Queue
Data Structure

Data Structure : Queue

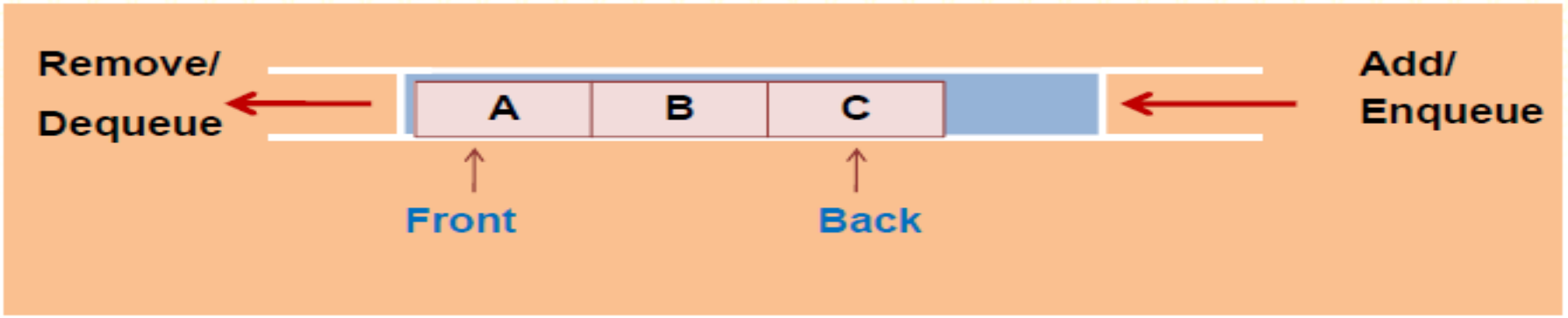
1. What is queue?
2. Operation on queues
   - Queue implementation
   - Implementation of insert and delete operations on a queue
3. Limitations of linear queues
4. Circular queues
   - Operations on circular queues
   - Implementation of insert and delete operations on a circular queue
5. Application

What is Queue?

- Ordered collection of elements in which the elements are **added at one end, called the rear, and deleted from the other end, called the front.**
- It is characterized by FIFO (First In First Out) principle.

- All elements are entered into the queue from a fix end called as **Rear** whereas all the elements are removed from an opposite end called as **Front** of the queue.

Implementation of Queue

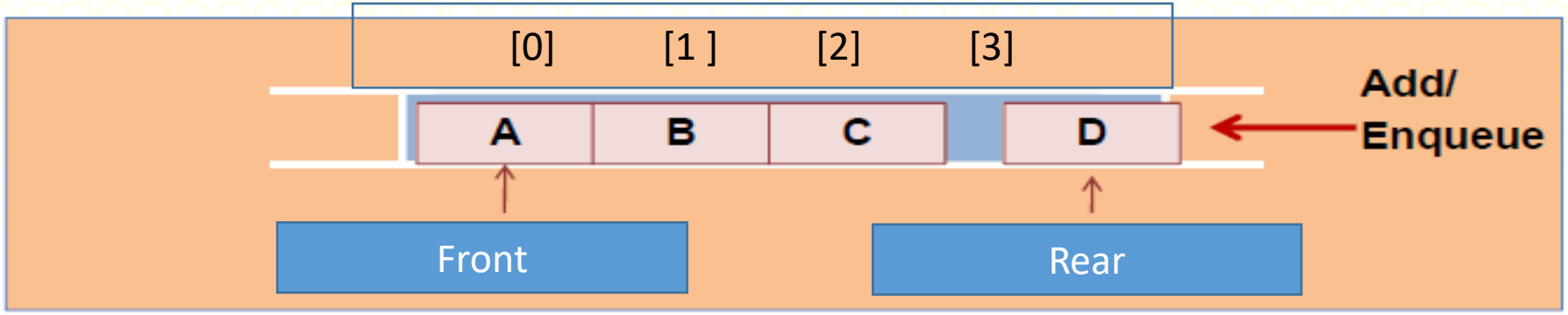


Basic Structure of a Queue:

