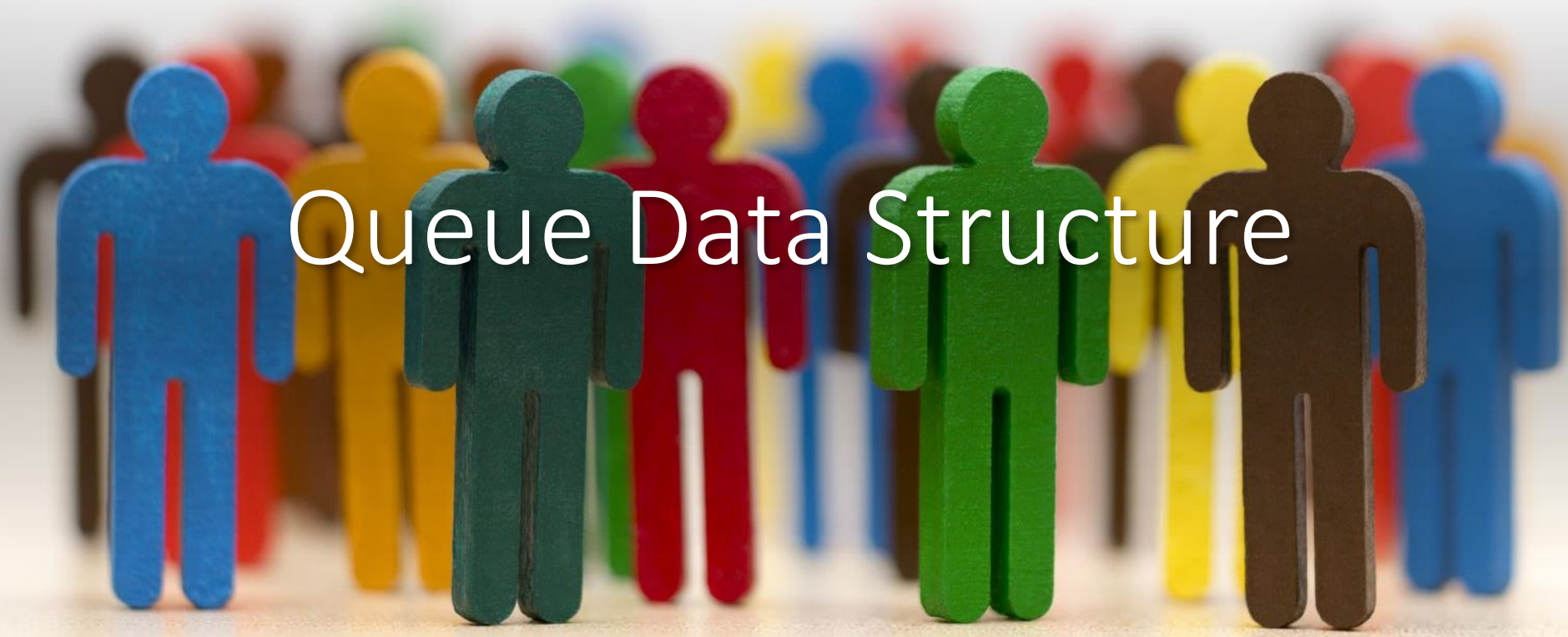


