

Module 3 Unit II

Queue

What is a Queue?

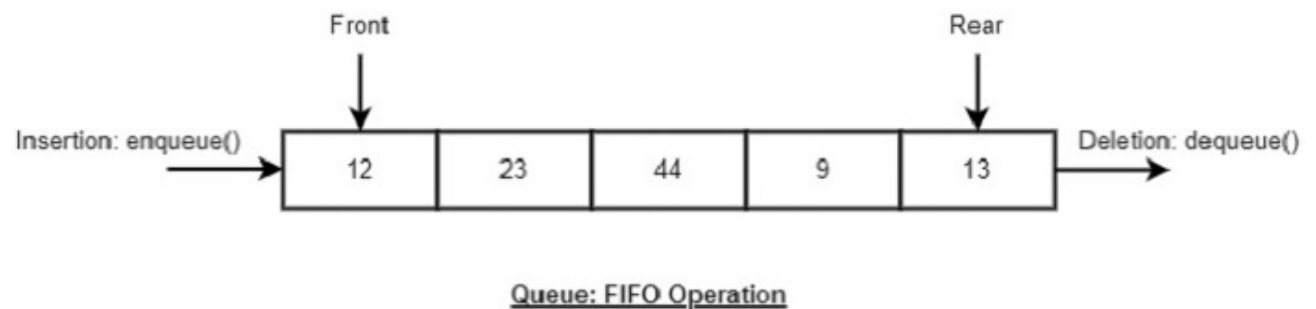
- A **queue** is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed.
- A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends.
- The data is inserted into the queue through one end and deleted from it using the other end. Queue is very frequently used in most programming languages.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus stops.

Representation of Queues

Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. As a small example in this tutorial, we implement queues using a one-dimensional array.



Basic Operations in Queue

Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.

The most fundamental operations in the queue ADT include: `enqueue()`, `dequeue()`, `peek()`, `isFull()`, `isEmpty()`. These are all built-in operations to carry out data manipulation and to check the status of the queue.

Queue uses two pointers – **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Queue Insertion Operation: Enqueue()

The `enqueue()` is a data manipulation operation that is used to insert elements into the queue. The following algorithm describes the `enqueue()` operation in a simpler way.

Algorithm

1. START
2. Check if the queue is full.
3. If the queue is full, produce overflow error and exit.
4. If the queue is not full, increment rear pointer to point the next empty space.
5. Add data element to the queue location, where the rear is pointing.
6. return success.

7. END

Example:

```
#include <stdio.h>

#include <stdbool.h>

#include <string.h>

#include <stdlib.h>

#define MAX 6


int queue[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;


// Function to check if the queue is full
bool isFull() {
    return itemCount == MAX;
}


// Function to check if the queue is empty
bool isEmpty() {
    return itemCount == 0;
}


// Function to insert an element into the queue
void insert(int data) {
    if (!isFull()) {
        if (rear == MAX - 1) {
            rear = -1;
        }
        queue[++rear] = data;
```

```
        itemCount++;  
    } else {  
        printf("Queue is full! Cannot insert %d\n", data);  
    }  
}
```

// Function to remove an element from the queue

```
int removeData() {  
    int data = queue[front++];  
    if (front == MAX) {  
        front = 0;  
    }  
    itemCount--;  
    return data;  
}
```

// Main function

```
int main() {  
    // Inserting elements into the queue  
    insert(3);  
    insert(5);  
    insert(9);  
    insert(1);  
    insert(12);  
    insert(15);
```

// Displaying queue elements

```
printf("Queue: ");  
while (!isEmpty()) {  
    int data = removeData();
```

```

        printf("%d ", data);
    }
    printf("\n");

    return 0;
}

```

Output

Queue: 3 5 9 1 12 15

Queue Deletion Operation: dequeue()

The dequeue() is a data manipulation operation that is used to remove elements from the queue. The following algorithm describes the dequeue() operation in a simpler way.