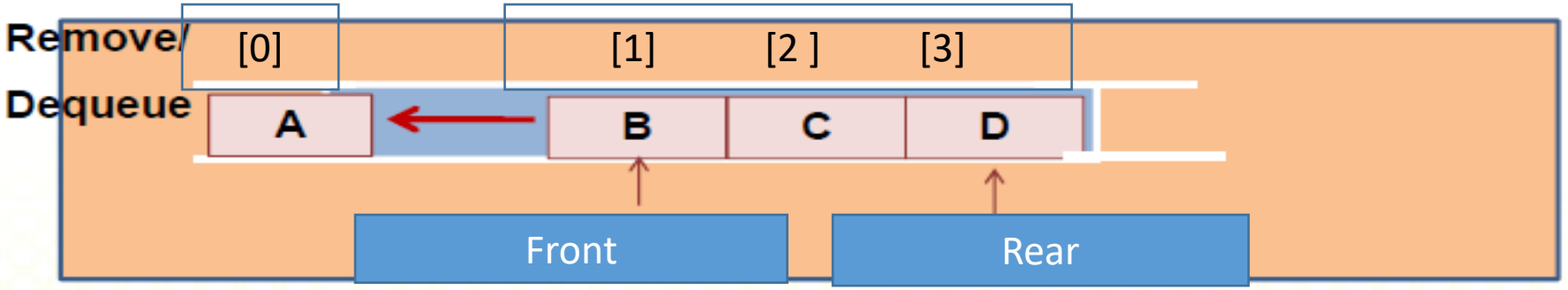
-Data structure that hold the queue

-front

-back/rear

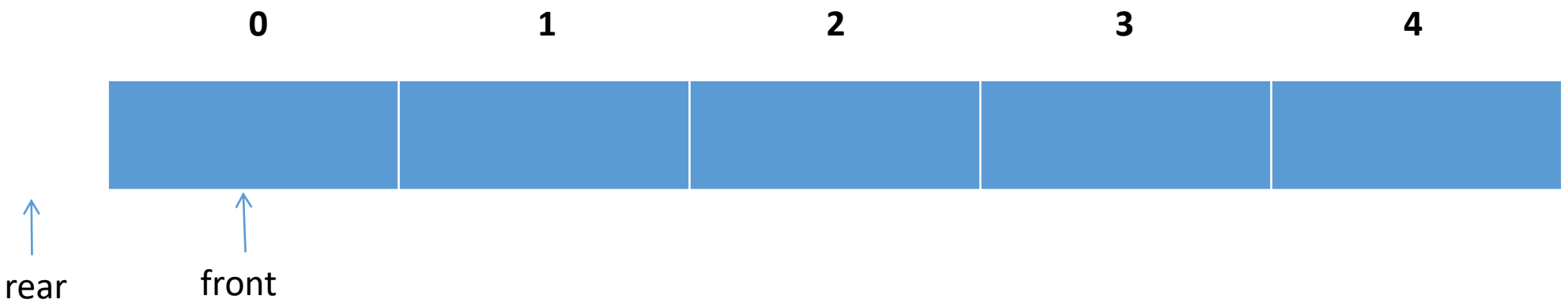


Insert D into Queue (enQueue) : D is inserted at rear



Delete from Queue (deQueue) : A is removed

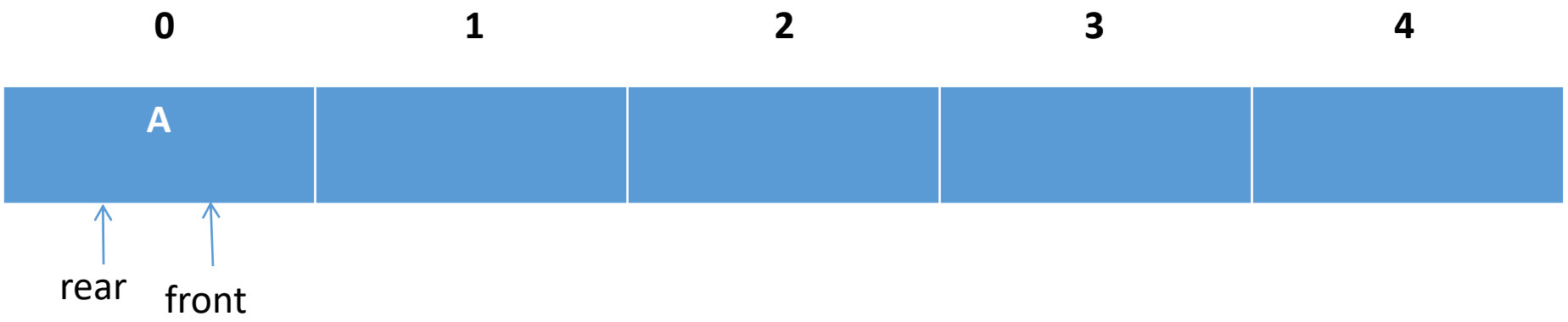
Linear Queue



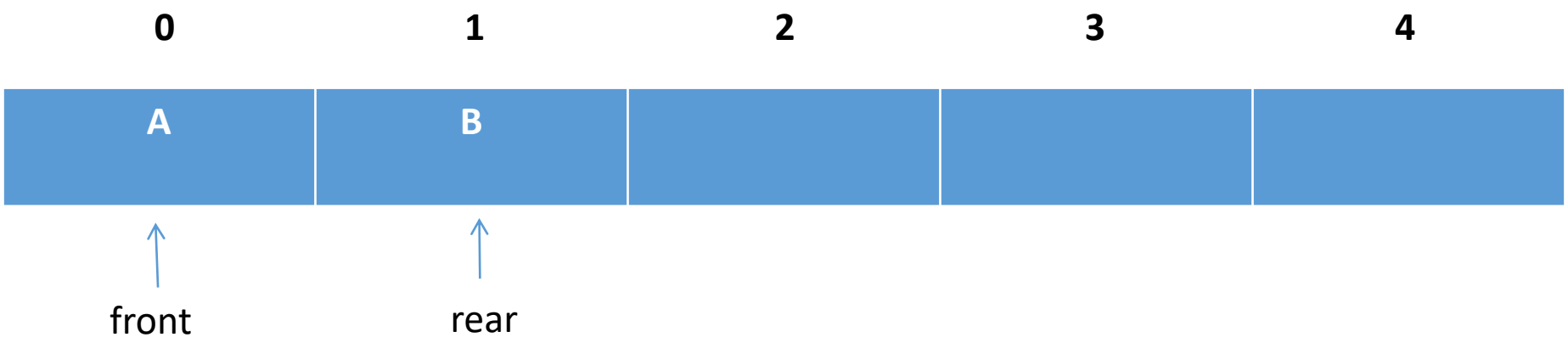
Queue Size :5

Status of the queue : QUEUE IS EMPTY

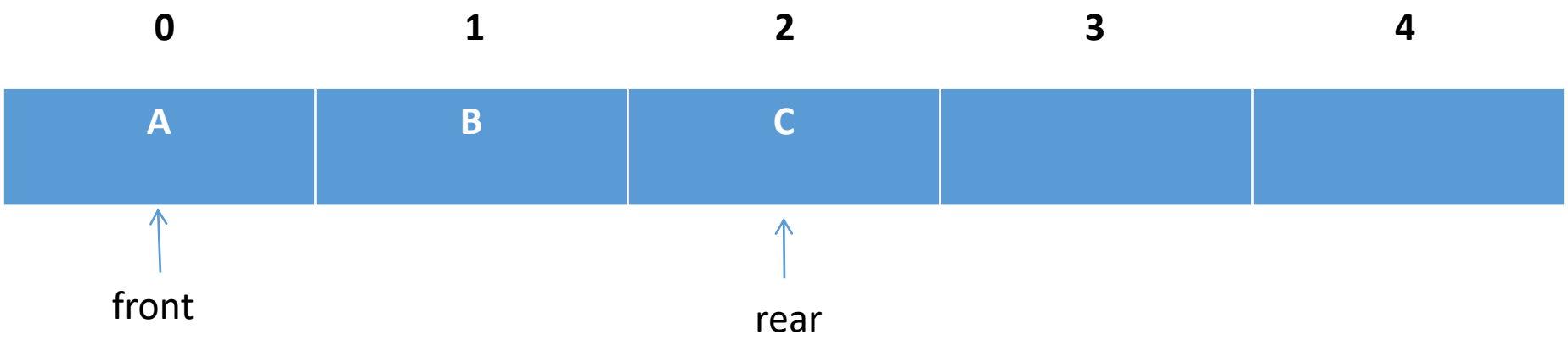
Linear Queue- Insert(A)



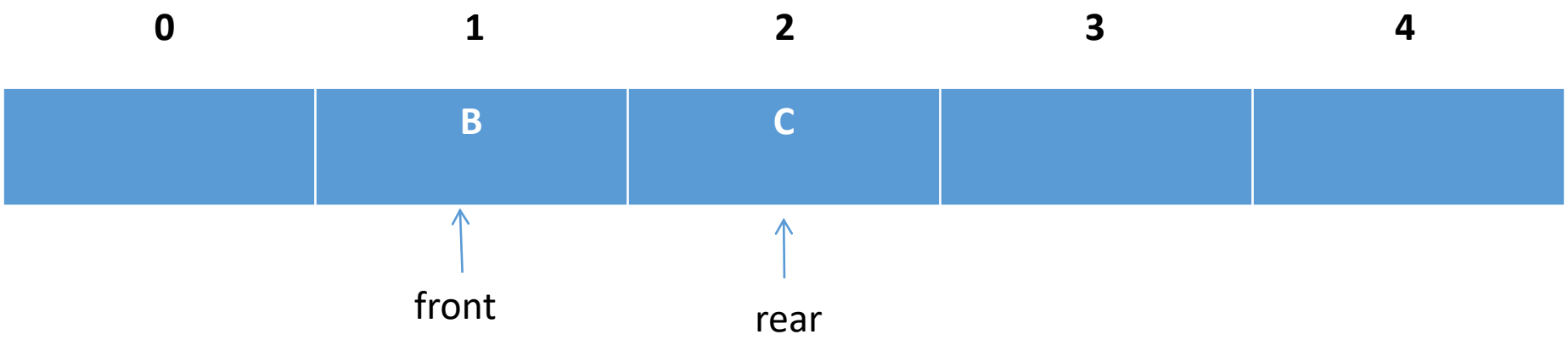
Linear Queue- Insert(B)



