
struct Node* delete(struct Node* root, int data) {
    if (root == NULL) {
        return root;
    }
    if (data < root->data) {
        root->left = delete(root->left, data);
    } else if (data > root->data) {
        root->right = delete(root->right, data);
    } else {
        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;
        }

        struct Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = delete(root->right, temp->data);
    }
    return root;
}

// Depth-First Search Traversal (Preorder, Inorder, Postorder)
void preorder(struct Node* root) {
    if (root) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

void inorder(struct Node* root) {
    if (root) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void postorder(struct Node* root) {
    if (root) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}

// Breadth-First Search Traversal (Level Order)
void bfs(struct Node* root) {
    if (root == NULL) {
        return;
    }

    struct Node* queue[100];
    int front = 0, rear = 0;

    queue[rear++] = root;

    while (front < rear) {
        struct Node* current = queue[front++];
        printf("%d ", current->data);

        if (current->left != NULL) {
            queue[rear++] = current->left;
        }
        if (current->right != NULL) {
            queue[rear++] = current->right;
        }
    }
}

// Main function
int main() {
    struct Node* root = NULL;
    int choice, value;

    while (1) {
        printf("\nBST Operations:\n");
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Preorder Traversal (DFS)\n");
        printf("4. Inorder Traversal (DFS)\n");
        printf("5. Postorder Traversal (DFS)\n");
        printf("6. Level Order Traversal (BFS)\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter values to insert (-1 to stop): ");
                while (1) {
                    scanf("%d", &value);
                    if (value == -1) {
                        break;
                    }
                    root = insert(root, value);
                }
                break;

            case 2:
                printf("Enter value to delete: ");
                scanf("%d", &value);
                root = delete(root, value);
                break;

            case 3:
                printf("Preorder Traversal: ");
                preorder(root);
                printf("\n");
                break;

            case 4:
                printf("Inorder Traversal: ");
                inorder(root);
                printf("\n");
                break;
            case 5:
                printf("Postorder Traversal: ");
                postorder(root);
                printf("\n");
                break;

            case 6:
                printf("Level Order Traversal: ");
                bfs(root);
                printf("\n");
                break;

            case 7:
                printf("Exiting...\n");
                exit(0);

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Output

```
dsa/q10 on  main [?] via  v14.2.1-gcc
❯ ./bst

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 1
Enter values to insert (-1 to stop): 23 85 61 90 84 20
-1

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
```

```
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 3
Preorder Traversal: 23 20 85 61 84 90

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 4
Inorder Traversal: 20 23 61 84 85 90

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 5
Postorder Traversal: 20 84 61 90 85 23

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 6
Level Order Traversal: 23 20 85 61 90 84

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 2
Enter value to delete: 61

BST Operations:
```

```
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 3
Preorder Traversal: 23 20 85 84 90

BST Operations:
1. Insert
2. Delete
3. Preorder Traversal (DFS)
4. Inorder Traversal (DFS)
5. Postorder Traversal (DFS)
6. Level Order Traversal (BFS)
7. Exit
Enter your choice: 7
Exiting...
```