Algorithm

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available data element.
6. Return success.
7. END

Example:

```

#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int queue[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;

```

```
// Function to check if queue is full
```

```
bool isFull() {  
    return itemCount == MAX;  
}
```

```
// Function to check if queue is empty
```

```
bool isEmpty() {  
    return itemCount == 0;  
}
```

```
// Function to insert data into queue
```

```
void insert(int data) {  
    if (!isFull()) {  
        if (rear == MAX - 1) {  
            rear = -1;  
        }  
        queue[++rear] = data;  
        itemCount++;  
    } else {  
        printf("Queue is full. Cannot insert %d\n", data);  
    }  
}
```

```
// Function to remove data from queue
```

```
int removeData() {  
    if (!isEmpty()) {  
        int data = queue[front++];  
        if (front == MAX) {  
            front = 0;  
        }  
    }  
}
```

```

    }
    itemCount--;
    return data;
} else {
    printf("Queue is empty. Cannot remove data.\n");
    return -1;
}
}

```

// Function to display the queue content

```

void displayQueue() {
    printf("Queue: ");
    int count = itemCount;
    int i = front;
    while (count > 0) {
        printf("%d ", queue[i]);
        i = (i + 1) % MAX;
        count--;
    }
    printf("\n");
}

```

```

int main() {
    // Insert 6 items
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);
    insert(15);
}

```

```
displayQueue(); // Show queue

// Remove one item
int removed = removeData();
printf("Element removed: %d\n", removed);

// Display queue after removal
displayQueue();

// Remove remaining elements
printf("Removing remaining elements: ");
while (!isEmpty()) {
    printf("%d ", removeData());
}
printf("\n");

return 0;
}
```

Output

Queue: 3 5 9 1 12 15
Element removed: 3
Updated Queue: 5 9 1 12 15

Queue - The peek() Operation

The peek() is an operation which is used to retrieve the frontmost element in the queue, without deleting it. This operation is used to check the status of the queue with the help of the pointer.

Algorithm:

1. START
2. Return the element at the front of the queue
3. END

Example:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#define MAX 6

int intArray[MAX];

int front = 0;

int rear = -1;

int itemCount = 0;

bool isFull() {
    return itemCount == MAX;
}

bool isEmpty() {
    return itemCount == 0;
}

void insert(int data) {
    if (!isFull()) {
        if (rear == MAX - 1) {
            rear = -1;
        }
        intArray[++rear] = data;
```

```
        itemCount++;  
    } else {  
        printf("Queue is full! Cannot insert %d\n", data);  
    }  
}
```

```
int peek() {  
    if (!isEmpty())  
        return intArray[front];  
    else {  
        printf("Queue is empty!\n");  
        return -1;  
    }  
}
```

```
void displayQueue() {  
    printf("Queue: ");  
    if (isEmpty()) {  
        printf("Empty\n");  
        return;  
    }  
    int count = itemCount;  
    int i = front;  
    while (count > 0) {  
        printf("%d ", intArray[i]);  
        i = (i + 1) % MAX;  
        count--;  
    }  
    printf("\n");  
}
```

```
int main() {  
    insert(3);  
    insert(5);  
    insert(9);  
    insert(1);  
    insert(12);  
    insert(15); // Should fill the queue  
  
    displayQueue();  
    printf("Element at front: %d\n", peek());  
    return 0;  
}
```

Output:

Queue: 3 5 9 1 12 15

Element at front: 3

Queue - The isFull() Operation

The isFull() operation verifies whether the stack is full.

Algorithm

1. START
2. If the count of queue elements equals the queue size,
return true
3. Otherwise, return false
4. END

Example:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX 6
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;
bool isFull(){
    return itemCount == MAX;
}
void insert(int data){
    if(!isFull()) {
        if(rear == MAX-1) {
            rear = -1;
        }
        intArray[++rear] = data;
        itemCount++;
    }
}
```

```
int main(){
    int i;
    /* insert 5 items */
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);
    insert(15);
    printf("Queue: ");
    for(i = 0; i < MAX; i++)
        printf("%d ", intArray[i]);
```

```
    if(isFull()) {
        printf("Queue is full!\n");
    }
}
```

Output

Queue: 3 5 9 1 12 15

Queue is full!