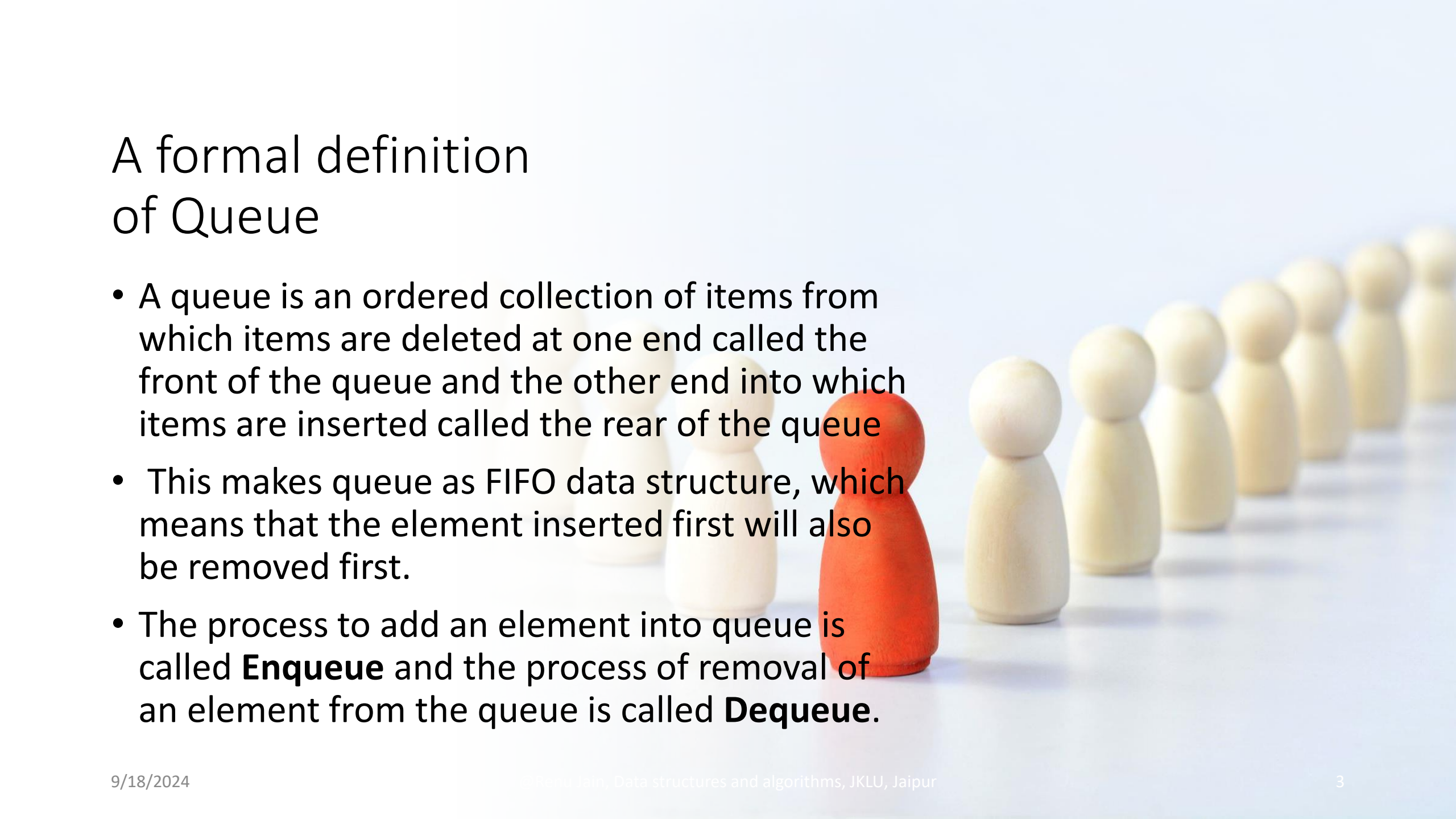
Queue Data Structure

What is a Queue??