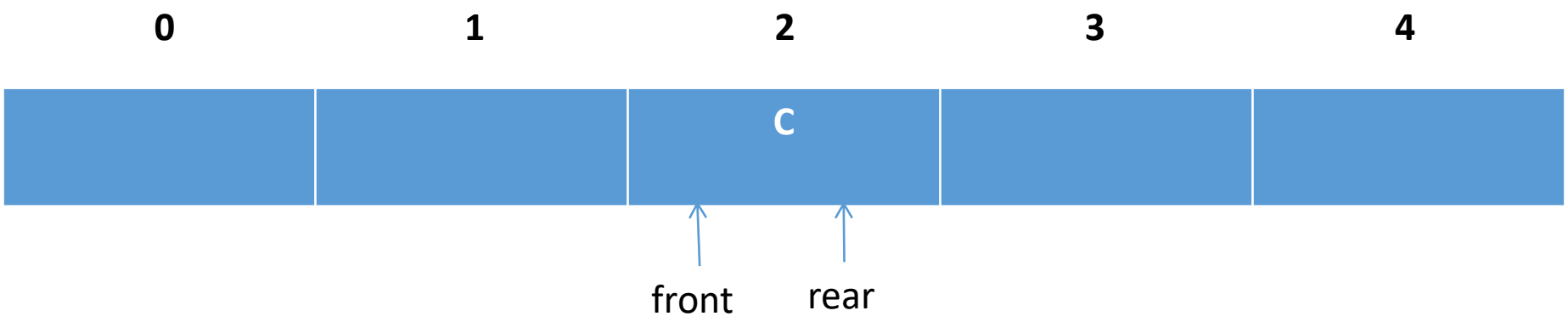
Linear Queue- Insert(C)



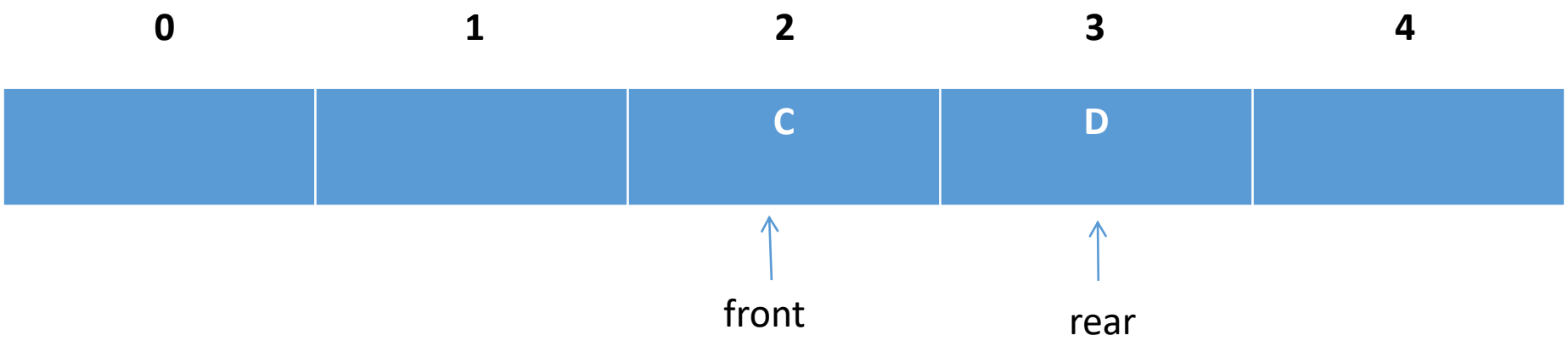
Linear Queue- Delete()



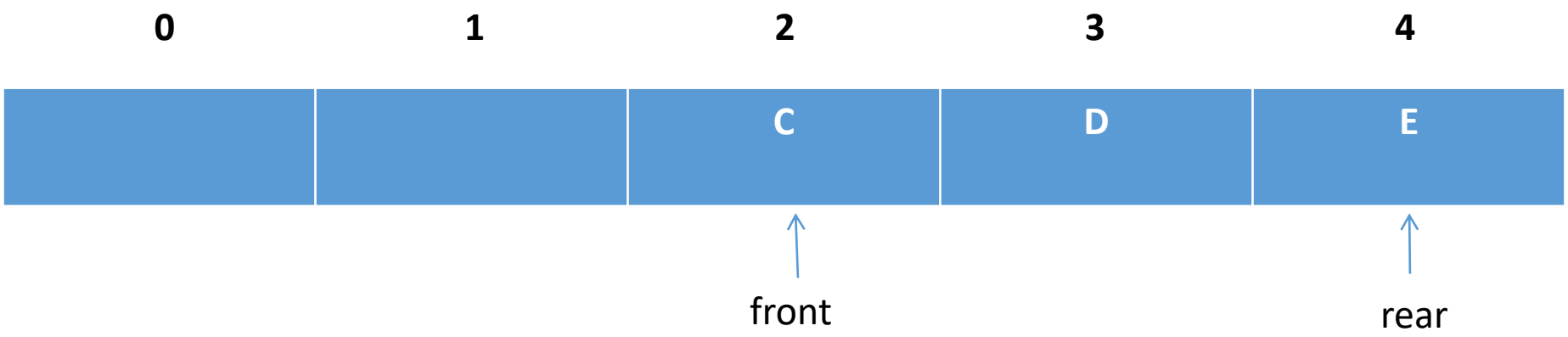
Linear Queue- Delete()



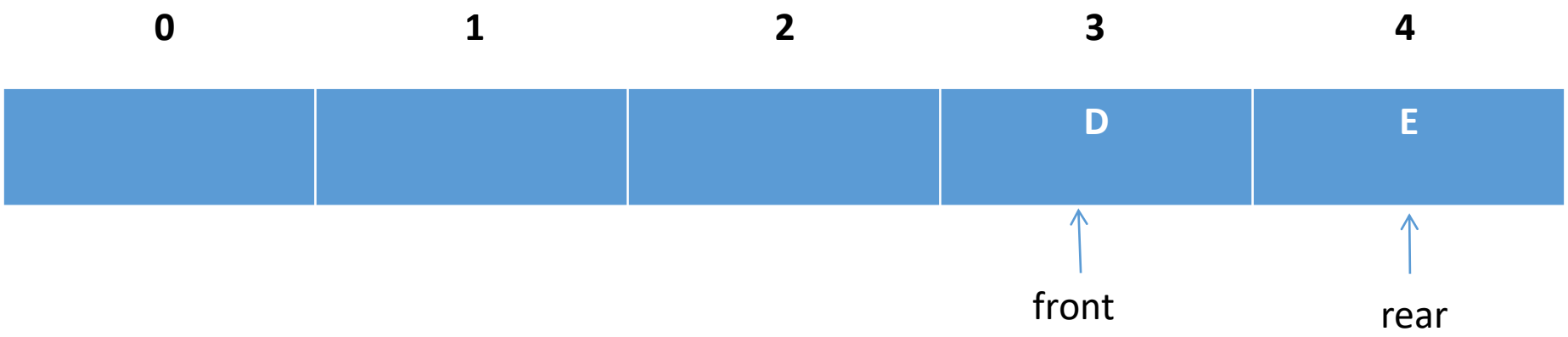
Linear Queue- Insert(D)



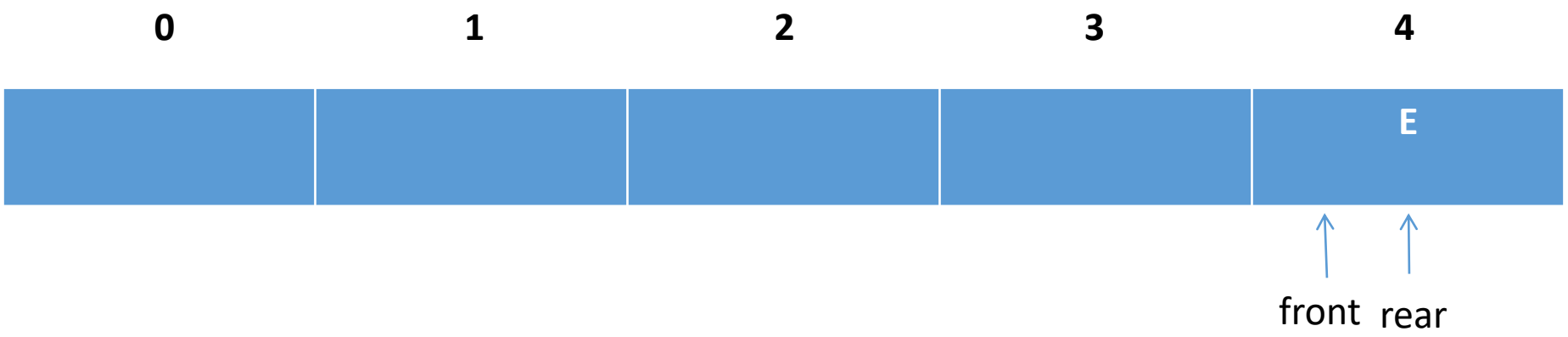
Linear Queue- Insert(E)



