ADITYA COLLEGE OF ENGINEERING & TECHNOLOGY

DATA STRUCTURES

UNIT III : Queues & Stacks

Topic: Queues

Branch: I-II IT

T. Srinivasulu

Dept. of Information Technology

Aditya College of Engineering & Technology

Surampalem.

Contents

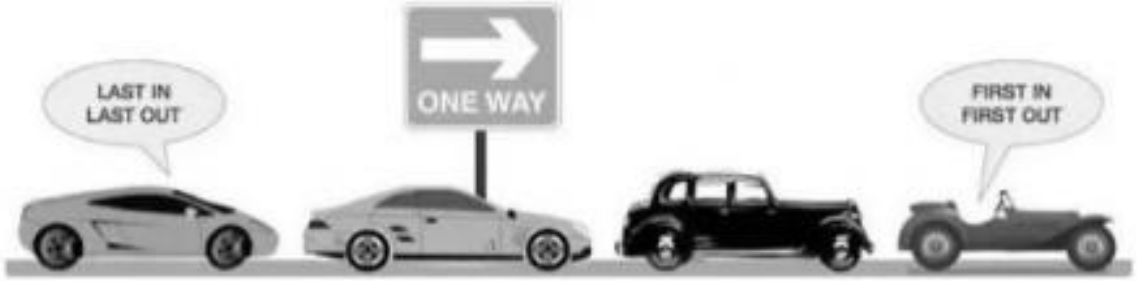
Queues:

- Introduction to Queues
- Representation of Queues-using Arrays
- Implementation of Queues-using Arrays
- Representation of Queues-using Linked list
- Implementation of Queues- using Linked list
- Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.

Stacks:

- Introduction to Stacks
- Array Representation of Stacks, Operations on Stacks,
- Linked list Representation of Stacks, Operations on Linked Stack,
- Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

Introduction to Queues



Introduction to Queues

Let us explain the concept of queues using the analogies given below.

- People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.
- Luggage kept on conveyor belts. The bag which was placed first will be the first to come out at the other end.
- Cars lined at a toll bridge. The first car to reach the bridge will be the first to leave.

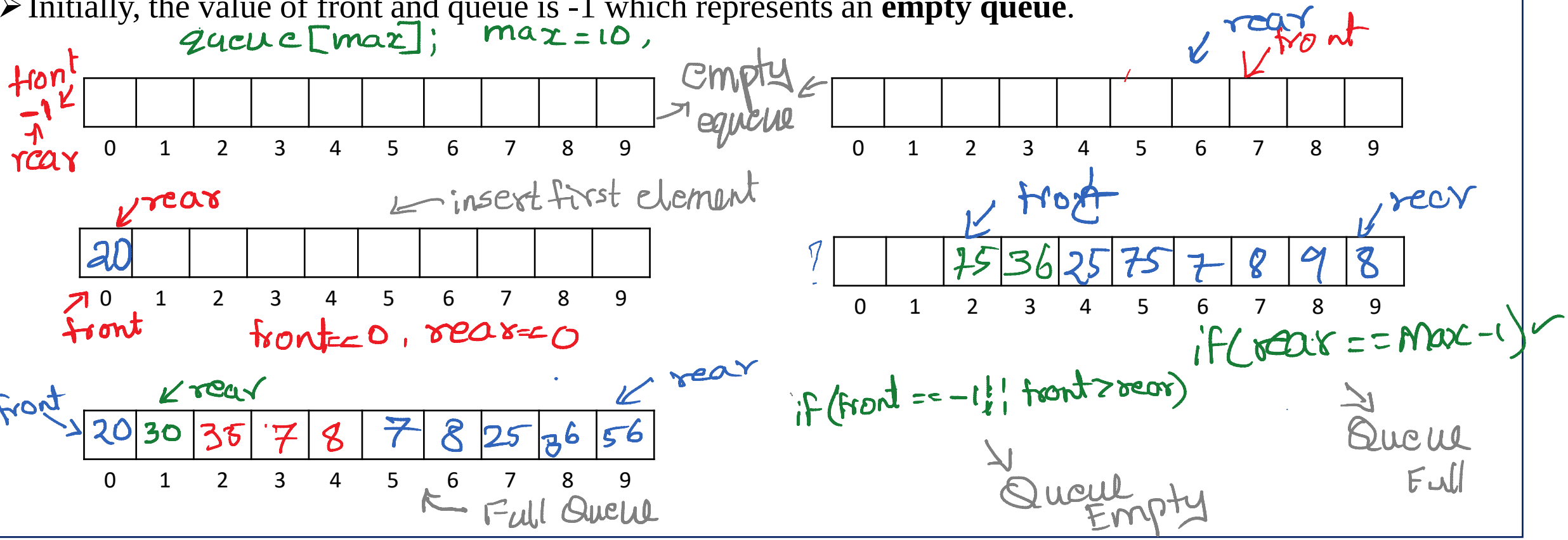
A **queue** is a **FIFO** (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.

The elements in a queue are added at one end called the **REAR** and removed from the other end called the **FRONT.**

A queue is a lineal list in which insertions (also called additions) and removals (also called deletions) takes place at different ends. The end at which new elements are add is called the **back or rear**, and that from which old elements are deleted is called the **front**.

ARRAY REPRESENTATION OF QUEUES

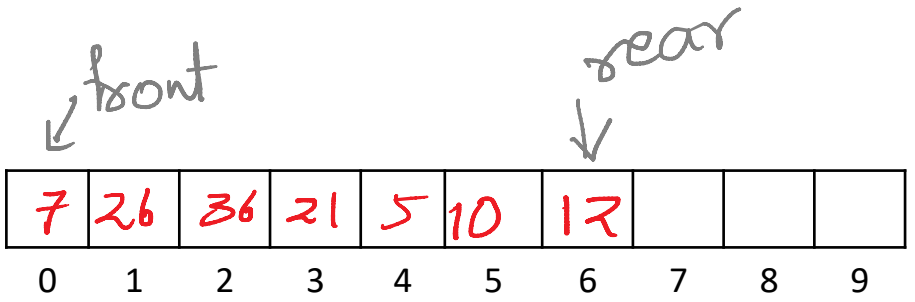
- We can easily represent queue by using **linear arrays**.
- There are two variables i.e. **front** and **rear**, that are implemented in the case of every queue.
- Front and rear variables point to the position from where insertions and deletions are performed in a queue.
- Initially, the value of front and queue is -1 which represents an **empty queue**.



Implementation of Queues-using Arrays

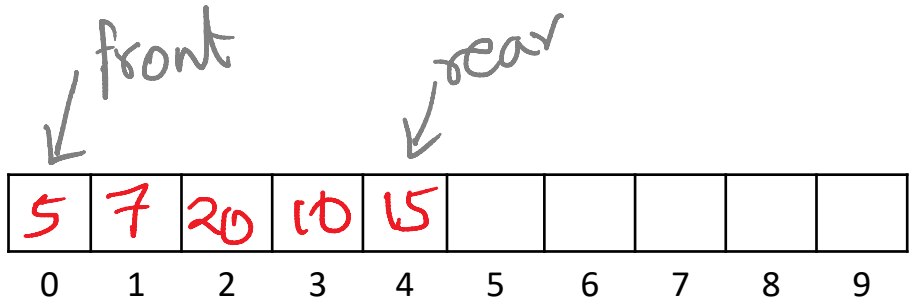
Implementation of queue operations are as follows:

1. Insert an element(Enqueue)
2. Delete an element(Dequeue)
3. isEmpty operation
4. isFull operation
5. Peek (front element)
6. Display the queue



Enqueue operation using Arrays

- Enqueue means inserting an element in the queue.
- Before inserting an element in a queue, we must check for **overflow** conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1.
- In case the queue is empty, then both FRONT and REAR are set to zero, so that the new value can be stored at the 0th location.
- if the queue already has some values, then REAR is incremented so that it points to the next location in the array



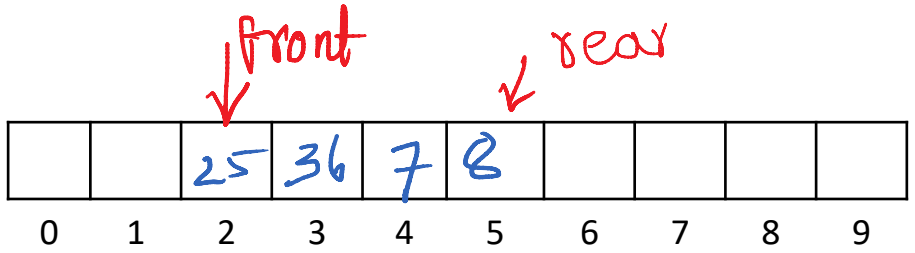
Algorithm for Enqueue operation

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Algorithm to insert an element in a queue

Dequeue operation using Arrays

- Dequeue means removing an element from the queue.
- Before removing an element from a queue, we must check for **underflow** conditions. An underflow will occur when we try to delete an element from a queue when it is empty. An underflow occurs if FRONT = –1 or FRONT > REAR.
- If queue has some values, then FRONT is incremented so that it now points to the next value in the queue.

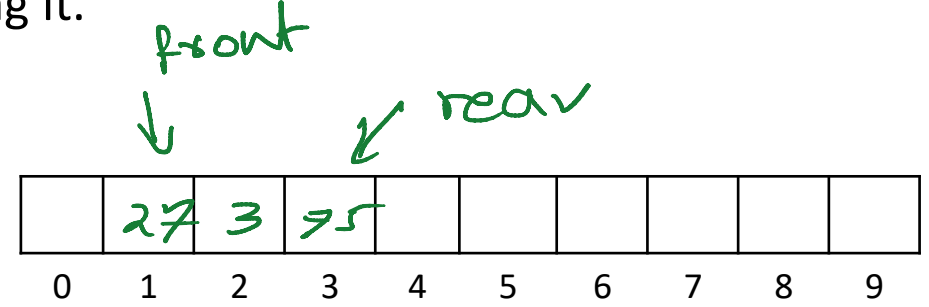



```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Algorithm to delete an element from a queue

Peek Operation:

- This operation helps to see the data at the front of the queue.
- It returns the value of the element at the front without removing it.
- If no elements are there it will return -1.



isEmpty Operation:

- This operation is used to check the queue is empty or not.
- Returns true if the queue is empty otherwise returns false.

isFull Operation:

- This operation is used to check the queue is full or not.
- Returns true if the queue is full otherwise returns false.

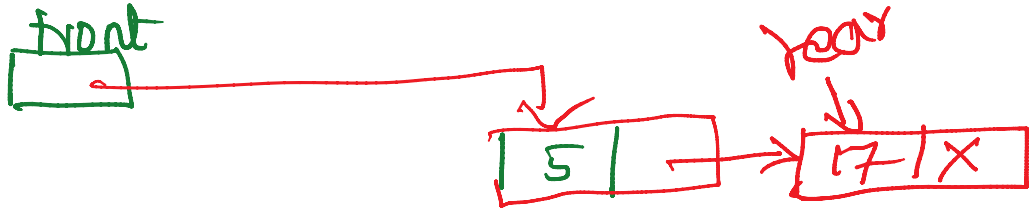
Display Operation:

- This operation is used to display all the elements in the queue.

LINKED REPRESENTATION OF QUEUES

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.
- The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue.
- All insertions will be done at the **rear end** and all the deletions will be done at the **front end**.
- If FRONT = REAR = NULL, then it indicates that the queue is empty.

struct node * front = NULL;
struct node * rear = NULL;



struct node * ptr;
ptr = (struct node *) malloc (sizeof (struct node));