Queue - The isEmpty() operation

The isEmpty() operation verifies whether the stack is empty. This operation is used to check the status of the stack with the help of top pointer.

Algorithm

1. START
2. If the count of queue elements equals zero, return true
3. Otherwise, return false
4. END

Queue Complete Implementation:

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

#define MAX 6

int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

int peek(){
    return intArray[front];
}

bool isEmpty(){
    return itemCount == 0;
}

bool isFull(){
    return itemCount == MAX;
}

int size(){
    return itemCount;
}

void insert(int data){
    if(!isFull()) {
```

```

        if(rear == MAX-1) {
            rear = -1;
        }
        intArray[++rear] = data;
        itemCount++;
    }
}

int removeData(){
    int data = intArray[front++];
    if(front == MAX) {
        front = 0;
    }
    itemCount--;
    return data;
}

int main(){

    /* insert 5 items */
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);
    insert(15);
    printf("Queue size: %d", size());
    printf("\nQueue: ");
    for(int i = 0; i < MAX; i++){
        printf("%d ", intArray[i]);
    }
    if(isFull()) {

```

```

        printf("\nQueue is full!");
    }

    // remove one item
    int num = removeData();
    printf("\nElement removed: %d", num);
    printf("\nSize of Queue after deletion: %d", size());
    printf("\nElement at front: %d", peek());
}

```

Output