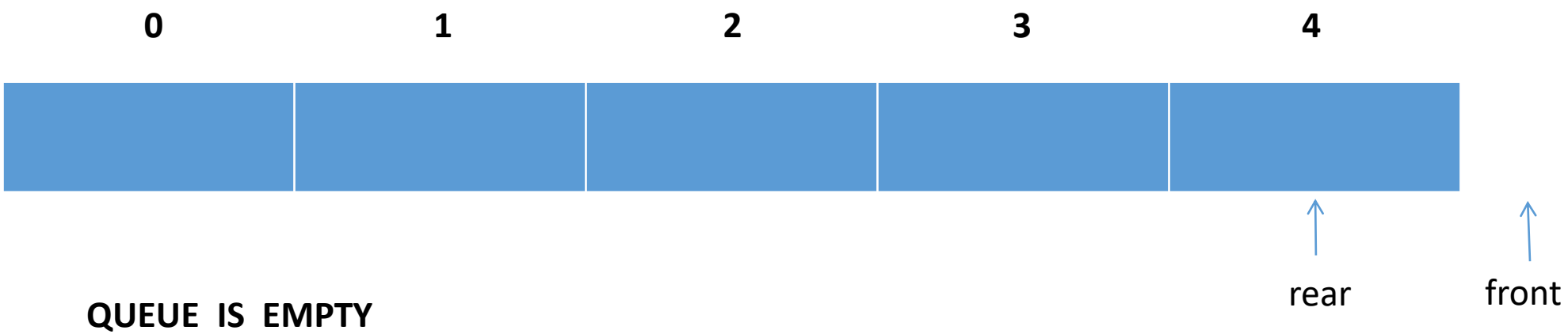
Linear Queue- Delete()



Linear Queue- Delete()

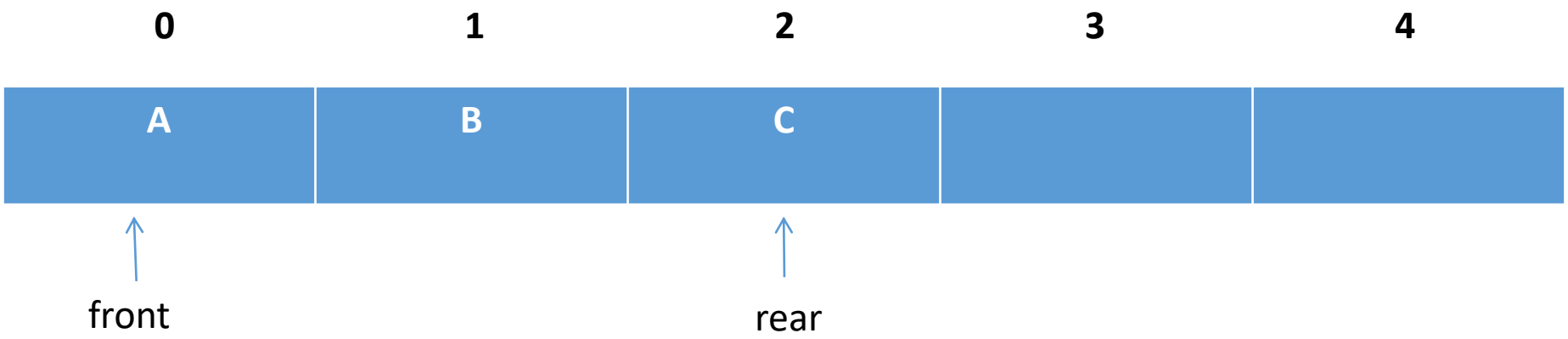


Linear Queue- Delete()