```
if(front == NULL)
{
  pf("underflow");
}
else
{
  ptr = front;
  front = front -> next;
  free(ptr);
}
```

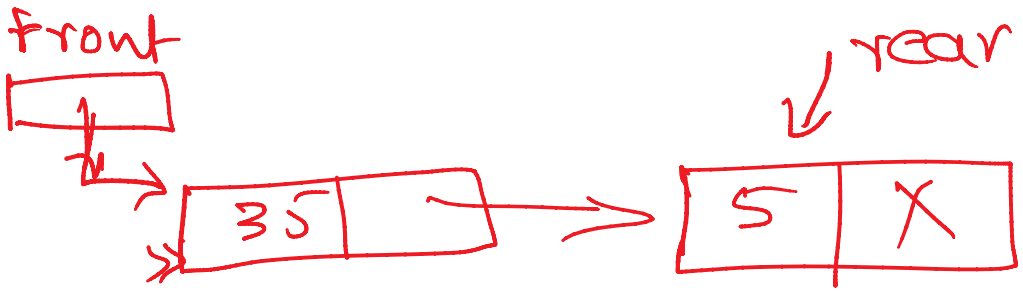
Implementation of Queues-using Linked list

Implementation of queue operations are as follows:

1. Insert an element(Enqueue)
2. Delete an element(Dequeue)
3. isEmpty operation
4. Peek (front element)
5. Display the queue

Enqueue operation using Linked list

- The insert operation adds an element to the end of the queue.
- To insert an element, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and REAR.
- However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear.



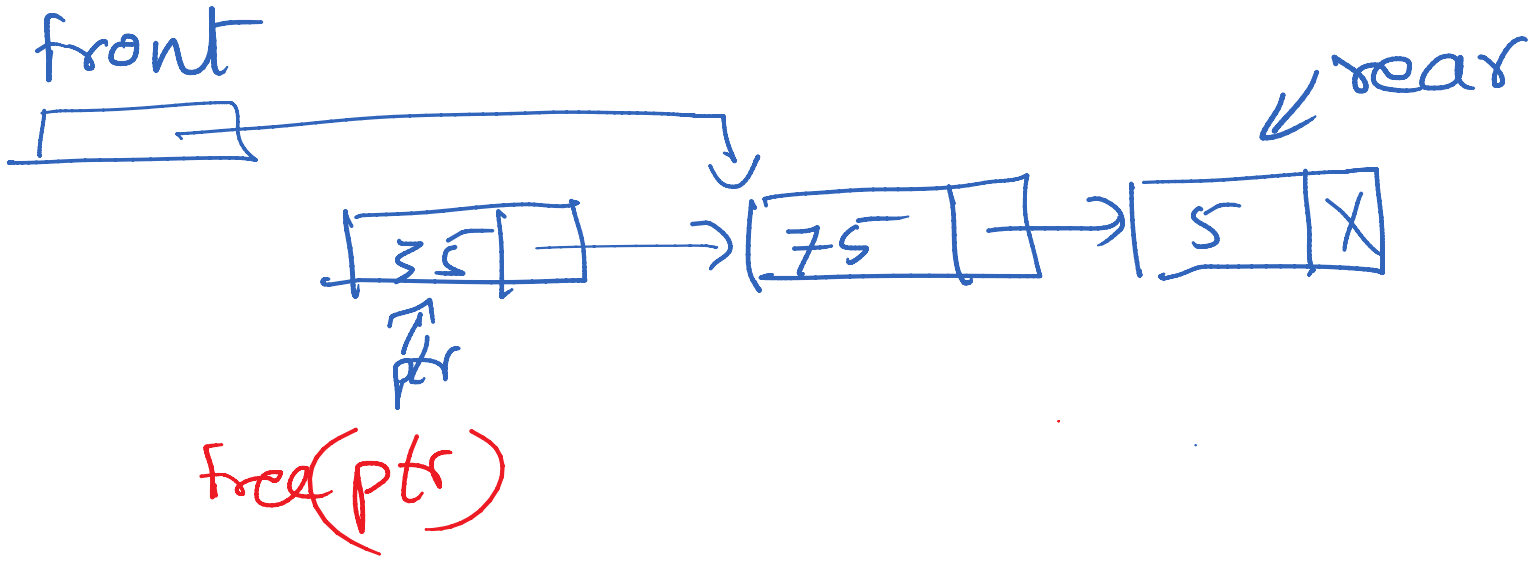
Algorithm for Enqueue operation

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

Algorithm to insert an element in a linked queue

Dequeue operation using Linked list

- The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT.
- However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done.
- we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue.



Algorithm for Dequeue operation

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

Algorithm to delete an element from a linked queue

Peek Operation:

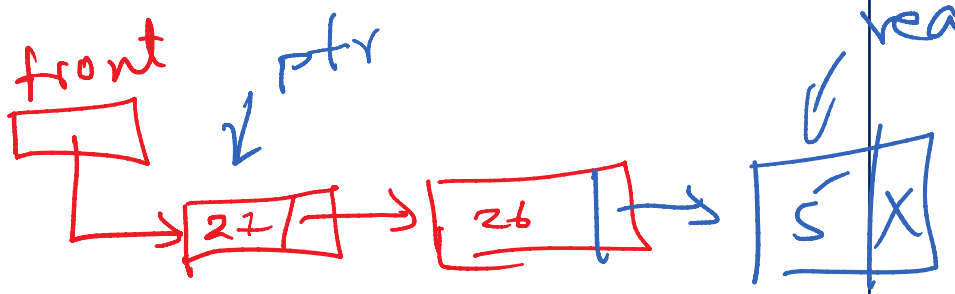
- This operation helps to see the data at the front of the queue.
- It returns the value of the element at the front without removing it.
- If no elements are there it will return -1.

isEmpty Operation:

- This operation is used to check the queue is empty or not.
- Returns true if the queue is empty otherwise returns false.

Display Operation:

- This operation is used to display all the elements in the queue.




```
ptr = front;
if (front == NULL)
{
    printf("underflow");
}
else
{
    while(ptr != rear)
    {
        printf("%d", ptr->d
        ptr = ptr->next
    }
    printf("%d", ptr->
```

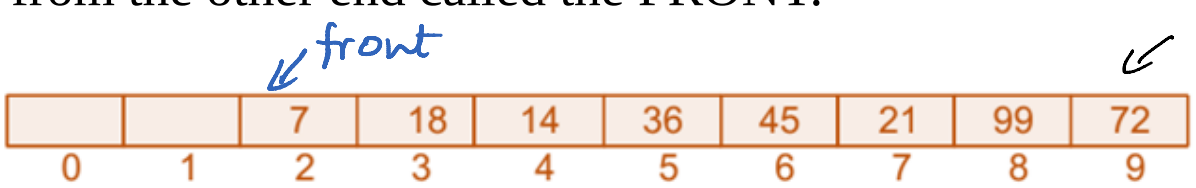
Types of Queues

- A queue data structure can be classified into the following types:

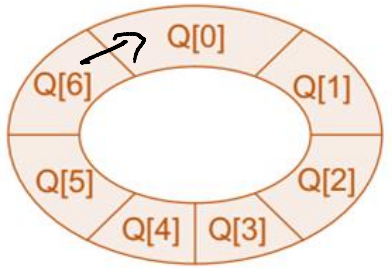
1. **Circular Queue** → A circular queue is a queue in which all nodes are treated as circular such that the first node follows the last node.
2. **Deque** → In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.
3. **Priority Queue** → A priority queue is a queue that contains items that have some preset priority. When an element has to be removed from a priority queue, the item with the highest priority is removed first
4. **Multiple Queue** → A multiple queue contains more than one queue in a single array.

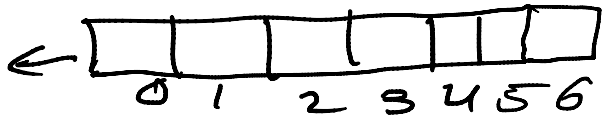
Circular Queues

- In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT.
- Consider a queue.



- Suppose we want to insert a new element in the queue shown in above Fig. Even though there is space available, the **overflow** condition still exists because the condition rear = MAX – 1. This is a major drawback of a linear queue.
- To resolve this problem, we have two solutions.
  1. shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large.
  2. Use a **circular queue**. In the circular queue, the first index comes right after the last index. Conceptually, the circular queue is represented as…





Circular Queues

- A circular queue is implemented in the same manner as a linear queue is implemented in both arrays and linked lists.
- The only difference will be in the code that performs insertion and deletion operations.
- The circular queue will be full only when front = 0 and rear = Max – 1.

Implementation of **circular queue** operations in arrays are as follows:

1. Insert an element(Enqueue)
2. Delete an element(Dequeue)
3. isEmpty operation
4. Peek (front element)
5. Display the queue
6. isFull operation

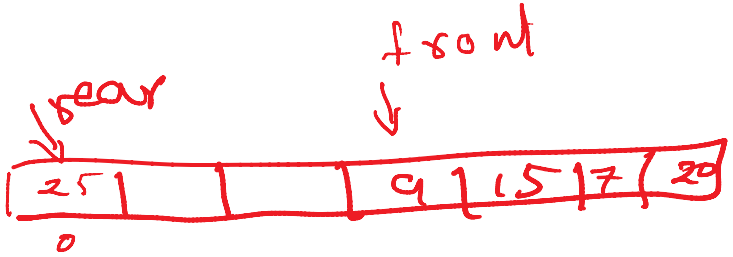
Enqueue operation of circular queue using Array

Three conditions :-

1. If FRONT = 0 and REAR = MAX - 1 then overflow
2. If REAR != MAX - 1 then REAR++, insert
3. If FRONT != 0 and REAR = MAX - 1 then

REAR = 0, insert.

else Rear = Rear+1; insert



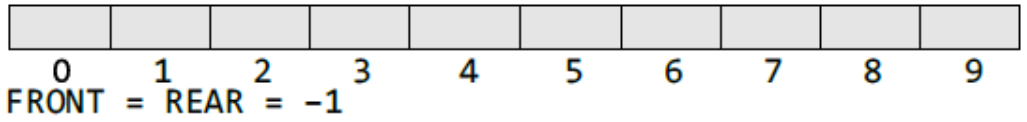
Enqueue operation of circular queue using Array

```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write "OVERFLOW"
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
            SET REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

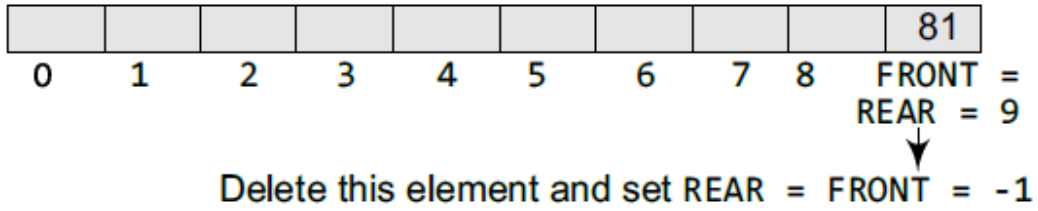
- Algorithm to insert an element in a circular queue

Dequeue operation of circular queue using Array

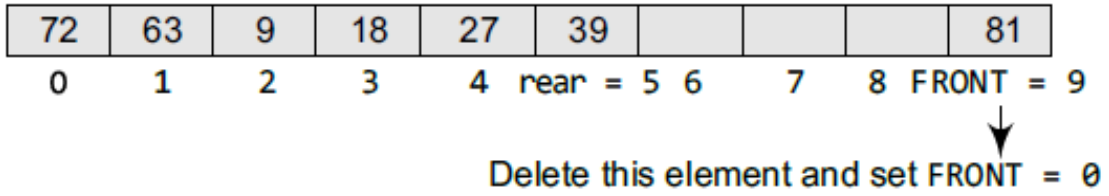
- If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.
- If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1.
- If the queue is not empty and front = MAX–1, then after deleting the element at the front,front is set to 0.