```

Queue size: 6
Queue: 3 5 9 1 12 15
Queue is full!
Element removed: 3
Size of Queue after deletion: 5
Element at front: 5

```

Types of queues –

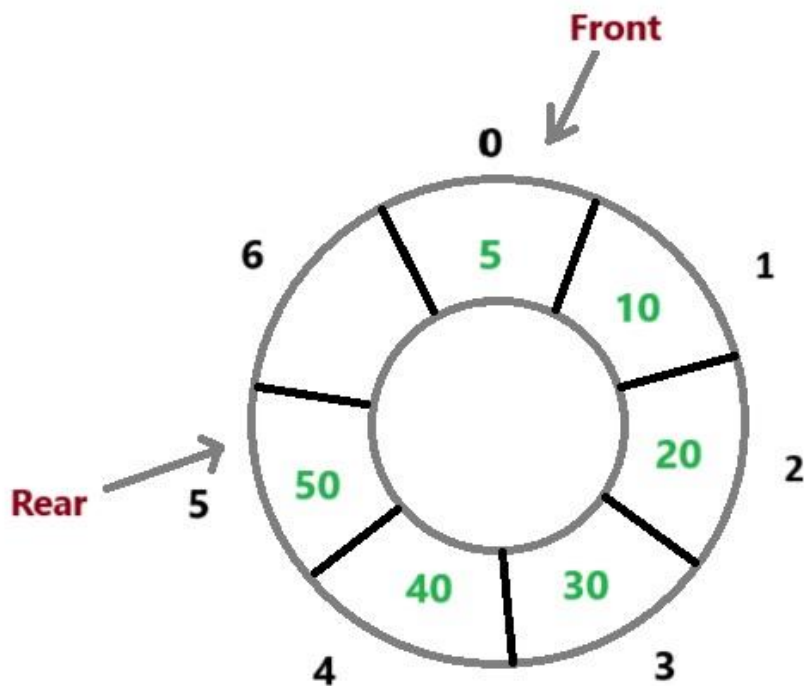
1. Simple queues
2. Circular queues
3. Double-Ended queues
4. Priority queues

Circular queues:

- A circular queue is a type of queue in which the last position is connected back to the first position to make a circle. It is also known as a **Ring Buffer**.
- In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using a circular queue, we can use the space to insert elements.
- It is a linear data structure that follows the FIFO mechanism. The circular queue is a more efficient way to implement a queue in a fixed size array. In a circular queue, the last element points to the first element making a circular link.

Representation of Circular Queue

Following is the representation of a circular queue, where the front is the index of the first element, and the rear is the index of the last element.



Operations on Circular Queue

There are mainly four operations that can be performed on a circular queue:

- **Enqueue:** Insert an element into the circular queue.
- **Dequeue:** Delete an element from the circular queue.
- **Front:** Get the front element from the circular queue.
- **Rear:** Get the last element from the circular queue.

Enqueue Operation on Circular Queue

Enqueue operation is used for inserting an element into the circular queue. The steps to perform the enqueue operation are as follows:

Algorithm for Enqueue Operation

Following are the steps to perform the enqueue operation on a circular queue:

1. Initialize an array or any data structure to store the elements of the circular queue.
2. Initialize two variables front and rear.
3. Check if the circular queue is full.
4. If it is not full, and if the front is -1, set the front to 0.
5. Increment the rear by 1 and store it in the rear index.
6. Update the rear index using $\text{rear} = (\text{rear} + 1) \% \text{SIZE}$.

Dequeue Operation on Circular Queue

Dequeue operation is used for deleting an element from the circular queue. The steps to perform the dequeue operation are as follows:

Algorithm for Dequeue Operation

Following are the steps to perform the dequeue operation on a circular queue:

1. Check if the circular queue is empty.
2. If the queue is not empty, store the element at the front index.
3. If the front and rear are equal, set the front and rear to -1.
4. Else, increase the front index by 1 using the formula $\text{front} = (\text{front} + 1) \% \text{SIZE}$.
5. At last print the deleted element.

Front Operation on Circular Queue

Front operation is used to get the front element from the circular queue. The steps to perform the front operation are as follows:

Algorithm for Front Operation

1. Check if the circular queue is empty.
2. If not empty, print the front index element/

Rear Operation on Circular Queue

Rear operation is used to get the last element from the circular queue. The steps to perform the rear operation are as follows:

Algorithm for Rear Operation

1. Check if the circular queue is empty..
2. If it is not empty, print the element at the rear index.

Implementation of Circular queue:

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

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

```
#define MAX_SIZE 5 // Define the maximum size of the circular queue
```

```
int cqueue_arr[MAX_SIZE];
```

```
int front = -1;
```

```
int rear = -1;
```

```
// Function to check if the queue is empty
```

```
int isEmpty() {
```

```
    return (front == -1);
```

```
}
```

```
// Function to check if the queue is full
```

```
int isFull() {
```

```
    return ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1));
```

```
}
```

```
// Function to insert an element into the circular queue
```

```
void enqueue(int item) {
```

```
    if (isFull()) {
```

```
        printf("Queue Overflow\n");
```

```
        return;
```

```
    }
```

```
    if (front == -1) { // If the queue is initially empty
```

```
        front = 0;
```

```
        rear = 0;
```

```
    } else if (rear == MAX_SIZE - 1) { // Wrap around if rear reaches the end
```

```
        rear = 0;
```

```
    } else {
```

```
        rear++;
```

```
    }
```

```
    cqueue_arr[rear] = item;
```

```
printf("Enqueued: %d\n", item);

}

// Function to delete an element from the circular queue

void dequeue() {

    if (isEmpty()) {

        printf("Queue Underflow\n");

        return;

    }

    printf("Dequeued: %d\n", cqueue_arr[front]);

    if (front == rear) { // If only one element was present

        front = -1;

        rear = -1;

    } else if (front == MAX_SIZE - 1) { // Wrap around if front reaches the end

        front = 0;

    } else {

        front++;