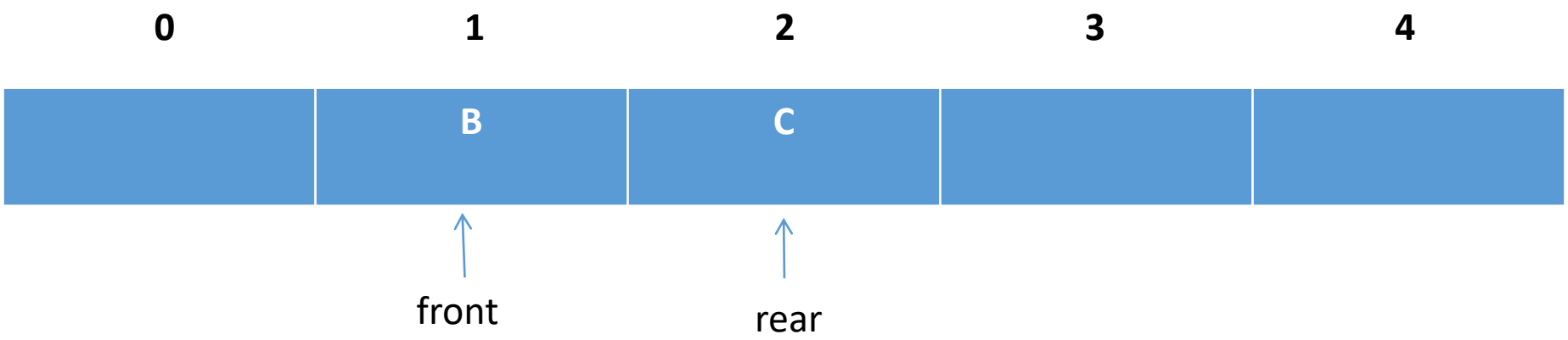


QUEUE IS EMPTY

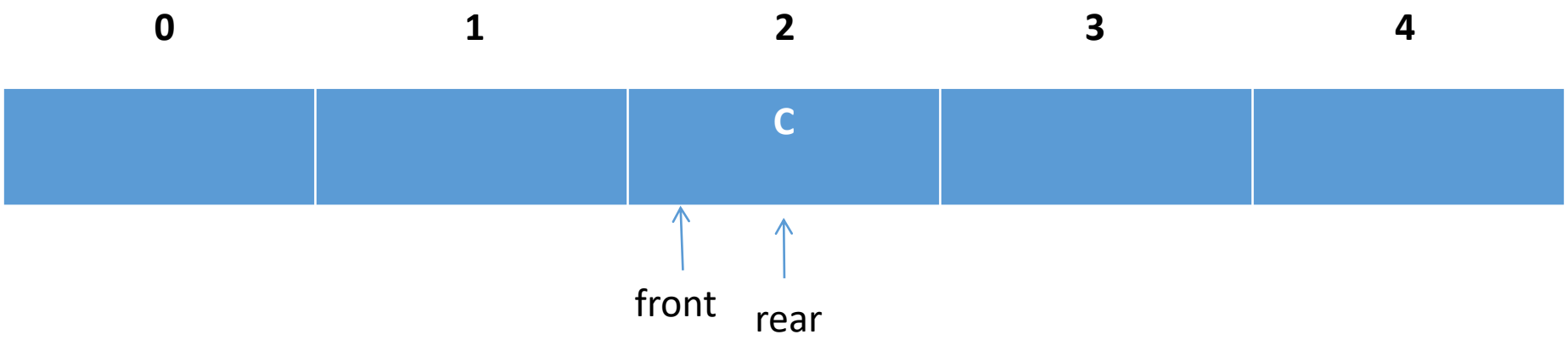
Linear Queue



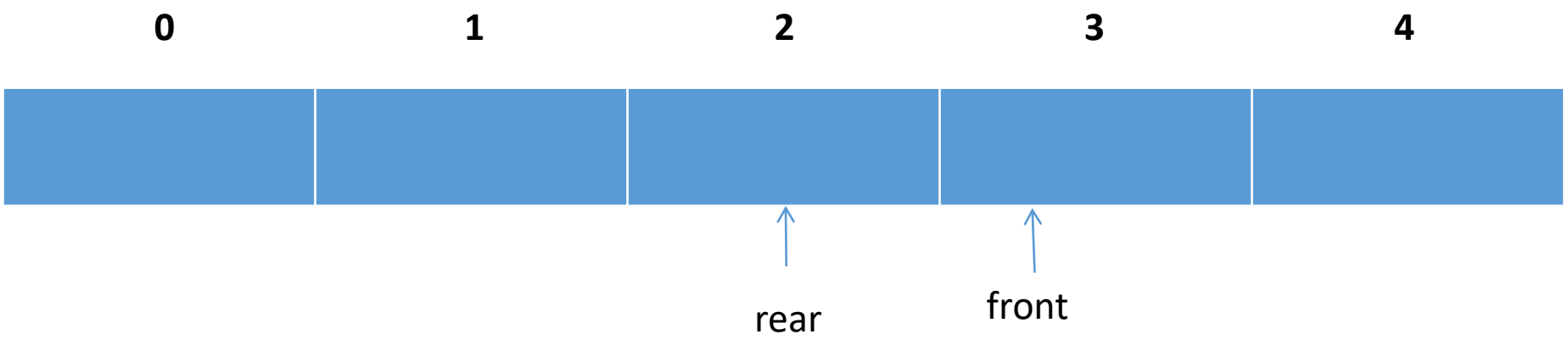
Linear Queue-Delete



Linear Queue-Delete

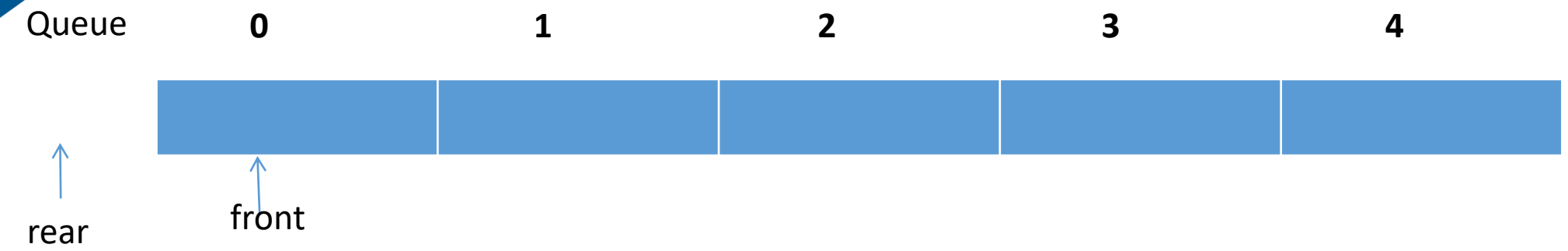


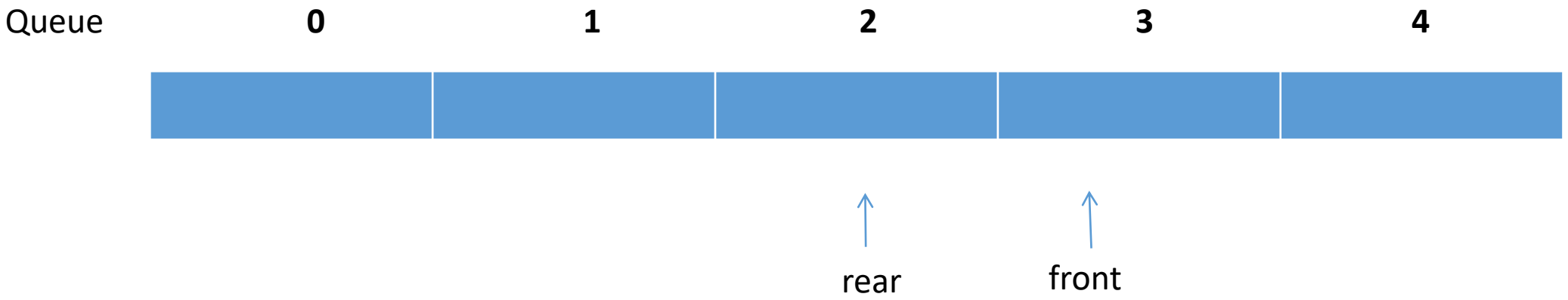
Linear Queue-Delete



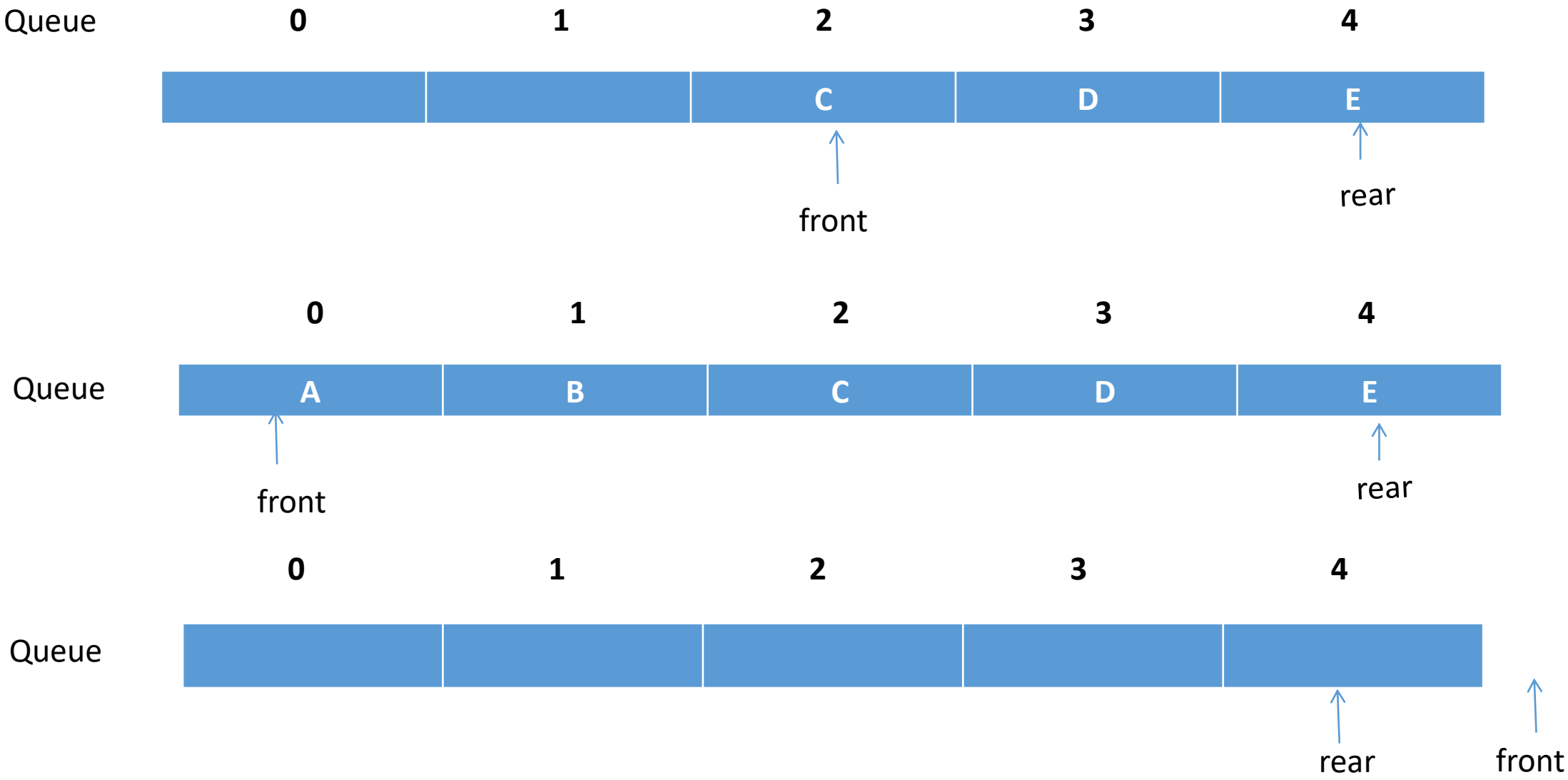
QUEUE IS EMPTY

Different cases of **Queue Empty Status** of a Linear Queue





Different cases of Queue full status



Queue Operations

- **Primitive Operations:**
  - **Insert**: add a value at the rear end of the queue(**enqueue**)
  - **Delete**: remove a value from the front end of the queue(**dequeue**)
- Other Operations:
  - `isFull`: `true` if the queue is currently full, *i.e.*, has no more space to hold additional elements
  - isEmpty: `true` if the queue currently contains no elements
  - queueFront : `returns` the front element of the queue

Linear Queue Implementation

```
#define QUEUESIZE  100
struct queue
 {
    int items[QUEUESIZE]   ;
    int front ;
    int rear ;
};
struct queue lq ;

lq.front = 0;
lq.rear = -1
```

Definition of QUEUE

Declaration of QUEUE

Initialization of QUEUE, empty QUEUE

QUEUE : Implementation of primitive operations Insert & Delete

```
void enqueue( struct queue *q ,int x)
{
    if(q->rear == QUEUESIZE-1 )
    {
        printf("QUEUE OVERFLOW");
        exit(0);
    }
   q->rear ++;
   q->items[q->rear] = x;
}
```

Function Call : enqueue(&lq,y)

Linear Queue : Implementation of primitive operations Insert & Delete

```
int dequeue(struct queue *q)
{
    int x;
    if(q->front  >  q->rear)
    {
          printf("QUEUE UNDERFLOW");
          exit(0);
    }
    x= q->items[q->front];
    q->front ++;
    return x;
}
```

Stack : Implementation of operations isEmpty & isFull

```
int isEmpty( struct queue *q )
{
    if(q->front  >  q->rear)
          return 1;
    return 0;
}

int isFull(struct queue *q )
{
   if(q->rear == QUEUESIZE-1 )
        return 1;
    return 0;
}
```

```
int queueFront(struct queue *q)
{
    int x;
    if(q->front  >  q->rear)
    {