→ empty queue



→ Only one element queue



Dequeue operation of circular queue using Array

```
Step 1: IF FRONT = -1
            Write "UNDERFLOW"
            Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
            SET FRONT = REAR = -1
         ELSE
            IF FRONT = MAX -1
                  SET FRONT = 0
            ELSE
                  SET FRONT = FRONT + 1
            [END of IF]
        [END OF IF]
Step 4: EXIT
```

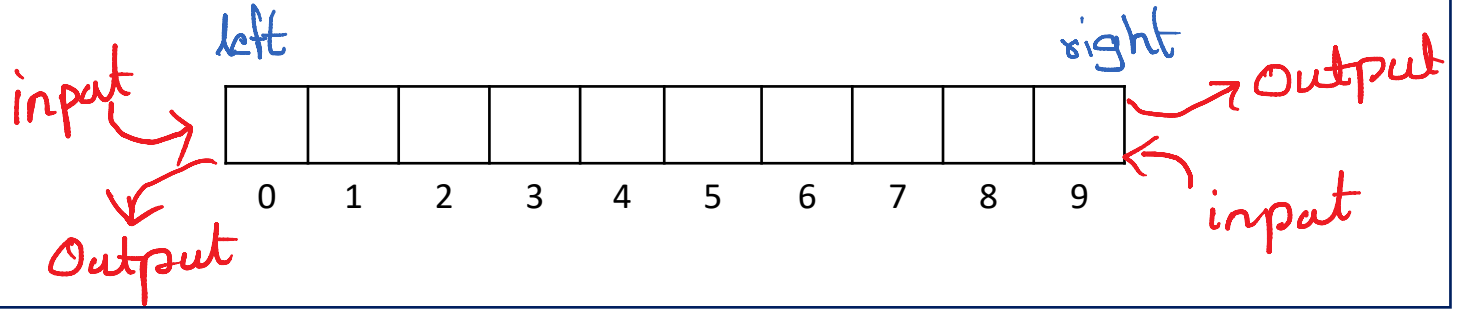
Algorithm to delete an element from a circular queue

Deques

- A **Deque** (pronounced as '**deck**' or '**Dequeue**') is a list in which the elements can be inserted or deleted at either end.
- It is also known as a **head-tail linked list** because elements can be added to or removed from either the **front (head)** or the **back (tail)** end. However, no element can be added and deleted from the **middle**.
- In the computer's memory, a Deque is implemented using either a **circular array** or a **circular doubly linked list**.
- In a Deque, two pointers are maintained, **LEFT** and **RIGHT**, which point to either end of the Deque.
- There are **two variants** of a double-ended queue. They include

1. **Input restricted Deque** In this Dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.
2. **Output restricted Deque** In this Dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends

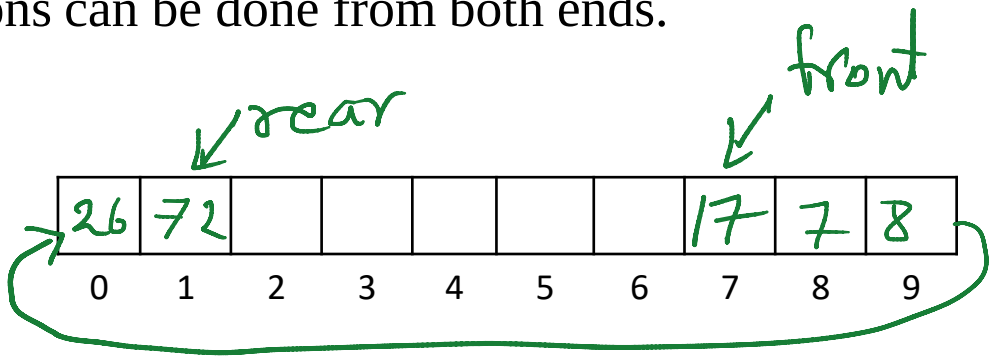
left = right = -1



Input restricted Deque

Insertions can be done only at one of the ends, while deletions can be done from both ends.

1. Insert at right.
2. Delete from left.
3. Delete from right
4. Display



Output restricted Deque

insert at right
delete from left
> clock wise direct

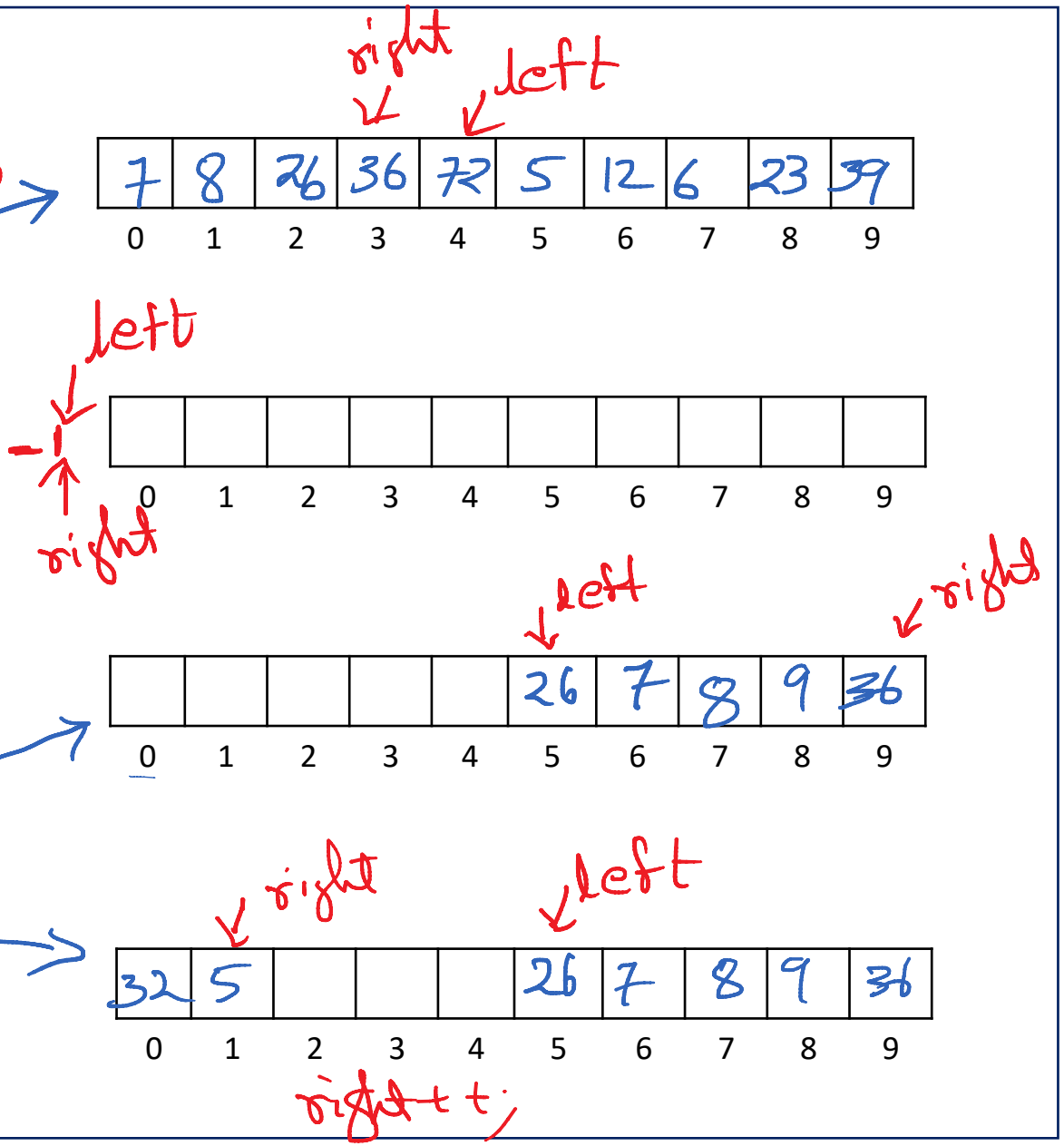
Deletions can be done only at one of the ends, while insertions can be done on both ends

1. Insert at right
2. Insert at left
3. Delete from left
4. Display

insert at left
delete from right
> anti wise direc

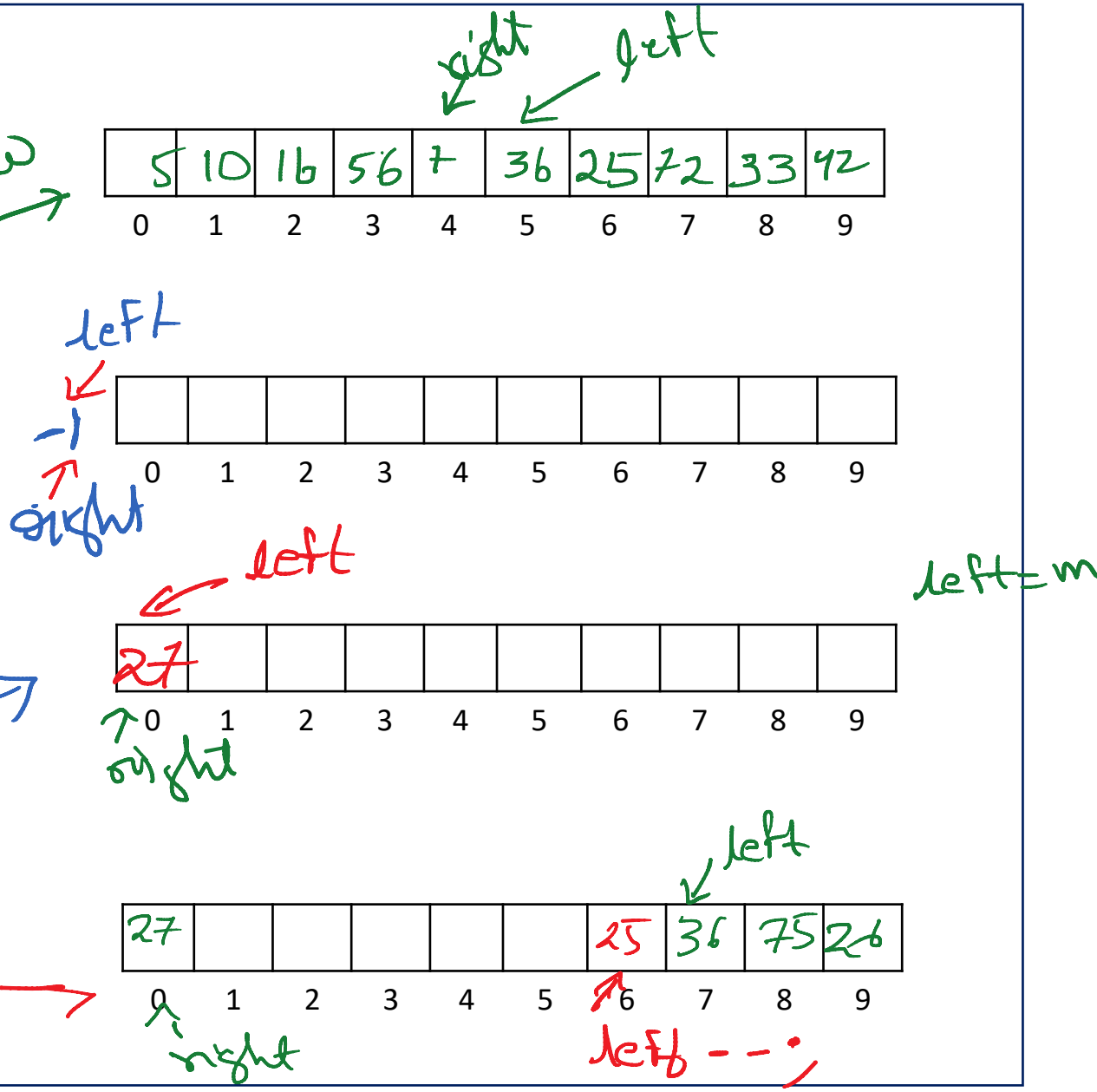
Insert at right:

```
void insert_right()
{
        int val;
        printf("\n Enter the value to be added:");
        scanf("%d", &val);
        if((left == 0 && right == MAX-1) || (left == right+1))
        {
                printf("\n OVERFLOW");
                return;
        }
        if(left == -1) //if qeueue is initially empty
        {
                left = 0;
                right = 0;
        }
        else
        {
                if(right == MAX-1) // right is at last position of queue
                        right = 0;
                else
                        right = right+1;
        }
        deque[right] = val ;
}
```