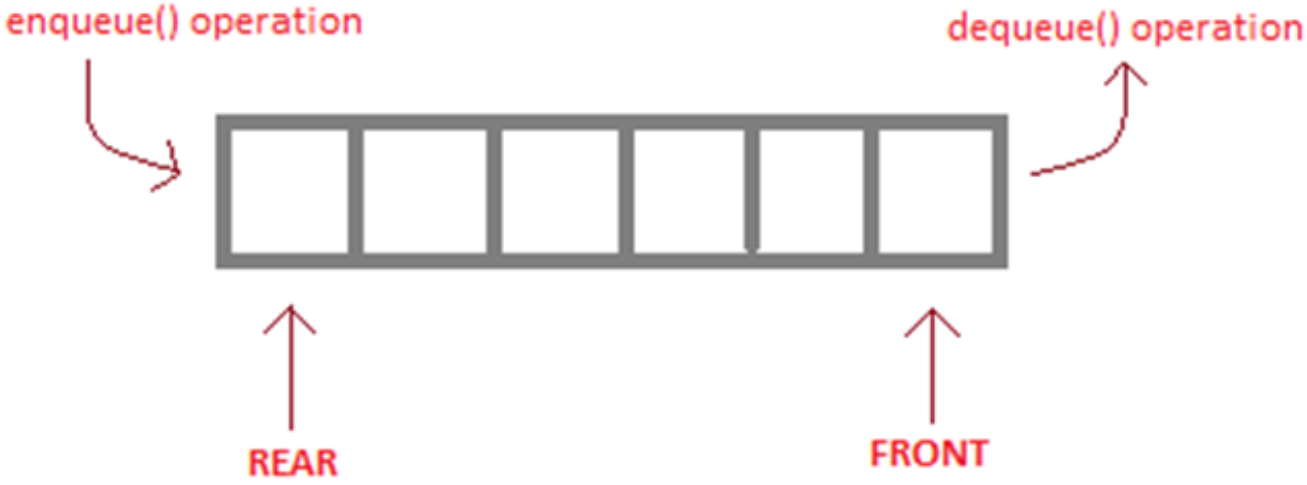
- A queue is an arrangement of persons/users who want to avail a certain service provided by a single agent such that the first come first served principle is followed.
- An arrangement of things/jobs to be done such that first come first served principle is followed.
- Queue has two ends, one called the **front** and other called the **rear** of queue.
- Deletion is done from and addition at rear (Person at front is serviced first and new person is added to the rear of queue)

A formal definition of Queue