    }

}
```

```
// Function to display the elements of the circular queue
```

```
void display() {
```

```
    if (isEmpty()) {
```

```
        printf("Queue is empty\n");
```

```
        return;
```

```
    }
```

```
    printf("Queue elements: ");
```

```
    int i = front;
```

```
    if (front <= rear) {
```

```
        while (i <= rear) {
```

```
            printf("%d ", cqueue_arr[i]);
```

```
            i++;
```

```
        }
```

```
    } else { // When the queue wraps around
```

```
        while (i < MAX_SIZE) {
```

```
            printf("%d ", cqueue_arr[i]);
```

```
            i++;
```

```
        }
```

```
        i = 0;
```

```
        while (i <= rear) {

            printf("%d ", cqueue_arr[i]);

            i++;

        }

    }

    printf("\n");

}

int main() {

    enqueue(10);

    enqueue(20);

    enqueue(30);

    display();

    dequeue();

    display();

    enqueue(40);

    enqueue(50);

    enqueue(60); // This will cause overflow as MAX_SIZE is 5

    display();

    dequeue();
```

```
    dequeue();

    display();

    enqueue(70);

    display();

return 0;

}
```

Output:

Code

```
Enqueued: 10
Enqueued: 20
Enqueued: 30
Queue elements: 10 20 30
Dequeued: 10
Queue elements: 20 30
Enqueued: 40
Enqueued: 50
Queue Overflow
Queue elements: 20 30 40 50
Dequeued: 20
Dequeued: 30
Queue elements: 40 50
Enqueued: 70
Queue elements: 40 50 70
```


Applications of Circular Queue

Following are some of the applications of a circular queue:

- **Memory Management:** Circular queues are used in memory management to manage the memory efficiently. It is used to allocate memory to the processes when they are in need of memory.
- **Buffer Management:** These queues are also useful in buffer management. Consider a situation where data is produced at a faster rate than it is consumed. In such cases, a circular queue is used to store the data temporarily.
- **Operating System:** Suppose your system has a lot of processes to execute. In such cases, for the better management of processes, the operating system uses a circular queue to allocate the CPU to the processes.
- **Traffic Management:** Traffic signals are controlled by the circular queue. Let's say there are three signals, red, yellow, and green. The signals are in a circular queue. When the green signal is on, the next signal will be yellow, and then red. This process will continue in a circular manner.
- **Multimedia Player:** When we play songs in a multimedia player, the songs are played in a circular manner. Once the last song is played, the first song will be played next. This is done using a circular queue.

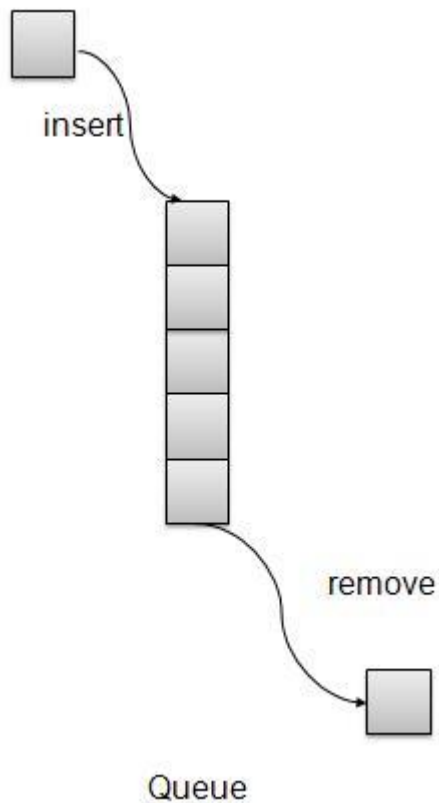
Priority Queue :

Priority Queue is more specialized data structure than Queue. Like ordinary queue, priority queue has same method but with a major difference. In Priority queue items are **ordered by key value** so that item with the lowest value of key is at front and item with the highest value of key is at rear or vice versa. So we're assigned priority to item based on its key value. Lower the value, higher the priority. Following are the principal methods of a Priority Queue.

Basic Operations

- **insert / enqueue** – add an item to the rear of the queue.
- **remove / dequeue** – remove an item from the front of the queue.

Priority Queue Representation



There are few more operations supported by queue which are following.

- **Peek** – get the element at front of the queue.
- **isFull** – check if queue is full.
- **isEmpty** – check if queue is empty.

Example Program:

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

#define MAX 6

int intArray[MAX];
int itemCount = 0;
```

```
int peek(){
    return intArray[itemCount - 1];
}
```

```
bool isEmpty(){
    return itemCount == 0;
}
```

```
bool isFull(){
    return itemCount == MAX;
}
```

```
int size(){
    return itemCount;
}
```

```
void insert(int data){
    int i = 0;
```

```
    if(!isFull()){
        // if queue is empty, insert the data
        if(itemCount == 0){
            intArray[itemCount++] = data;
        }else{
            // start from the right end of the queue
```

```
            for(i = itemCount - 1; i >= 0; i-- ){
                // if data is larger, shift existing item to right end
                if(data > intArray[i]){
                    intArray[i+1] = intArray[i];
```

```

        }else{
            break;
        }
    }

    // insert the data
    intArray[i+1] = data;
    itemCount++;
}
}

int removeData(){
    return intArray[--itemCount];
}

int main() {
    /* insert 5 items */
    insert(3);
    insert(5);
    insert(9);
    insert(1);
    insert(12);

    // -----
    // index : 0 1 2 3 4
    // -----
    // queue : 12 9 5 3 1
    insert(15);

