Implement a circular queue data structure to efficiently store and manage elements in a fixed-size array, allowing for reuse of space as elements are dequeued.

Let's break down what's needed to implement a circular queue in C:

Aim

- Implement a circular queue data structure to efficiently store and manage elements in a fixed-size array, allowing for reuse of space as elements are dequeued.

Procedure

1. **Include Headers**: Include the standard input/output library `stdio.h`.
2. **Define Constants**:
    - `MAX_SIZE`: Set the maximum number of elements the queue can hold.
3. **Declare Variables**:
    - `queue`: An array to store the queue elements.
    - `front`: Index of the front element (where deletion occurs). Initialize to 0.
    - `rear`: Index where the next element will be inserted. Initialize to -1 (indicating an empty queue).
    - `itemCount`: Keeps track of the number of elements currently in the queue. Initialize to 0.
4. **Implement `insert(data)` function**:
    - Check if the queue is full (`itemCount == MAX_SIZE`). If so, print an overflow message.
    - If not full:
        - Calculate the new `rear` index using the modulo operator (`%`) to handle circular wrapping: `rear = (rear + 1) % MAX_SIZE`.
        - Insert the `data` at the `rear` index: `queue[rear] = data`.
        - Increment `itemCount`.
5. **Implement `removeData()` function**:
    - Check if the queue is empty (`itemCount == 0`). If so, print an underflow message.
    - If not empty:
        - Store the data at the `front` index in a temporary variable.
        - Calculate the new `front` index using modulo: `front = (front + 1) % MAX_SIZE`.
        - Decrement `itemCount`.

        - Return the removed data.

6. **Implement `display()` function**:
    - Check if the queue is empty. If so, print a message.
    - If not empty:
        - Print the `front` element.
        - Iterate through the queue using circular indexing:
            - Calculate the current index: `index = (front + i) % MAX_SIZE`.
            - Print the element at `queue[index]`.
        - Print the `rear` element.
7. `main` **function**:
    - Demonstrate the circular queue operations by calling `insert` and `removeData`.
    - Use `display` to visualize the queue contents at different stages.

Code

C

```c
#include <stdio.h>

#define MAX_SIZE 5

int queue[MAX_SIZE];
int front = 0, rear = -1, itemCount = 0;

// Function to insert an element into the circular queue
void insert(int data) {
   if(itemCount == MAX_SIZE) {
      printf("Queue Overflow\n");
   } else {
      rear = (rear + 1) % MAX_SIZE;
      queue[rear] = data;
      itemCount++;
      printf("Inserted: %d\n", data);
   }
}

// Function to remove an element from the circular queue
int removeData() {
   int data;

   if(itemCount == 0) {
      printf("Queue Underflow\n");
      return -1;
   } else {
      data = queue[front];
      front = (front + 1) % MAX_SIZE;
      itemCount--;
      return data;
   }
}

// Function to display the elements of the circular queue
void display() {
   if(itemCount == 0) {
      printf("Queue is empty\n");
   } else {
      printf("Front -> %d\n", queue[front]);
      printf("Items -> ");
      for(int i = 0; i < itemCount; i++) {
         int index = (front + i) % MAX_SIZE;
         printf("%d ", queue[index]);
      }
      printf("\nRear -> %d\n", queue[rear]);
   }
}

int main() {
   // Test the circular queue
   insert(3);
   insert(5);
   insert(9);
   insert(1);
   insert(12);

   display();
   int removedItem = removeData();

   if(removedItem != -1) {
      printf("Removed: %d\n", removedItem);
   }

   display();

   return 0;
}
```

Let me know if you have any other questions or would like to see how to implement other data structures or algorithms!