          printf("QUEUE UNDERFLOW");
          exit(0);
    }
    x= q->items[q->front];
    return x;
}
```

Display Function

```
void display( struct stack  *q)
{
    int I ;
    if(q->front  >  q->rear)
    {
          printf("QUEUE UNDERFLOW");
          return;
    }
    printf("Queue elements are \n");
    for(i =q->front; i<=q->rear; i++)
        printf("   %d",q->items[i]);
}
```

Disadvantages of linear queue

- On deletion of an element from existing queue, front pointer is shifted to next position.
- This results into virtual deletion of an element.
- By doing so memory space which was occupied by deleted element is wasted and hence inefficient memory utilization is occur.

Limitation of Linear Queues

- By the definition of a queue, when we add an element in Queue, rear pointer is increased by 1 whereas, when we remove an element front pointer is increased by 1. But in array implementation of queue this may cause problem as follows:

- Consider operations performed on a Queue (with SIZE = 5) as follows:

- 1. Initially empty Queue is there so,front = 0 and rear = -1

- 2. When we add 5 elements to queue, the state of the queue becomes as follows with
- front = 0 and rear = 4

10	20	30	40	50

3. Now suppose we delete 2 elements from Queue then, the state of the Queue becomes as follows, with front = 2 and rear = 4

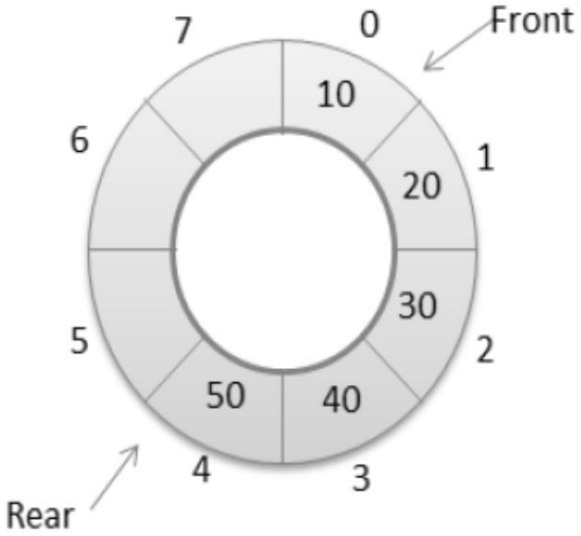
		30	40	50

4. Now, actually we have deleted 2 elements from queue so, there should be space for another 2 elements in the queue, but as rear pointer is pointing at last position and Queue overflow condition (Rear == SIZE-1) is true, we can't insert new element in the queue even if it has an empty space.

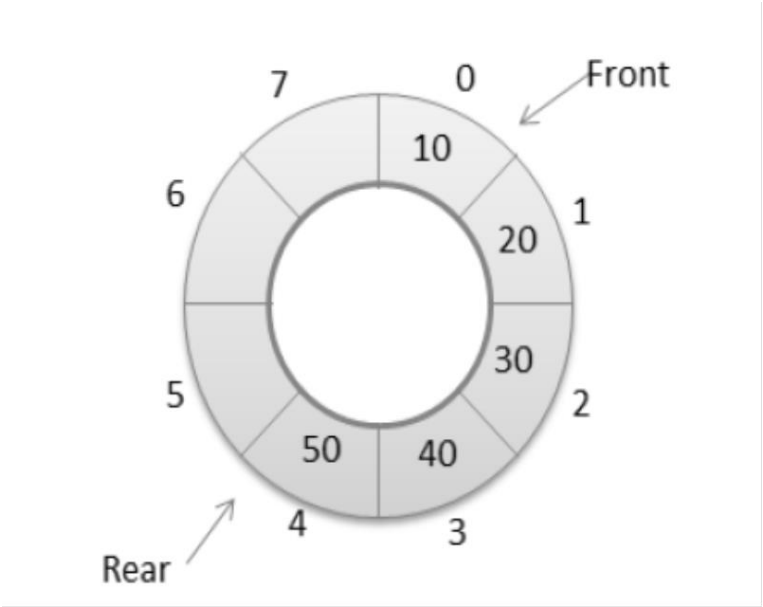
To overcome this problem there is another variation of queue called CIRCULAR QUEUE.

Overcome disadvantage of linear queue:

- To overcome disadvantage of linear queue, **circular queue** is use.
- We can solve this problem by joining the front and rear end of a queue to make the queue as a circular queue .
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.