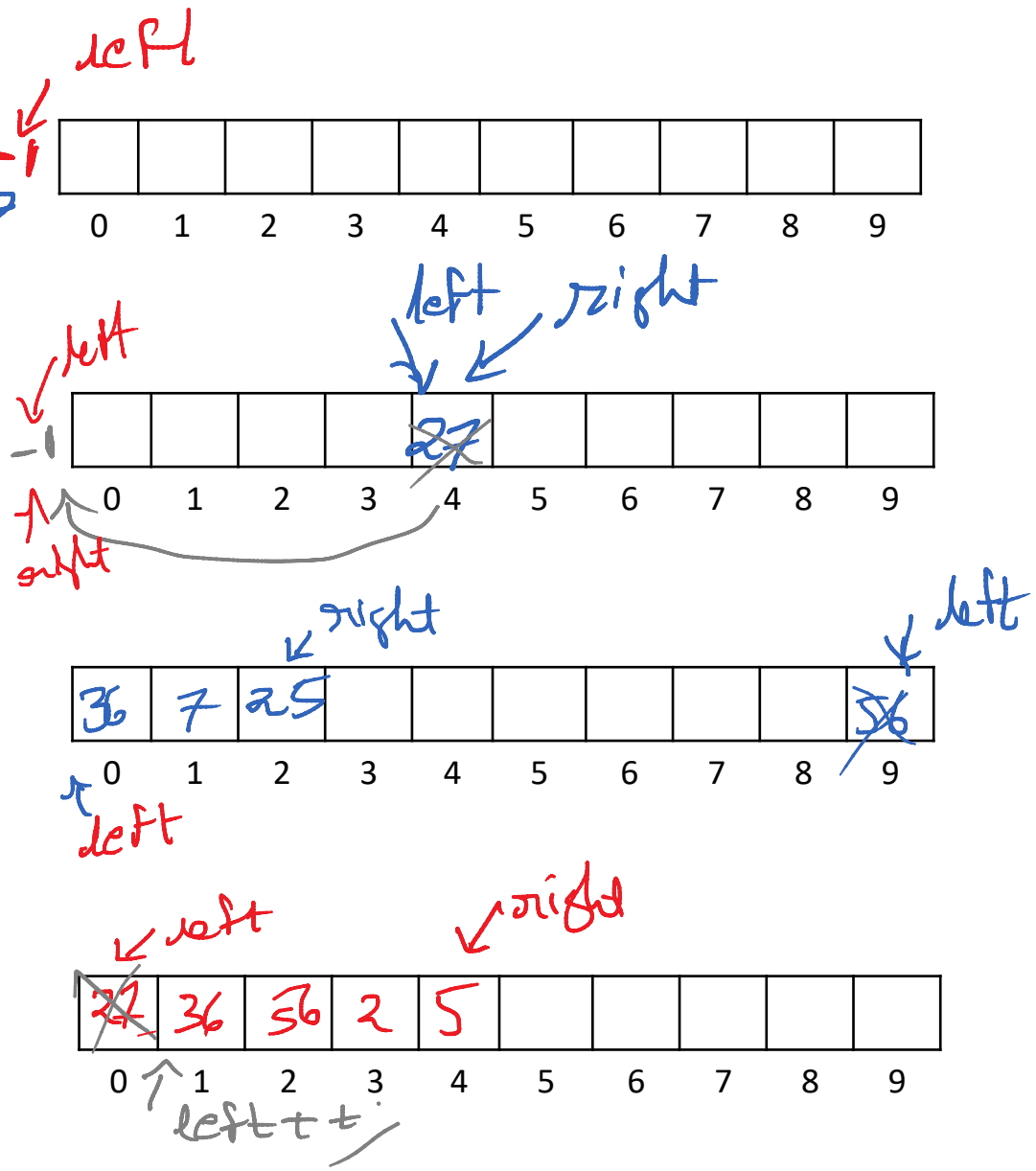
Insert at left:

```
void insert_left()
{
        int val;
        printf("\n Enter the value to be added:");
        scanf("%d", &val);
        if((left == 0 && right == MAX-1) || (left == right+1))
        {
                printf("\n Overflow");
                return;
        }
        if(left == -1) //if queue is initially empty
        {
                left = 0;
                right = 0;
        }
        else
        {
                if(left == 0)
                        left=MAX-1;
                else
                        left=left-1;
        }
        deque[left] = val;
}
```



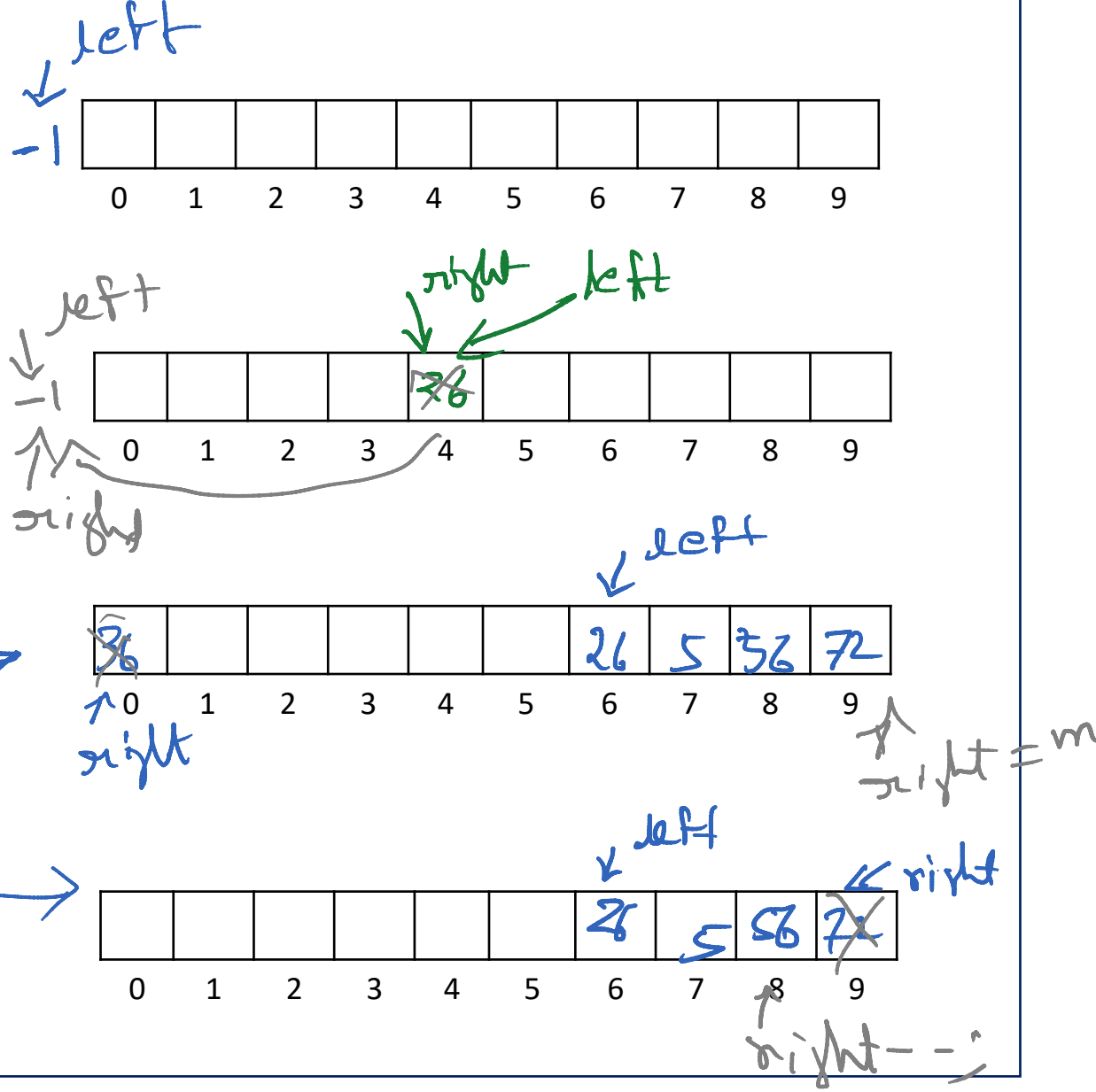
Delete from left:

```
void delete_left()
{
        if(left == -1)
        {
                printf("\n UNDERFLOW");
                return ;
        }
        printf("\n The deleted element is : %d", deque[left]);
        if(left == right) /*Queue has only one element */
        {
                left = -1;
                right = -1;
        }
        else
        {
                if(left == MAX-1)
                        left = 0;
                else
                        left = left+1;
        }
}
```



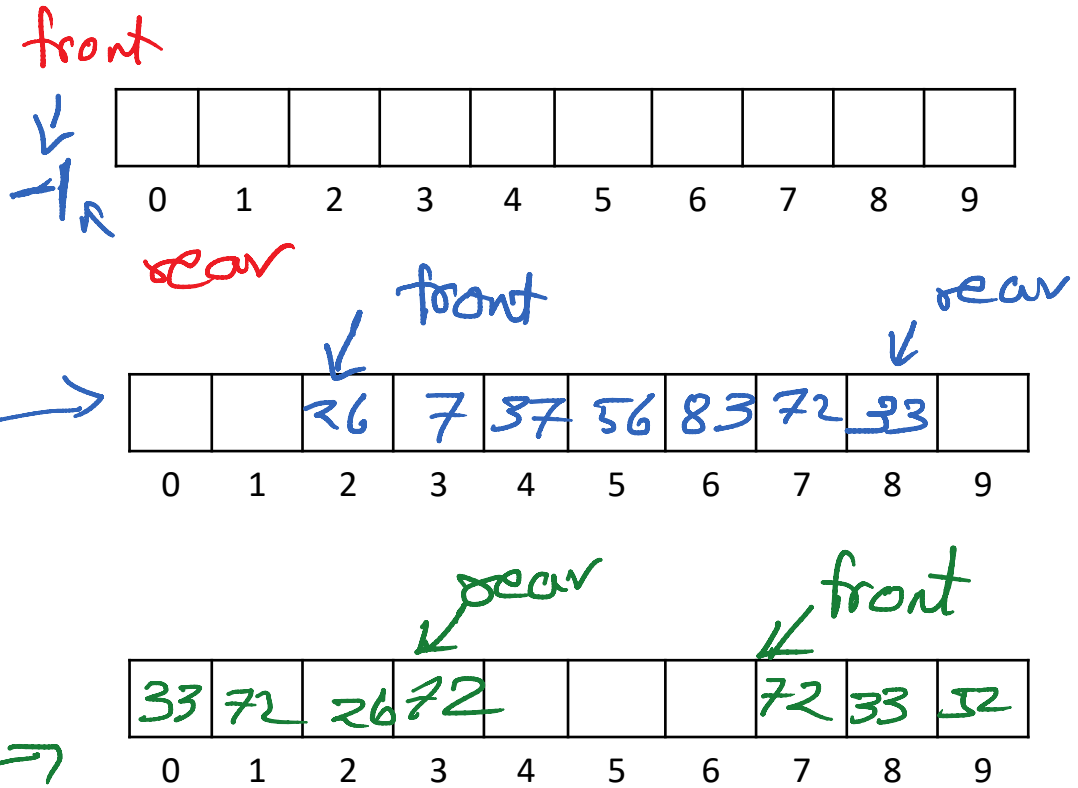
Delete from right:

```
void delete_right()
{
        if(left == -1)
        {
                printf("\n UNDERFLOW");
                return ;
        }
        printf("\n The element deleted is : %d", deque[right]);
        if(left == right) /*queue has only one element*/
        {
                left = -1;
                right = -1;
        }
        else
        {
                if(right == 0)
                        right=MAX-1;
                else
                        right = right-1;
        }
}
```



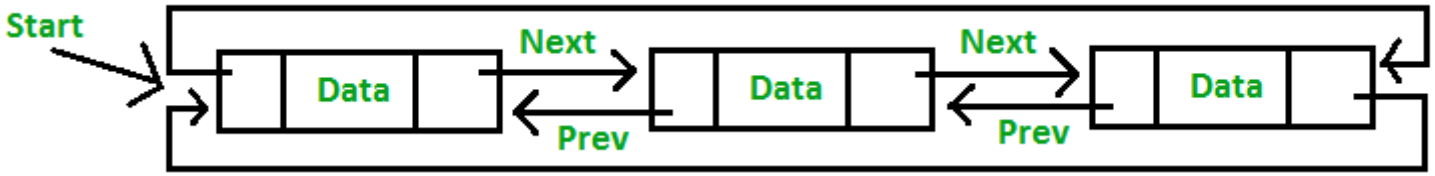
Display:

```
void display()
{
        int front = left, rear = right;
        if(front == -1)       {
                printf("\n QUEUE IS EMPTY");
                return;
        }
        printf("\n The elements of the queue are : ");
        if(front <= rear )    {
                while(front <= rear)    {
                        printf("%d\t",deque[front]);
                        front++;
                }
        }
        else    {
                while(front <= MAX-1)           {
                        printf("%d\t", deque[front]);
                        front++;
                }
                front = 0;
                while(front <= rear)       {
                        printf("%d\t",deque[front]);
                        front++;
        }   }
        printf("\n");
}
```