- A queue is an ordered collection of items from which items are deleted at one end called the front of the queue and the other end into which items are inserted called the rear of the queue
- This makes queue as FIFO data structure, which means that the element inserted first will also be removed first.
- The process to add an element into queue is called **Enqueue** and the process of removal of an element from the queue is called **Dequeue**.



Queue….



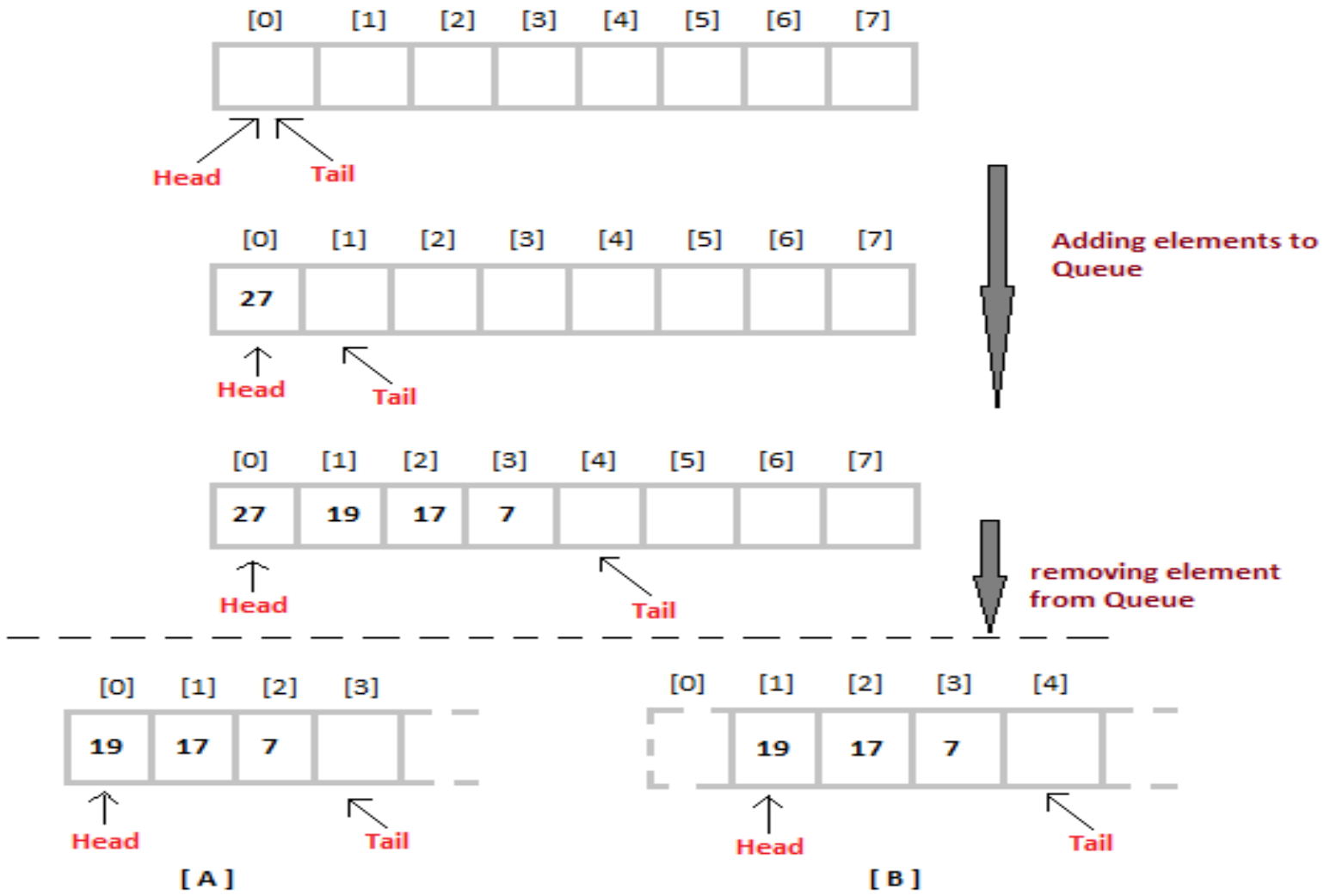
enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Insertion and Deletion