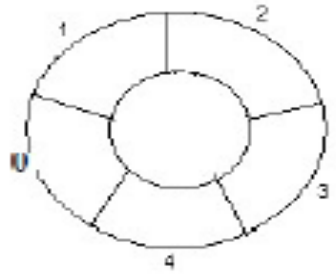


Circular Queue

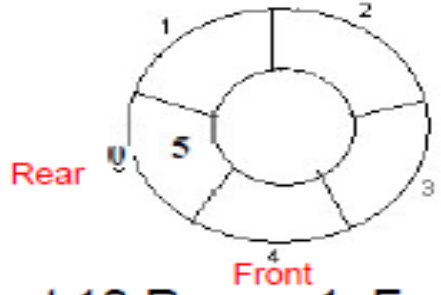


Example: Consider the following circular queue with N = 5.

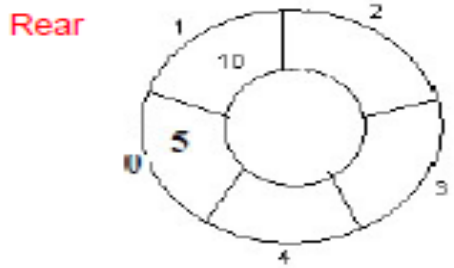
1. Rear = size-1, Front = size-1.



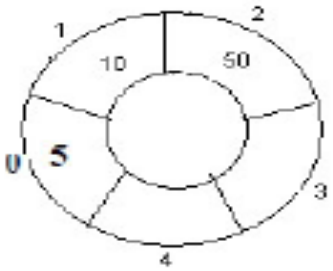
2. Insert 5 Rear = 0, Front = size-1.



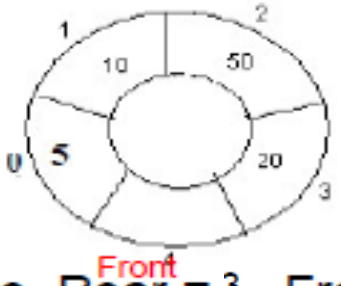
3. Insert 10 Rear = 1, Front = size-1.



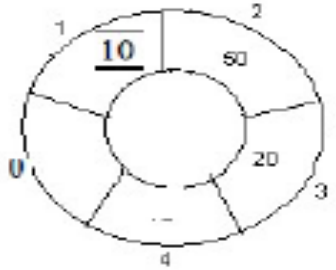
4. Insert 50, Rear = 2 Front = size-1.



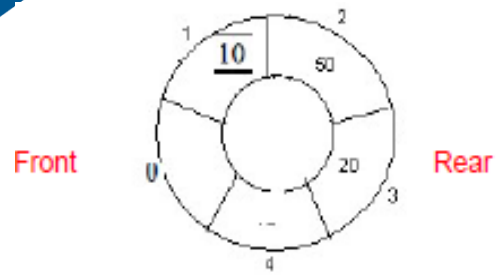
5. Insert 20, Rear = 3, Front = size-1.



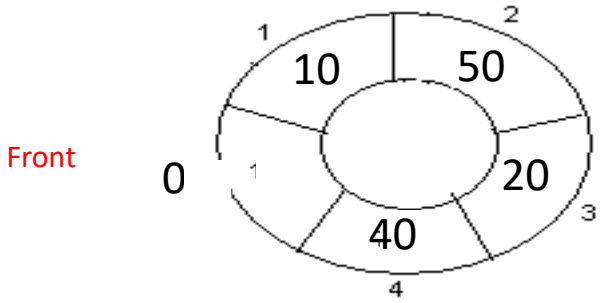
6. Delete Rear = 3, Front = 0



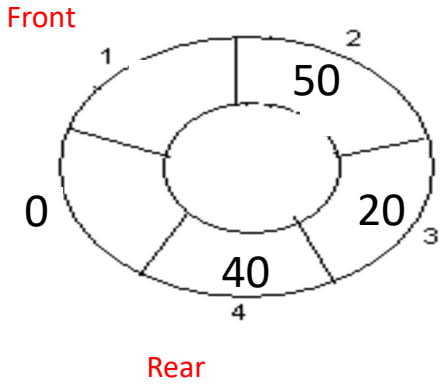
Rear = 3, Front = 0



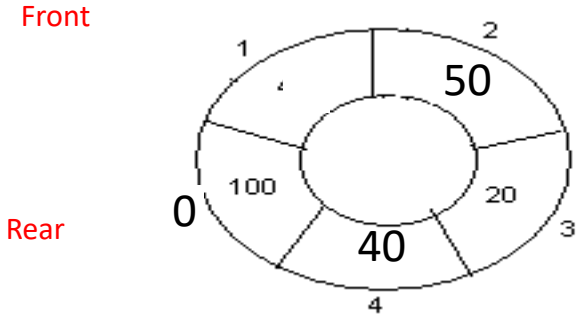
8. Insert 40, Rear = 4 Front = 0.



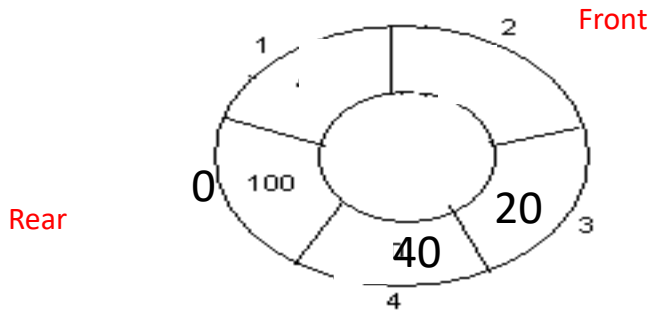
9. Delete, Rear = 4, Front = 1



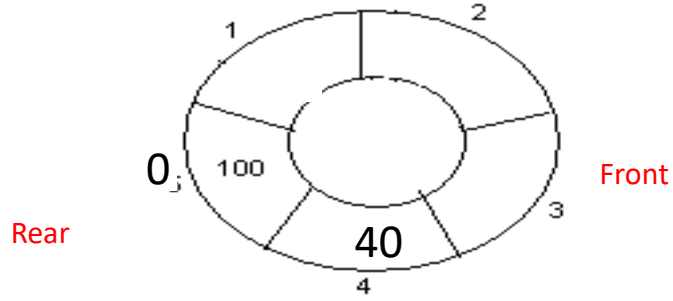
10. Insert 100, Rear = 0, Front = 1.