```

```
// -----  
// index : 0 1 2 3 4 5  
// -----  
// queue : 15 12 9 5 3 1
```

```
if(isFull()){  
    printf("Queue is full!\n");  
}
```

```
// remove one item  
int num = removeData();  
printf("Element removed: %d\n",num);
```

```
// -----  
// index : 0 1 2 3 4  
// -----  
// queue : 15 12 9 5 3
```

```
// insert more items  
insert(16);
```

```
// -----  
// index : 0 1 2 3 4 5  
// -----  
// queue : 16 15 12 9 5 3
```

```
// As queue is full, elements will not be inserted.  
insert(17);  
insert(18);
```

```

// -----
// index : 0  1  2  3  4  5
// -----
// queue : 16 15 12 9 5 3
printf("Element at front: %d\n",peek());

printf("-----\n");
printf("index : 5 4 3 2  1  0\n");
printf("-----\n");
printf("Queue: ");

while(!isEmpty()){
    int n = removeData();
    printf("%d ",n);
}
}

```

Output:

```

Queue is full!
Element removed: 1
Element at front: 3
-----
index : 5 4 3 2  1  0
-----
Queue:  3 5 9 12 15 16

```

Double-ended queues:

Deque is a hybrid data structure that combines the features of a **stack** and a **queue**. It allows us to insert and delete elements from both ends of the queue. The name **Deque** is an abbreviation of **Double-Ended Queue**.

Imagine an event where you have two gates to enter and exit a place. People are entering from the **front gate** and some are entering from the **side gate**. Now, when people are leaving, they are leaving from the **front gate** and some sneak from the **side gate**. Now, we need to manage flow of people from both ends. This is where **Deque** comes into play.

A Double Ended Queue (Deque), pronounced "deck," is a linear data structure that allows for insertion and deletion of elements from both ends, namely the front and the rear. This contrasts with a traditional queue, which only allows insertions at the rear and deletions from the front (First-In, First-Out).

Implementation in C:

A Deque can be implemented in C using either an array or a linked list. The array implementation typically uses two pointers, front and rear, to manage the ends of the deque within a fixed-size array. The linked list implementation uses nodes with data and pointers to the next and previous nodes, allowing for dynamic sizing.