Implementation of Deque using Doubly Linked List

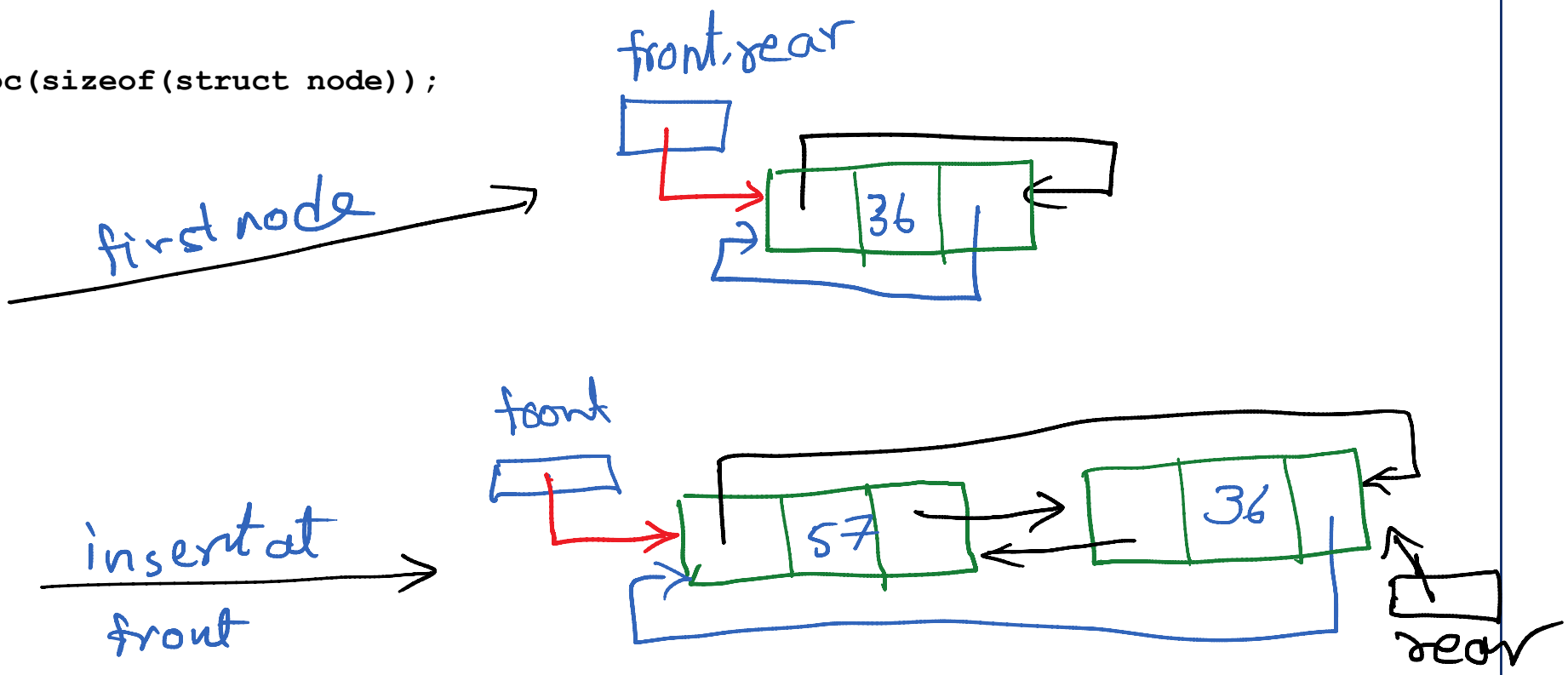
- To implement the algorithm we need to know the circular doubly ended linked list. In circular doubly linked list the start node and node at the end of list are connected as shown in the figure.



- Mainly the following four basic operations are performed on queue :
  1. **insertFront():** Add an element at the front of the Deque.
  2. **insertRear():** Add an element at the rear of the Deque.
  3. **deleteFront():** Deletes an element at the front of the Deque.
  4. **deleteRear():** Deletes an element at the rear of the Deque.

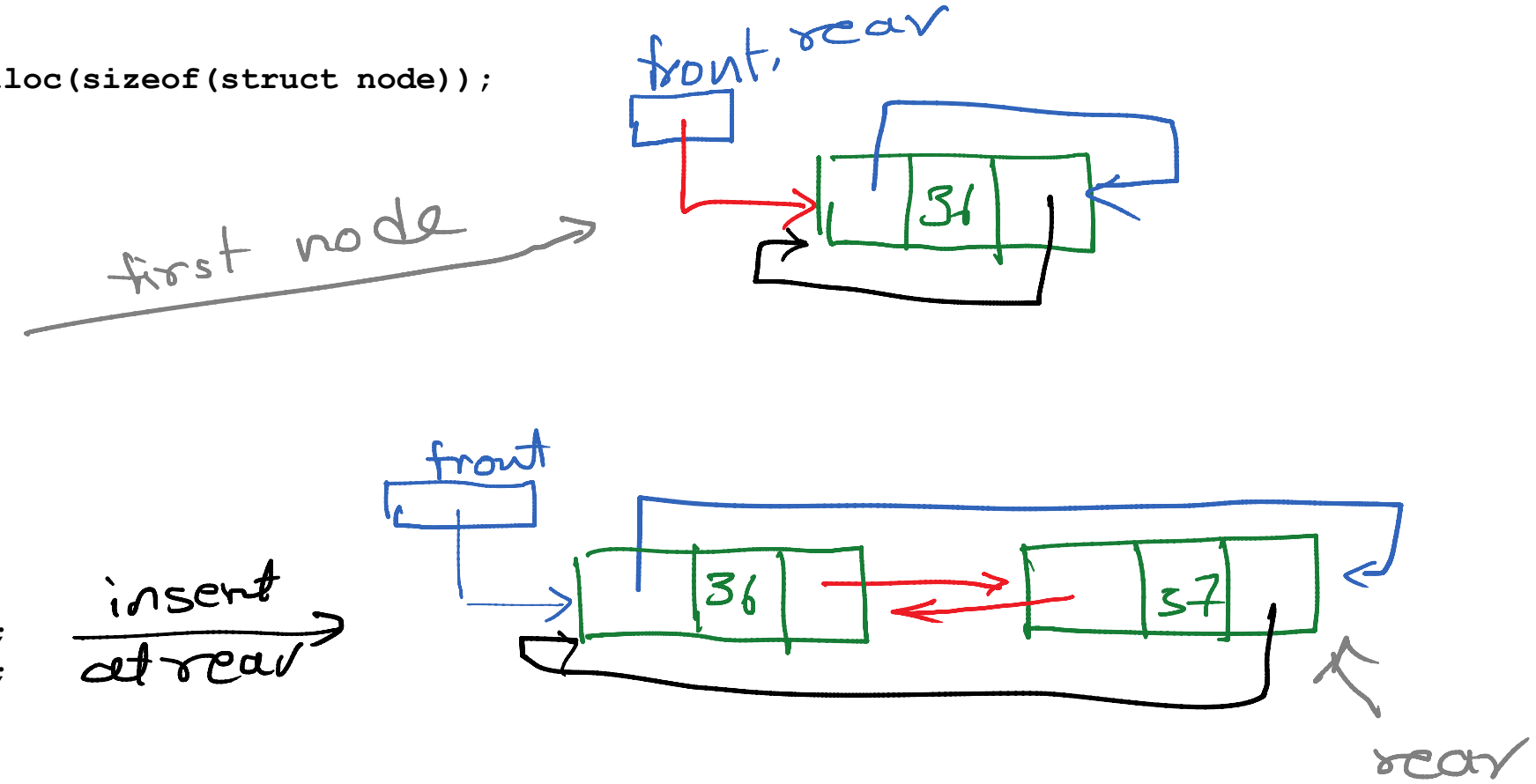
insertFront():

```
struct node* insert_front(struct node* front)
{
     int num;
     printf("\n enter a number to insert into queue: ");
     scanf("%d",&num);
     struct node* ptr;
     ptr=(struct node*)malloc(sizeof(struct node));
     ptr->data=num;
     if(front == NULL)
     {
          front = ptr;
          rear = ptr;
          ptr->next=front;
          ptr->prev=rear;
     }
     else
     {
          ptr->next=front;
          front->prev=ptr;
          front=ptr;
          front->prev=rear;
          rear->next=front;
     }
     return front;
}
```



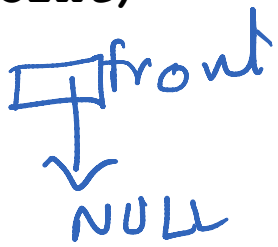
insertRear():

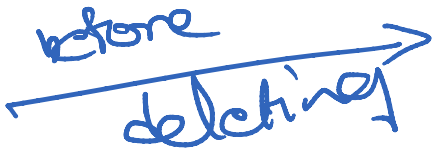
```
struct node* insert_rear(struct node* front)
{
     int num;
     printf("\n enter a number to insert into queue: ");
     scanf("%d",&num);
     struct node* ptr;
     ptr=(struct node*)malloc(sizeof(struct node));
     ptr->data=num;
     if(front == NULL)
     {
          front = ptr;
          rear = ptr;
          ptr->next=front;
          ptr->prev=rear;
     }
     else
     {
          rear->next=ptr;
          ptr->prev=rear;
          rear=ptr;
          rear->next=front;
          front->prev=rear;
     }
     return front;
}
```

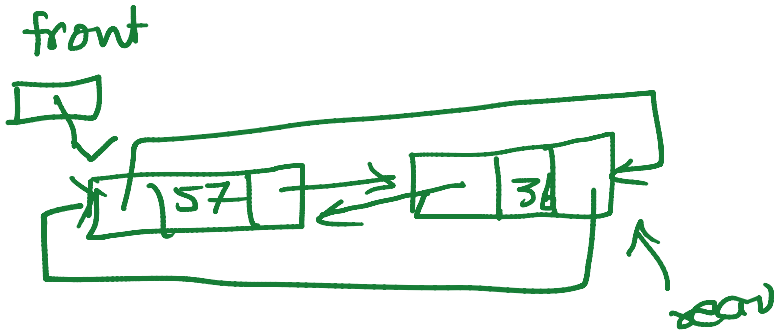


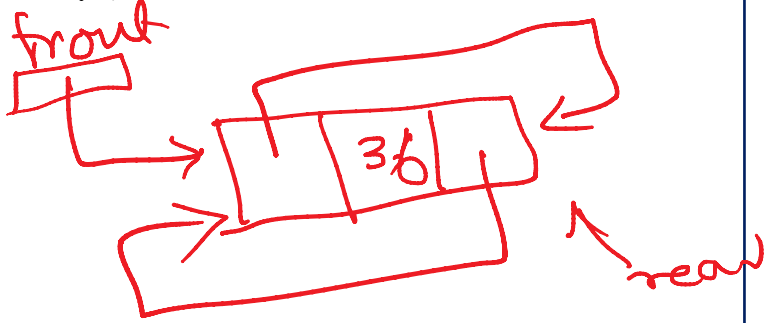
deleteFront():

```
struct node* delete_front(struct node* fornt)
{
    struct node* ptr=front;
    if(front == NULL)
        printf("\n underflow");
    else
    {
        front=front->next;
        front->prev=rear;
        rear->next=front;
        printf("\n the value being deleted is: %d",ptr->data);
        free(ptr);
    }
    return front;
}
```







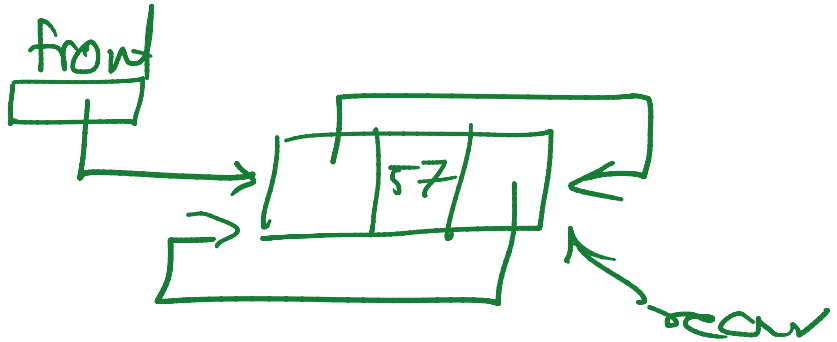


deleteRear():

```
struct node* delete_rear(struct node* front)
{
    struct node* ptr=rear;
    if(front == NULL)
        printf("\n underflow");
    else
    {
        rear=rear->prev;
        front->prev=rear;
        rear->next=front;
        printf("\n the value being deleted is: %d",ptr->data);
        free(ptr);
    }
    return front;
}
```

before deleting

after deleting



Priority Queues

- A **priority queue** is a data structure in which each element is assigned a priority.
- The priority of the element will be used to determine the order in which the elements will be processed.
- The general rules of processing the elements of a priority queue are
  1. An element with higher priority is processed before an element with a lower priority.
  2. Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

- Apriority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first.
- Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely.

Implementation of a Priority Queue

- There are two ways to implement a priority queue.
- We can use a **sorted list** to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority
- we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.
- While a sorted list takes O(n) time to insert an element in the list, it takes only O(1) time to delete an element.
- On the contrary, An unsorted list will take O(1) time to insert an element and O(n) time to delete an element from the list.