11. Delete, Rear = 0, Front = 2

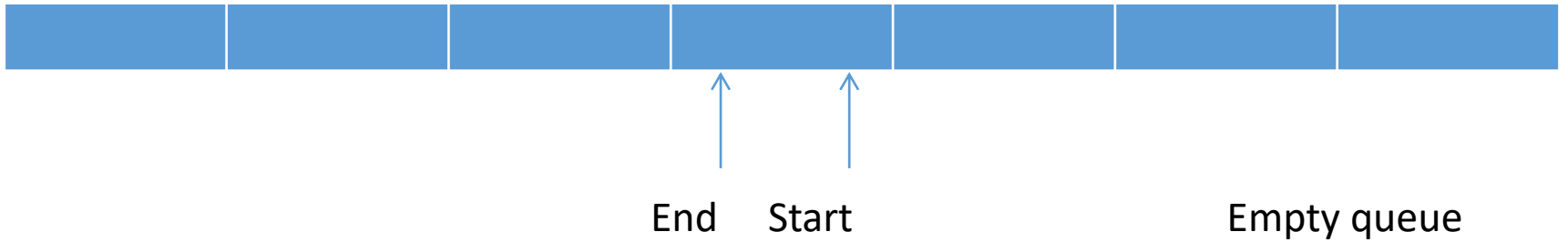


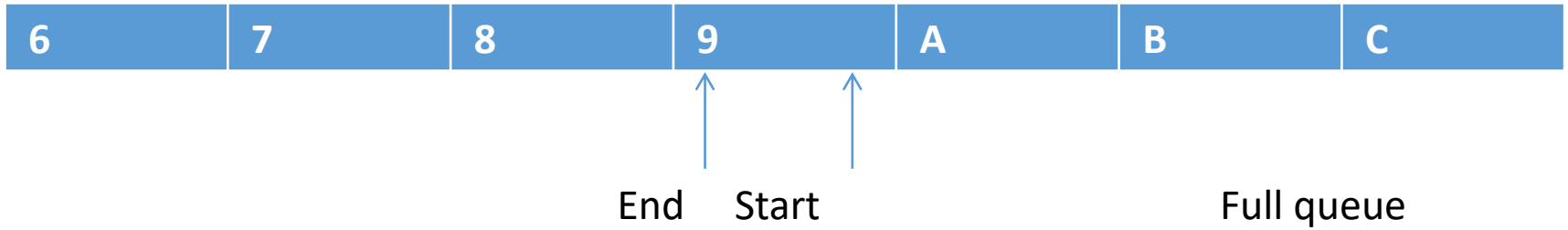
12. Delete , Rear = 0, Front = 3.



Circular queue

- There is a problem with this: Both an empty queue and a full queue would be indicated by having the head and tail point to the same element.





Circular Queue Implementation(with count)

```
#define QUEUESIZE  100
struct queue
 {
    int items[QUEUESIZE]   ;
    int front ;
    int rear ;
    int count;
};

struct queue cq ;

cq.front = QUEUESIZE-1;
cq.rear = QUEUESIZE-1;
```

Definition of QUEUE

Declaration of QUEUE

Initialization of QUEUE, empty QUEUE

Circular QUEUE : Implementation of primitive operations Insert (with count)

```
void enqueue( struct queue *q,int x )
{
    if( q->count == QUEUESIZE )
    {
        printf("QUEUE OVERFLOW");
        exit(0);
    }
   q->rear = (q->rear +1)% QUEUESIZE;
   q->items[q->rear] = x;
   q->count++;
}
```

Circular Queue : Implementation of primitive operations Delete(with count)

```
int dequeue(struct queue *q)
{
    int x;
    if(q->count==0)
    {
        printf("QUEUE UNDERFLOW");
        exit(0);
    }
   q->front = (q->front +1)%QUEUESIZE;
   x= q->items[q->front];
    q->count--;
    return x;
}
```

Circular Queue : Implementation of operations isEmpty & isFull(with count)

```
int isEmpty( struct queue *q )
{
    if(q->count==0)
        return 1;
    return 0;
}

int isFull(struct queue *q )
{
   if(q->count == QUEUESIZE )
       return 1;
    return 0;
}
```

```
int queueFront(struct queue *q)
{
    int x;
    if(q->count  ==  0)
    {
        printf("QUEUE UNDERFLOW");
        exit(0);
    }
    x= q->items[q->front];
    return x;
}
```

Circular Queue Display Function(with count)

```
void display( struct queue  *q)
{
    int i,pos ;
    if(q->count==0)
    {
          printf("QUEUE UNDERFLOW");
          return;
    }
    printf("Queue elements are \n");
    pos = (q->front +1)%QUEUESIZE;
    for(i =1; i<=q->count; i++)
     {
         printf("   %d",q->items[pos]);
         pos = (pos +1)%QUEUESIZE;
     }
}
```

Curcular Queue Implementation(with out count)

```
#define QUEUESIZE  100
struct queue
 {
    int items[QUEUESIZE]   ;
    int front ;
    int rear ;
};
struct queue cq ;
cq.front = QUEUESIZE-1;
cq.rear = QUEUESIZE-1;
```

Definition of QUEUE

Declaration of QUEUE

Initialization of QUEUE, empty QUEUE

Circular QUEUE : Implementation of primitive operation Insert (with out count)

```
void enqueue( struct queue *q,int x )
{
    if( (q->rear +1) % QUEUESIZE == q->front )
    {
        printf("QUEUE OVERFLOW");
        exit(0);
    }
   q->rear = (q->rear +1)% QUEUESIZE;
   q->items[q->rear] = x;
}
```

Circular Queue : Implementation of primitive operation Delete (with out count)

```
int dequeue(struct queue *q)
{
    int x;
    if(q-> rear== q->front )
    {
        printf("QUEUE UNDERFLOW");
        exit(0);
    }
   q->front = (q->front +1)%QUEUESIZE;
   x= q->items[q->front];
   return x;
}
```

Circular Queue : Implementation of operations isEmpty & isFull(with out count)

```
int isEmpty( struct queue *q )
{
    if(q-> rear== q->front )
          return 1;
    return 0;
}

int isFull(struct queue *q )
{
 if((q->rear +1) % QUEUESIZE == q->front )
        return 1;
    return 0;
}
```

```
int queueFront(struct queue *q)
{
    int x;
     if(q-> rear== q->front )
      {
            printf("QUEUE UNDERFLOW");
            exit(0);
    }
    x= q->items[q->front];
    return x;
}
```

Circular Queue Display Function(With out count)

```
void display( struct queue  *q)
{    int i ,pos;
     if(q-> rear== q->front )
     {
           printf("QUEUE UNDERFLOW");
           return;
     }
     printf("Queue elements are \n");
     pos = (q->front +1)%QUEUESIZE;
     do
      {
          printf("   %d",q->items[pos]);
          pos = (pos +1)%QUEUESIZE;
     }while(pos != q->rear);
}
```

Application of Queues

- Direct applications
  - Waiting lines: Queues are commonly used in systems where waiting line has to be maintained for obtaining access to a resource. For example:
    - an operating system may keep a queue of processes that are waiting to run on the CPU.
    - Access to shared resources (e.g., printer)
    - Simulation of real-world situations, e.g., determine how many tellers to have in order to serve each customer within "reasonable" wait time.
  - Multiprogramming

- Indirect applications
  - Auxiliary data structure for algorithms
  - Component of other data structures
    - Example: Tree and Graph Breadth-First traversal

Message Queue Implementation

```
#include<stdio.h>
#include<stdlib.h>
#include<string.>
#define QUEUESIZE  100
struct queue
 {
    char items[QUEUESIZE] [200 ]    ;
    int front ;
    int rear ;
    int count;
};
struct queue mcq ;
mcq.front = QUEUESIZE-1;
mcq.rear = QUEUESIZE-1;
```

Definition of QUEUE

Declaration of QUEUE

Initialization of QUEUE, empty QUEUE

Message Queue Implementation :Insert

```
void enqueue( struct queue *q, char  msg[ ] )
{
    if( q->count == QUEUESIZE )
    {
        printf("QUEUE OVERFLOW");
        exit(0);
    }
   q->rear = (q->rear +1)% QUEUESIZE;
   strcpy( q->items[q->rear] , msg);
   q->count++;
}
```

Message Queue Implementation :Delete

```
void  dequeue(struct queue *q, char msg [ ])
{
   if(q->count==0)
    {
          printf("QUEUE UNDERFLOW");
          exit(0);
    }
   q->front = (q->front +1)%QUEUESIZE;
   strcpy(msg,q->items[q->front]);
    q->count--;
}
```

Message Queue Implementation: Display

```
void display( struct queue  *q)
{   int i,pos ;
    if(q->count==0)
    {
          printf("QUEUE UNDERFLOW");
          return;
    }
    printf("Messages in Queue are \n");
    pos = (q->front +1)%QUEUESIZE;
    for(i =1; i<=q->count; i++)
     {
         printf("   %s",q->items[pos]);
         pos = (pos +1)%QUEUESIZE;
     }
}
```

THANK YOU