Core Operations:

The primary operations on a Deque include:

- `insertFront(value)`: Adds an element to the front of the deque.
- `insertRear(value)`: Adds an element to the rear of the deque.
- `deleteFront()`: Removes and returns the element from the front of the deque.
- `deleteRear()`: Removes and returns the element from the rear of the deque.
- `isEmpty()`: Checks if the deque is empty.
- `isFull()` (for array implementation): Checks if the deque is full.
- `display()`: Prints the elements of the deque.

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

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

```
#define MAX_SIZE 10
```

```
int deque[MAX_SIZE];
```

```
int front = -1, rear = -1;
```

```
// Function to check if the deque is empty
```

```
int isEmpty() {
```

```
    return (front == -1);
```

```
}
```

```
// Function to check if the deque is full
int isFull() {
    return ((front == 0 && rear == MAX_SIZE - 1) || (front == rear + 1));
}
```

```
// Function to insert an element at the front
void insertFront(int data) {
    if (isFull()) {
        printf("Deque is full. Cannot insert at front.\n");
        return;
    }
    if (isEmpty()) {
        front = 0;
        rear = 0;
    } else if (front == 0) {
        front = MAX_SIZE - 1;
    } else {
        front--;
    }
    deque[front] = data;
    printf("Inserted %d at front.\n", data);
}
```

```
// Function to insert an element at the rear
void insertRear(int data) {
    if (isFull()) {
        printf("Deque is full. Cannot insert at rear.\n");
        return;
    }
    if (isEmpty()) {
```



```

    front = 0;
    rear = 0;
} else if (rear == MAX_SIZE - 1) {
    rear = 0;
} else {
    rear++;
}
deque[rear] = data;
printf("Inserted %d at rear.\n", data);
}

```

// Function to delete an element from the front

```

int deleteFront() {
    if (isEmpty()) {
        printf("Deque is empty. Cannot delete from front.\n");
        return -1; // Indicate error
    }
    int data = deque[front];
    if (front == rear) { // Only one element in deque
        front = -1;
        rear = -1;
    } else if (front == MAX_SIZE - 1) {
        front = 0;
    } else {
        front++;
    }
    printf("Deleted %d from front.\n", data);
    return data;
}

```

```

// Function to delete an element from the rear
int deleteRear() {
    if (isEmpty()) {
        printf("Deque is empty. Cannot delete from rear.\n");
        return -1; // Indicate error
    }
    int data = deque[rear];
    if (front == rear) { // Only one element in deque
        front = -1;
        rear = -1;
    } else if (rear == 0) {
        rear = MAX_SIZE - 1;
    } else {
        rear--;
    }
    printf("Deleted %d from rear.\n", data);
    return data;
}

```

```

// Function to display the elements of the deque
void display() {
    if (isEmpty()) {
        printf("Deque is empty.\n");
        return;
    }
    printf("Deque elements: ");
    int i = front;
    while (1) {
        printf("%d ", deque[i]);
        if (i == rear) break;
    }
}

```

```
        i = (i + 1) % MAX_SIZE;
    }
    printf("\n");
}
```

```
int main() {
    insertRear(10);
    insertFront(5);
    insertRear(20);
    display();

    deleteFront();
    display();

    insertFront(3);
    deleteRear();
    display();

    return 0;
}
```

Output:

Double-Ended Queue Operations:

Inserted 10 at the rear.

Inserted 5 at the front.

Inserted 15 at the rear.

Inserted 2 at the front.

Deque elements: 2 5 10 15

Deleted 2 from the front.

Deque elements: 5 10 15

Deleted 15 from the rear.

Deque elements: 5 10

Inserted 20 at the rear.

Deque elements: 5 10 20

Deleted 5 from the front.

Deleted 10 from the front.

Deleted 20 from the rear.

Deque is empty.

Underflow: Deque is empty. Cannot delete from front.

Note : For Operations on Simple queues refer the lab program and Simple queue above mentioned.