- Queue operations:



Implementation of queue using arrays

- Queue structure requires space, a pointer pointing to the front of the queue and a pointer pointing to the last element of the queue

```
struct mydef_queue{ int a[10];
                    int front;
                    int rear;
                  }

typedef struct mydef_queue mqueue;

mqueue q1,q2;
```

Different algorithms to implement queue functions Implementation_1

- Operations: **Empty, Full, Insert, Delete, Initialize**

Algorithm1: (queue of size n)

- ***Initialize*** front = 0 and rear = -1;
- ***Empty condition:*** (rear <front);
- **Full condition:** (rear == n-1);
- The number of elements in the queue at any time **=**

  rear-front+1;

- **Insert algo** : if (rear not equal to n-1) increment rear and add element at rear position /* no change in front */
- ***Delete algo***: if(front<rear) delete front element and increment front /* no change in rear */
- ***Problem: Queue may have empty spaces but will not show***

Implementation_2

- **Operations: Empty, Full, Insert, Delete, Initialize**

Algorithm2: (queue of size n)

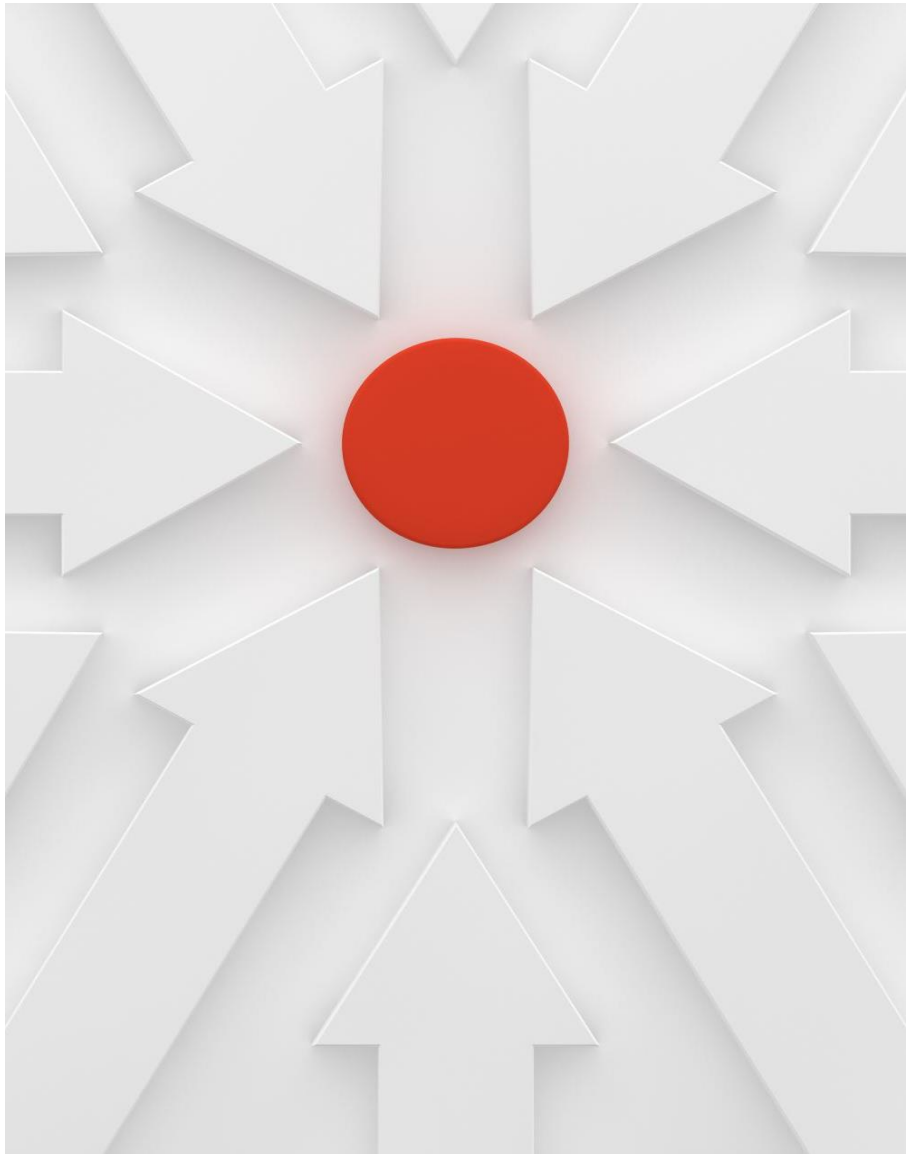
- ***Initialize*** front = 0 and rear = -1;
- ***Empty condition:*** (rear <front);
- **Full condition:** (rear == n-1);
- **Insert algo** : if (rear not equal to n-1) increment rear and add element at rear position
- ***Delete algo***: if(front<rear) delete front element ***and move forward all the remaining elements one position up and set rear = rear-1***
- ***Problem:*** *Every delete operation is O(n)*
- ***Advantage****: front is always 0, can remove one variable from the implementation*

Circular queue (implementation 3)

- **Operations**: *Empty, Full, Insert, Delete, Initialize*

Building Algorithm3: (queue of size n)

- Can we improve the first algorithm?
- ***View the queue as a circle rather than as a straight line***
- *Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make it a circle*.
- When the array holding queue is viewed as circular then it means **after location n-1 location 0** will come



How to implement the issue of circularity

- Whether structure of the Queue will change, or conditions will change??
- Issues!!!
- What will be the initial condition??
- Will the initial condition hold good correctly for empty queue condition in all situations??
- What will be the full queue condition??
- How will we change front and rear during insertion and deletion??

Implementation_ 1(Circular queue)

- **Operations**: Empty, Full, Insert, Delete, Initialize
- IF ***Initialize:*** front = 0 and rear = 0; ***Empty condition should be:*** (rear == front);
- To insert, we will first insert and then increment ***rear***
- To delete, first delete and then increment ***front***.
- Note: Addition is not simple addition, it is mod n

Check: If we insert n-1 elements in the queue then rear will be n-1. Problem??

Cont..

- Now, if we try to insert nth element, then value of ***rear will be 0*** (because it is a circle and we take module n)(so, ***front and rear*** both will be ***0***)
- **In that case, Full condition will be same as empty condition**
- ***Difficulty: we would not be able to distinguish between empty and full queue, but we would be able to add elements***
- ***So, what should we do??***