Linked Representation of a Priority Queue

- In the computer memory, a priority queue can be represented using **arrays** or **linked lists**.
- When a priority queue is implemented using a linked list, then every node of the list will have three parts:

  (a) the information or data part,

  (b) the priority number of the element, and

  (c) the address of the next element.
- If we are using a **sorted linked list**, then the element with the **higher priority** will precede the element with the **lower priority**.

- If we have to insert a new element with **data = F** and **priority number = 4**, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element. So, the priority queue now becomes as shown in Fig.

Enqueue operation of priority queue using Linked list

- When a new element has to be **inserted** in a priority queue, we have to **traverse** the entire list until we find a node that has a **priority lower than** that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the **same priority** as the new element, the new element is inserted after that element.

```
//creating a sorted list
struct node *insert(struct node *start)
{
    int val, pri;
    struct node *ptr, *p;
    ptr = (struct node *)malloc(sizeof(struct node));
    printf("\n Enter the value and its priority : " );
    scanf( "%d %d", &val, &pri);
    ptr->data=val;
    ptr->priority=pri;
    if(start==NULL || pri<start->priority)
    {
        ptr->next=start;
        start = ptr;
    }
```

```
else
    {
        p = start;
        while(p->next != NULL && p->
            next->priority <= pri)
            p=p->next;
        ptr->next=p->next;
        p->next=ptr;
    }
return start;
}
```

Dequeue operation of priority queue using Linked list

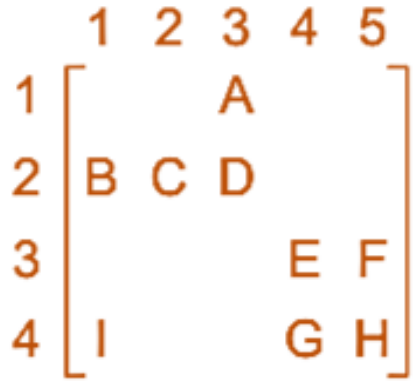
```
struct node *delete(struct node *start)
{
        struct node *ptr=start;
        if(start == NULL)
        {
                printf("\n UNDERFLOW" );
                return;
        }
        else
        {
                printf("\n Deleted item is: %d ",ptr->data);
                start = start->next;
                free(ptr);
        }
return start;
}
```

```
void display(struct node *start)
{
        struct node *ptr;
        ptr = start;
        if(start == NULL)
                printf("\nQUEUE IS EMPTY" );
        else
        {
                printf("\n PRIORITY QUEUE IS : " );
                while(ptr != NULL)
                {
                        printf("\t%d [priority = %d]",ptr->data,
                                                        ptr->priority);
                        ptr=ptr->next;
                }
        }
}
```

Array Representation of a Priority Queue

- When arrays are used to implement a priority queue, then a **separate queue** for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed as shown in Fig.

FRONT	REAR
3	3
1	3
4	5
4	1



- **Insertion** To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3, then the priority queue will be given as shown in Fig.

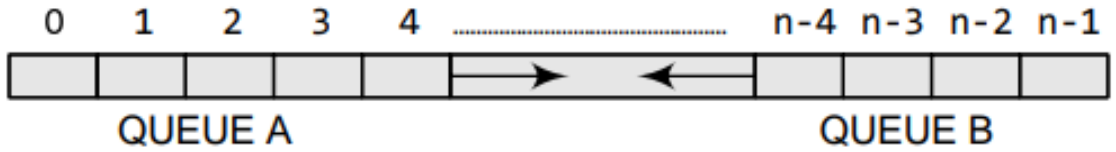
FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

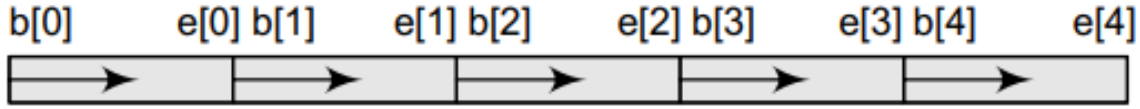
- **Deletion** To delete an element, we find the first non empty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

Multiple Queues

- When we implement a queue using **an array**, the size of the array must be known in **advance**. If the queue is allocated less space, then frequent **overflow** conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate **more space** for the array.
- In case we allocate a **large** amount of **space** for the queue, it will result in **wastage** of the memory. Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- So a better solution to deal with this problem is to have **multiple queues** or to have **more than one queue** in the **same array** of sufficient size.

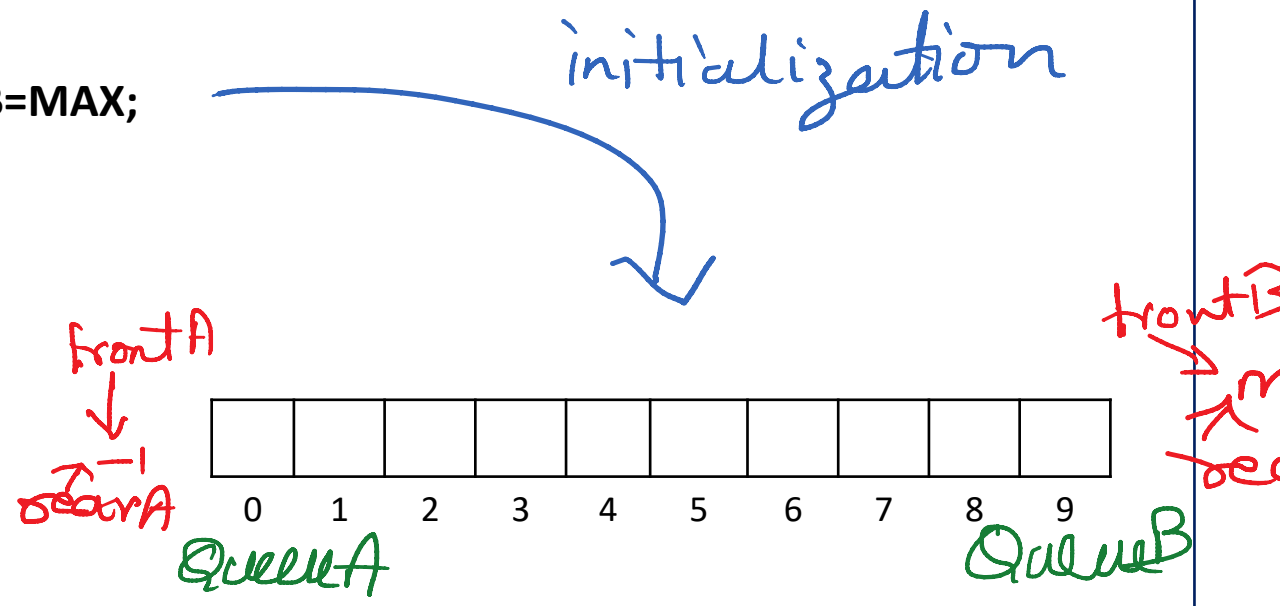


- While operating on these queues, it is important to note one thing— queue A will grow from **left to right**, whereas queue B will grow from **right to left** at the same time.
- Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each queue I will be allocated an equal amount of space bounded by indices b[i] and e[i]. This is shown in Fig



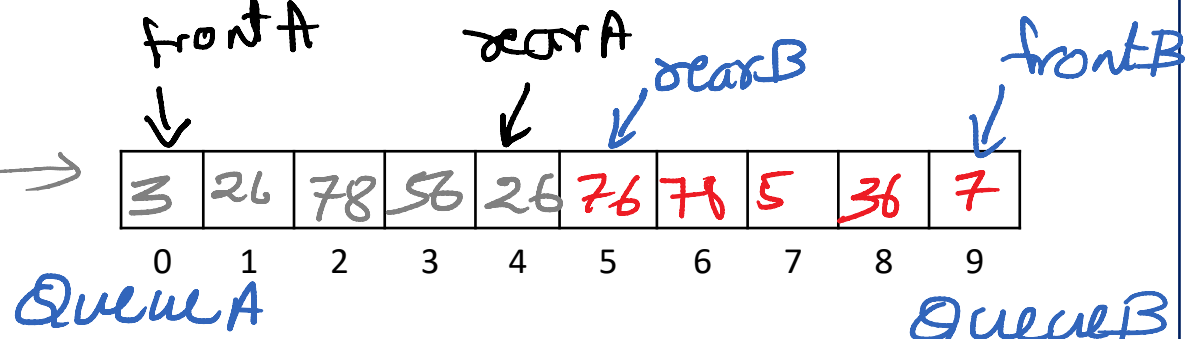
C program to implement multiple queues:

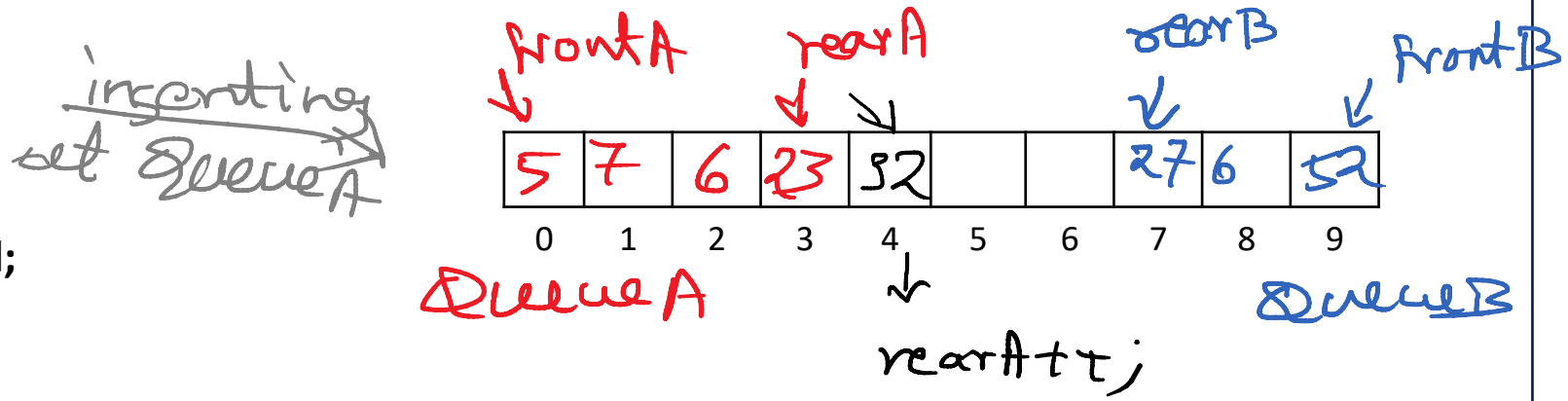
```
#include <stdio.h>
#define MAX 10
int QUEUE[MAX], rearA=-1, frontA=-1, rearB=MAX, frontB=MAX;
void main()
{
        int option, val;
        do
        {
                printf("\n *******MENU******");
                printf("\n 1. INSERT IN QUEUE A");
                printf("\n 2. INSERT IN QUEUE B");
                printf("\n 3. DELETE FROM QUEUE A");
                printf("\n 4. DELETE FROM QUEUE B");
                printf("\n 5. DISPLAY QUEUE A");
                printf("\n 6. DISPLAY QUEUE B");
                printf("\n 7. EXIT");
                printf("\n Enter your option : ");
                scanf("%d", &option);
```



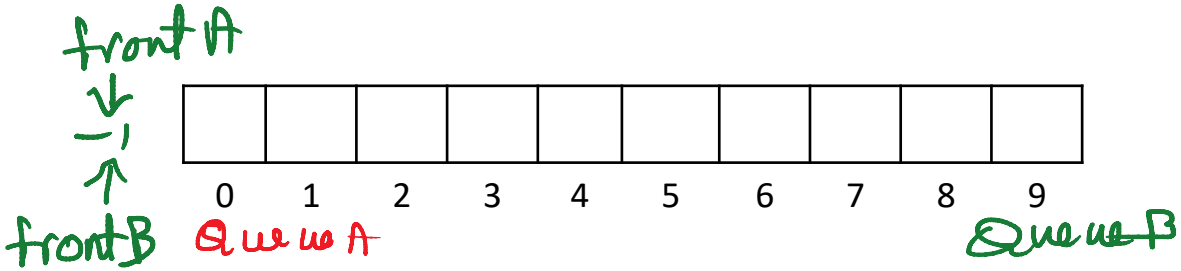
```
switch(option)
{
    case 1:
 printf("\n Enter the value to be inserted in Queue A : ");
          scanf("%d",&val);
          insertA(val);
          break;
    case 2:
    printf("\n Enter the value to be inserted in Queue B : ");
          scanf("%d",&val);
          insertB(val);
          break;
    case 3:
          val=deleteA();
          if(val!=-1)
  printf("\n The value deleted from Queue A = %d",val);
          break;
    case 4 :
          val=deleteB();
          if(val!=-1)
          printf("\n The value deleted from Queue B = %d",val);
          break;
case 5:
          printf("\n The contents of Queue A are : \n");
          display_queueA();
          break;
case 6:
          printf("\n The contents of Queue B are : \n");
          display_queueB();
          break;
    }
   }while(option!=7);
    }
```