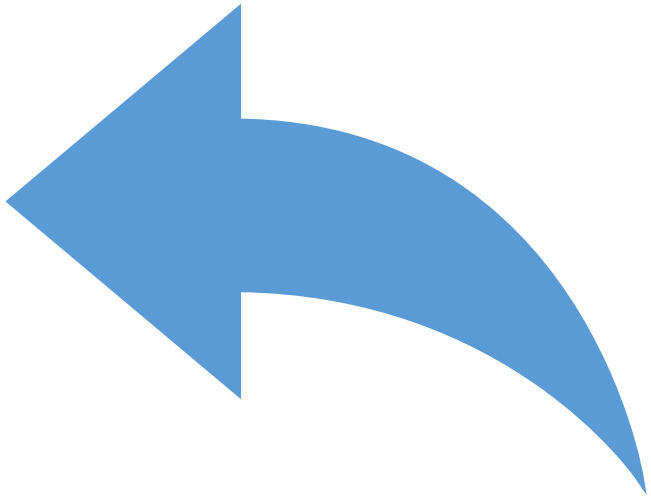
Final implementation for circular queue

- **Operations: E**mpty, Full, Insert, Delete, Initialize

If we sacrifice one memory location, then we can distinguish between ***empty and full queue***

- ***Initialize*** front = n-1 and rear = n-1; then ***Empty condition should be:*** (rear == front);
- **Full condition:** ?? If (rear+1 == front)
- **Insert algo** : Temporarily increment **rear**, check if (**rear not equal to front**) then actually **increment rear** and add element at **rear position**

  If (rear+1 == front) --- then it is **full** condition



Cont..

Delete algo: if(rear equal to front) then queue is empty otherwise ***first increment front and remove the front element***

In this implementation:

- **for *full queue*, we check condition (rear == front) after incrementing rear**
- **For *empty queue* we check condition (rear == front) without incrementing and *this results in sacrificing an element of the queue***

Writing functions …

Operations: Empty, Full, Insert, Delete, Initialize

```
#define max = 10;
struct mydef_queue{int front,rear;
                   int items[max];
};
typedef struct mydef_queue squeue;
void initialize( squeue *qs)
{ qs ->front = max-1; qs ->rear = max-1}
```

Cont…

```
int full(squeue *qs)
{int temp;
 if(qs->rear ==max-1)
 temp =0;
 else
 temp = qs->rear+1;
 if (qs ->front == temp)
return 1
 else return 0;}

int empty(squeue *qs)
{ if (qs ->front == qs ->rear) return 1 else return 0;}
```

Writing functions …

```
void insert( squeue *qs, int x)
{ if(!full(qs)) {if(qs->rear ==max-1)
                    qs->rear =0;
                   else qs->rear++;
                  qs->a[qs->rear] = x;
                 }
else printf("overflow\n");
}
int delete(squeue *qs)
{ if (!(empty)) {if(qs->front ==max-1) qs->front =0; else qs->front++;
                    return(qs->a[qs->front]);}
```

Writiing Queue program..

```
#include<stdio.h>
 #include<conio.h>
 #define SIZE 5
void main()
{ int choice, value;
myqueue mq;
Initialize(&mq);
while(1){ printf("\n****** MENU ******\n");
printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
```

```
printf("Enter your choice: ");
scanf("%d",&choice);
switch(choice){ case 1: printf("\nEnter the value to be inserted: ");
scanf("%d",&value); insert(&mq,value); break;
case 2: delete(&mq); break;
 case 3: display(); break;
case 4: exit(0);
default: printf("\nPlease select the correct choice!!!\n"); } } }
```

Complexity of insertion and deletion in queue

Some constant number of operations will be done irrespective of queue size. We can say **c** operations will be required for each insertion and deletion operation, so complexity in terms of Big(O) will be O(1)

O(1): means some constant number of operations

Priority Queue

- Simple Queue is FIFO
- Priority Queue: A priority is attached to each item and Deletion is done on the basis of that priority. Insertion can be done arbitrarily.
- A Priority Queue is a data structure in which intrinsic ordering of the elements does determine the results of its basic operations.
- Ascending Priority Queue : Element having smallest value as highest priority is deleted
- Descending Priority Queue: Element having largest value as highest priority is deleted

Implementation(ascending queue)

Algo1:

- keep the elements in an array **randomly**
- Insert at the end of array (**complexity O(1))**
- Delete the element having lowest priority value {**O(n) + move the elements}**

Algo2:

- keep the elements in an array in **sorted order** (**ascending order**)
- Insert the element at right place {**complexity O(n)** + moving of the elements}
- Delete the front element (**complexity O(1))**

Queue Exercises

- Implement railway reservation
- Implement allocation of resources in a computer system