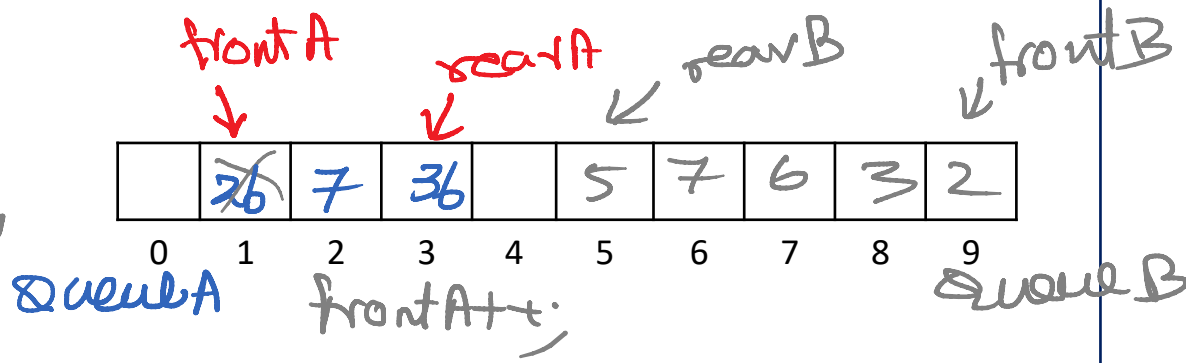
```
void insertA(int val)
{
        if(rearA==rearB-1)
                printf("\n OVERFLOW");
        else
        {
                if(rearA == -1 && frontA == -1)
                {
                        rearA = frontA = 0;
                        QUEUE[rearA] = val;
                }
                else
                        QUEUE[++rearA] = val;
        }
}
```

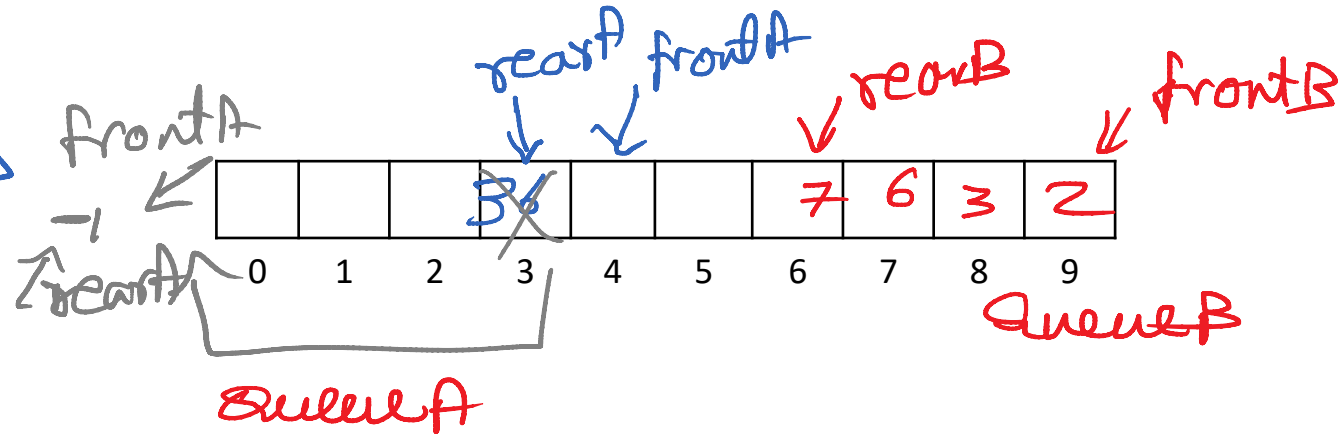




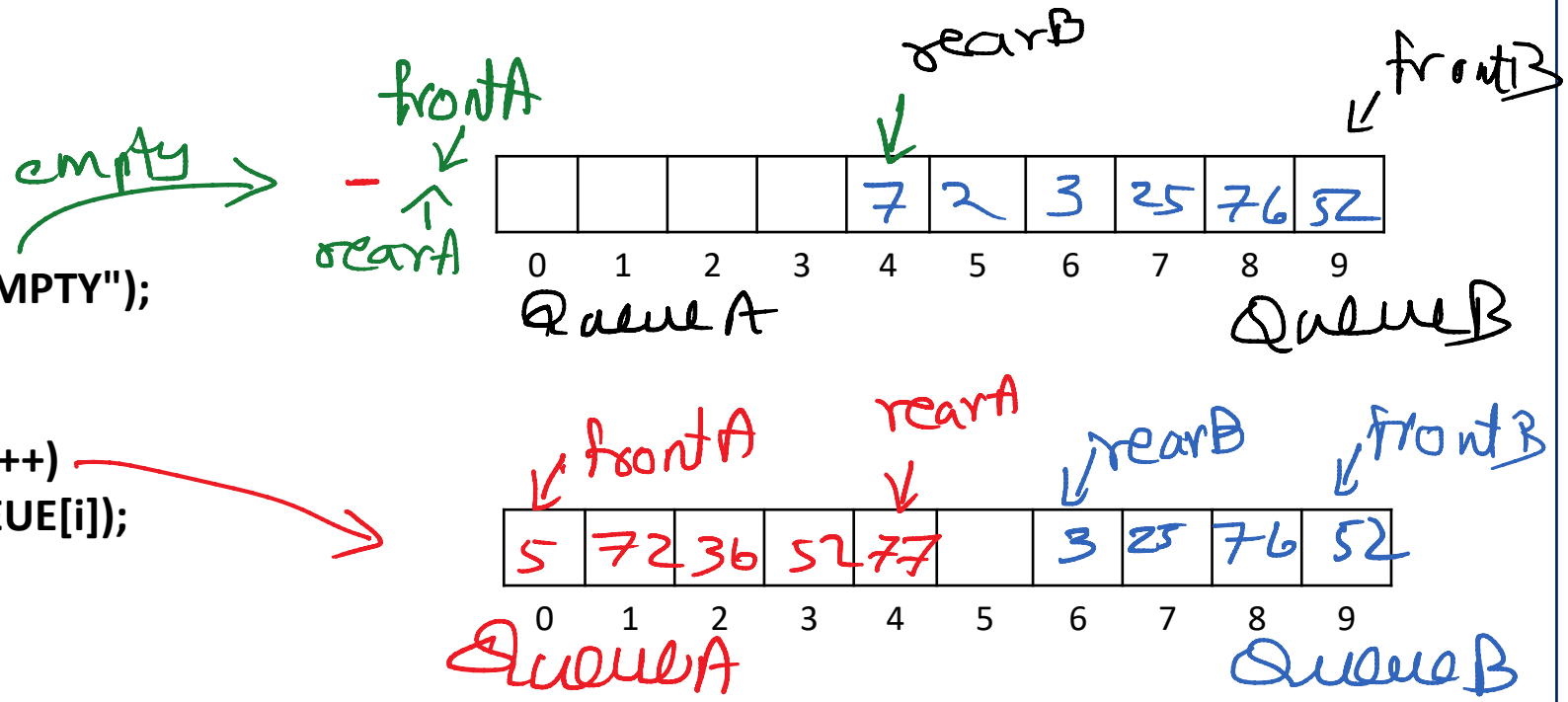
```
int deleteA()
{
        int val;
        if(frontA==-1)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                val = QUEUE[frontA];
                frontA++;
                if (frontA>rearA)
                        frontA=rearA=-1;
                return val;
        }
}
```



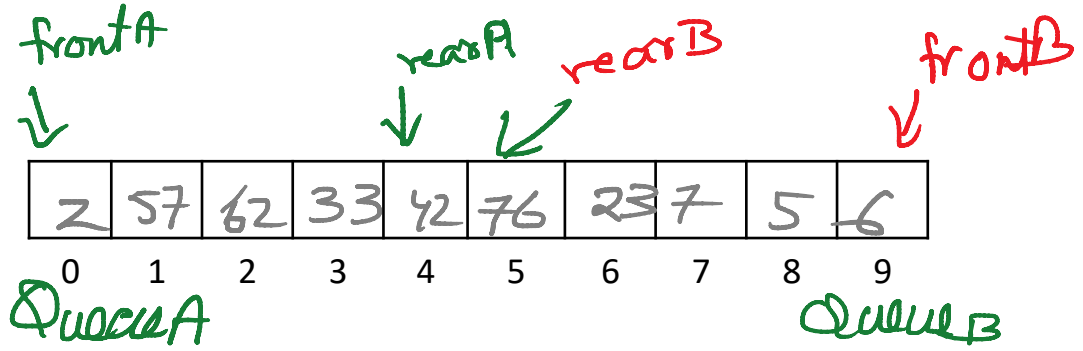


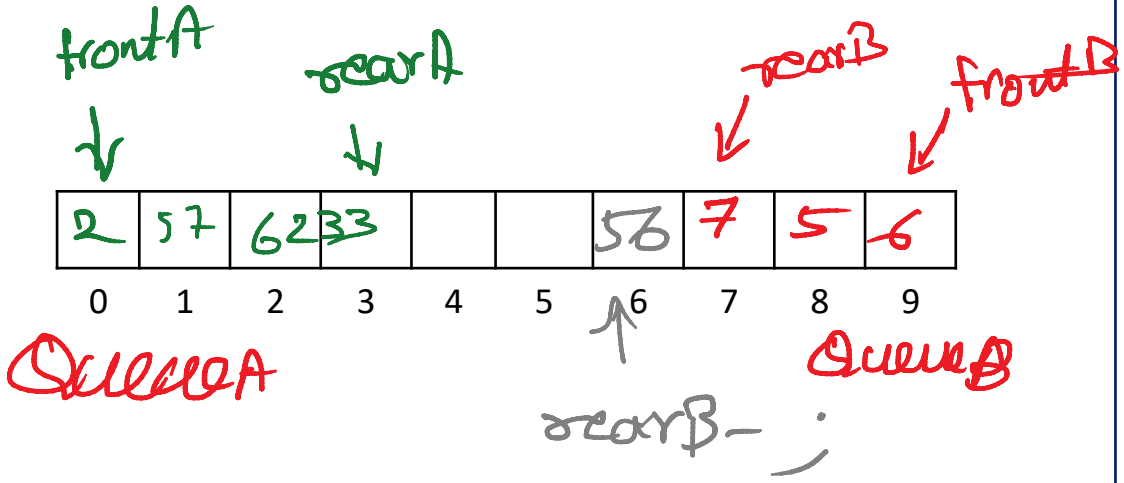


```
void display_queueA()
{
        int i;
        if(frontA==-1)
                printf("\n QUEUE A IS EMPTY");
        else
        {
                for(i=frontA; i <=rearA; i++)
                        printf("\t %d", QUEUE[i]);
        }
}
```

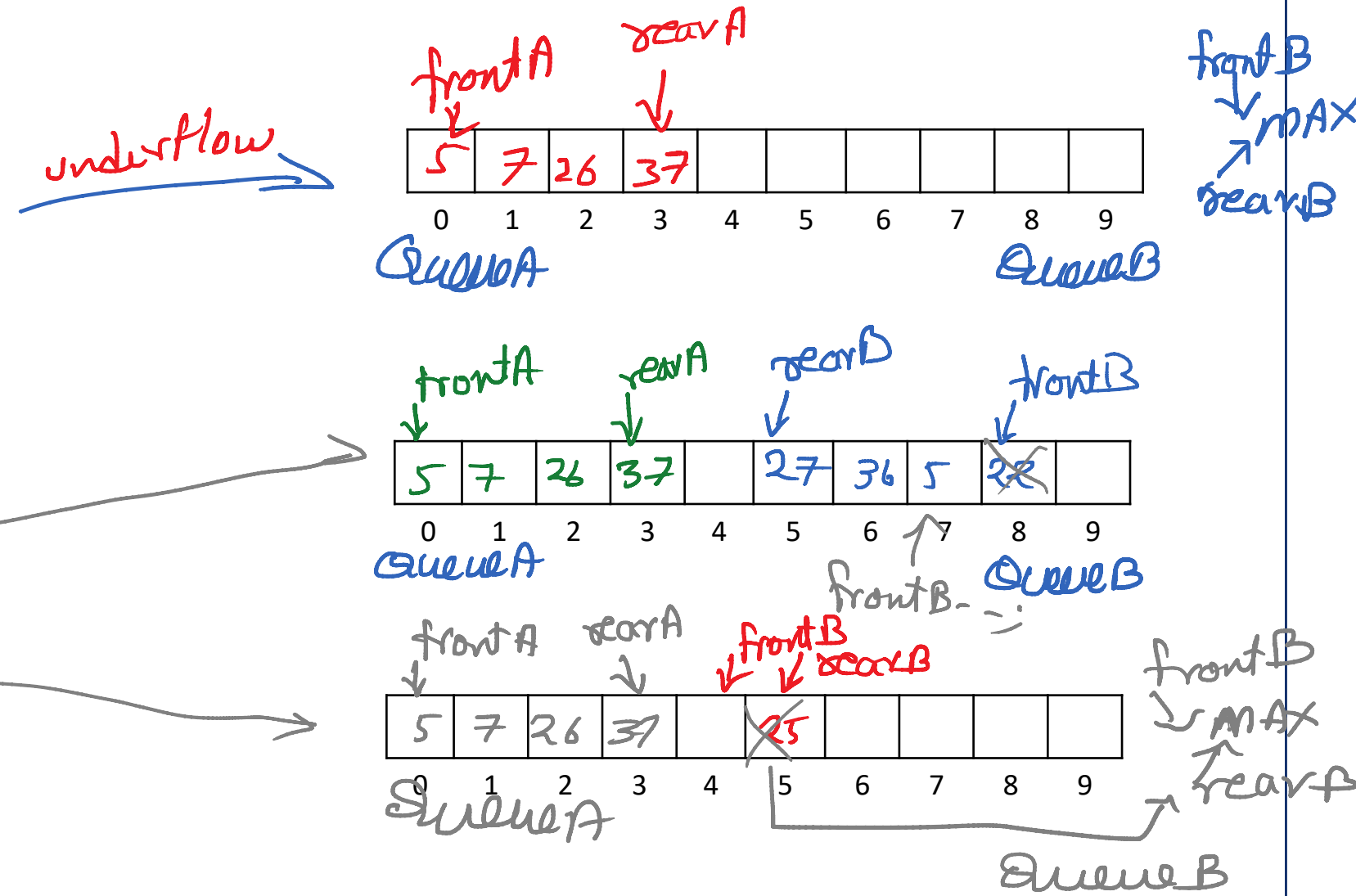


```
void insertB(int val)
{
        if(rearA==rearB-1)
                printf("\n OVERFLOW");
        else
        {
                if(rearB == MAX && frontB == MAX)
                {
                        rearB = frontB = MAX-1;
                        QUEUE[rearB] = val;
                }
                else
                        QUEUE[--rearB]=val;
        }
}
```

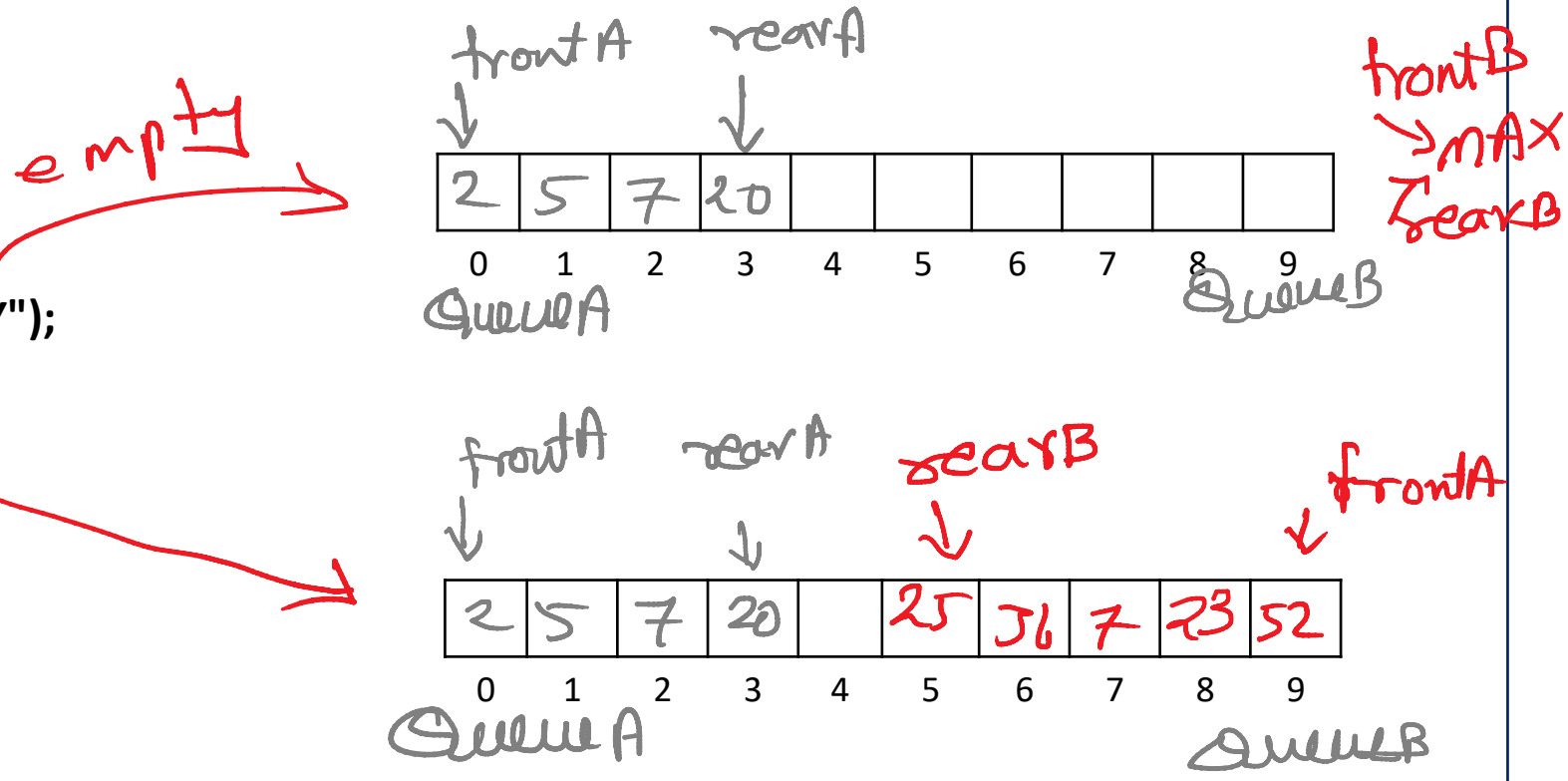




```
int deleteB()
{
        int val;
        if(frontB==MAX)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                val = QUEUE[frontB];
                frontB--;
                if (frontB<rearB)
                        frontB=rearB=MAX;
                return val;
        }
}
```



```
void display_queueB()
{
        int i;
        if(frontB==MAX)
                printf("\n QUEUE B IS EMPTY");
        else
        {
                for(i=frontB; i>= rearB; i--)
                printf("\t %d", QUEUE[i]);
        }
}
```

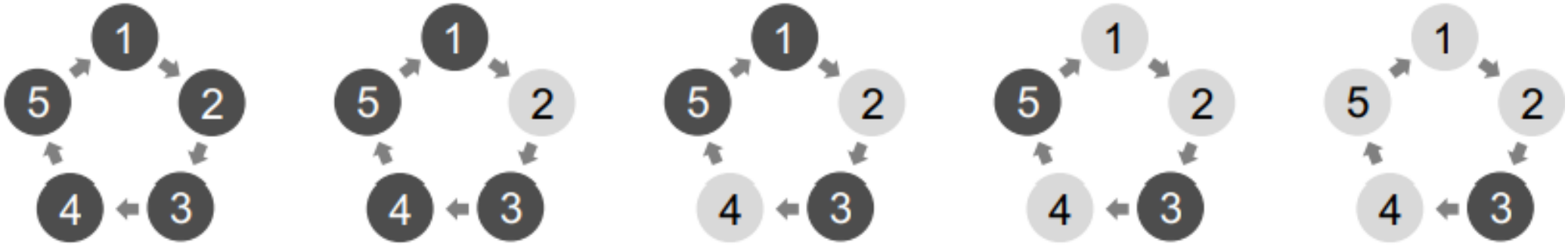


APPLICATIONS OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

Josephus Problem

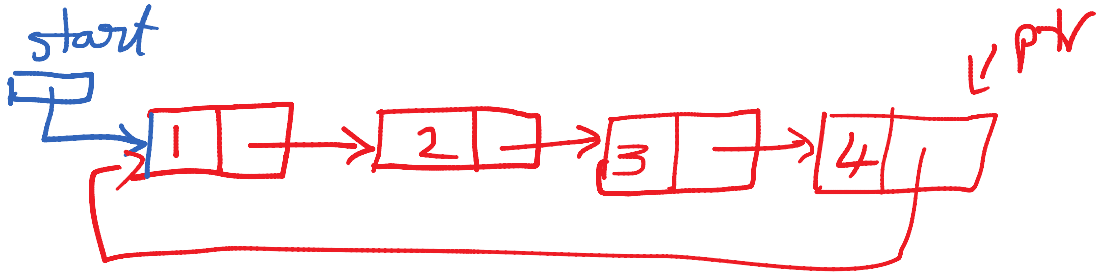
- In Josephus problem, **n people** stand in a **circle** waiting to be executed.
- The counting starts at some point in the circle and proceeds in a specific direction around the circle.
- In each step, a certain number of people are skipped and the next person is executed (or eliminated).
- The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the '**winner**'.
- **For example**, if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.



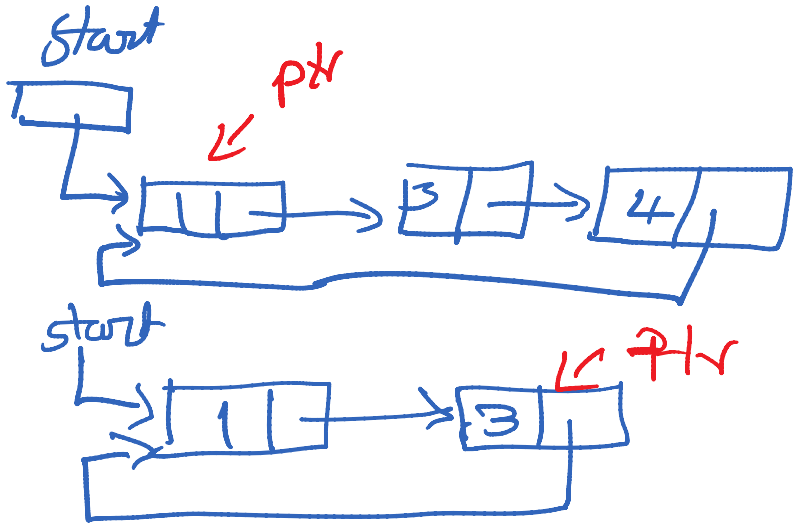
C program to implement Josephus Problem:

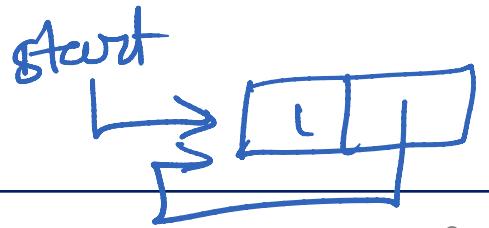
```
#include <stdio.h>
#include <stdlib.h>
struct node
{
int player_id;
struct node *next;
};

void main()
{
        struct node *start, *ptr, *new_node;
        int n, k, i, count;
        printf("\n Enter the number of players : ");
        scanf("%d", &n);
        printf("\n Enter the value of k (every kth player gets eliminated): ");
        scanf("%d", &k);
```

```
// Create circular linked list containing all the players
        start = (struct node*)malloc(sizeof(struct node));
        start->player_id = 1;
        ptr = start;
        for (i = 2; i <= n; i++)
        {
                new_node = (struct node*)malloc(sizeof(struct node));
                ptr->next = new_node;
                new_node->player_id = i;
                new_node->next=start;
                ptr=new_node;
        }
        for (count = n; count > 1; count--)
        {
                for (i = 0; i < k - 1; ++i)
                        ptr = ptr->next;
                ptr->next = ptr->next->next; // Remove the eliminated player from the circular linked list
        }
        printf("\n The Winner is Player %d", ptr->player_id);
}
```



n=4
k=2





winner