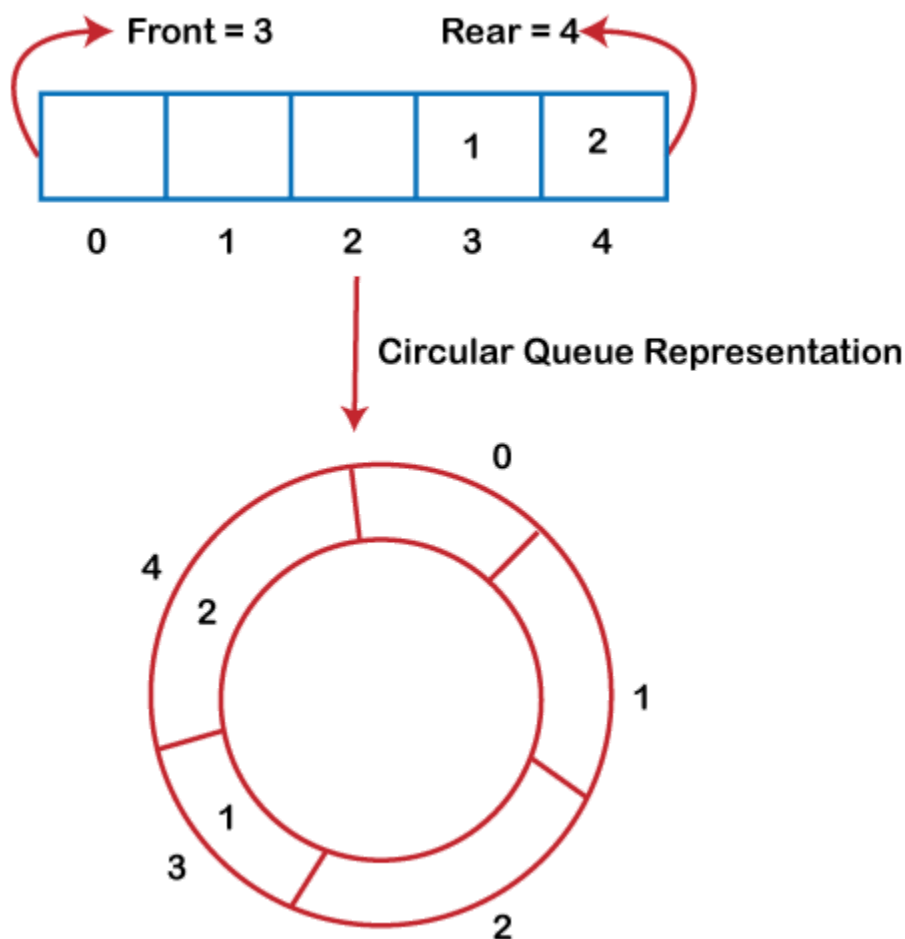


Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of **Queue**. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting

both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a ***Ring Buffer***.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- **Front:** It is used to get the front element from the Queue.
- **Rear:** It is used to get the rear element from the Queue.
- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- **Memory management:** The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- **CPU Scheduling:** The operating system also uses the circular queue to insert the processes and then execute them.

- **Traffic system:** In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every interval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., **$rear=rear+1$** .

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- **If $rear \neq \text{max} - 1$** , then rear will be incremented to **$\text{mod}(\text{maxsize})$** and the new value will be inserted at the rear end of the queue.
- **If $front \neq 0$ and $rear = \text{max} - 1$** , it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When **$front == 0$ & $rear = \text{max}-1$** , which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- **$front == rear + 1$** ;

Algorithm to insert an element in a circular queue

Step 1: IF $(\text{REAR}+1)\% \text{MAX} = \text{FRONT}$

Write " OVERFLOW "

Goto step 4

[End OF IF]

Step 2: IF $\text{FRONT} = -1$ and $\text{REAR} = -1$

SET $\text{FRONT} = \text{REAR} = 0$

ELSE IF $\text{REAR} = \text{MAX} - 1$ and $\text{FRONT} \neq 0$

SET $\text{REAR} = 0$

ELSE

SET $\text{REAR} = (\text{REAR} + 1) \% \text{MAX}$

[END OF IF]

Step 3: SET $\text{QUEUE}[\text{REAR}] = \text{VAL}$

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

Step 1: IF $\text{FRONT} = -1$

Write " UNDERFLOW "

Goto Step 4

[END of IF]

Step 2: SET $\text{VAL} = \text{QUEUE}[\text{FRONT}]$

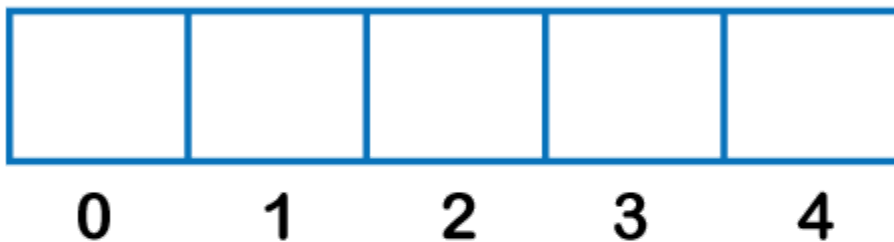
Step 3: IF $\text{FRONT} = \text{REAR}$

SET $\text{FRONT} = \text{REAR} = -1$

```
ELSE  
IF FRONT = MAX -1  
SET FRONT = 0  
ELSE  
SET FRONT = FRONT + 1  
[END of IF]  
[END OF IF]
```

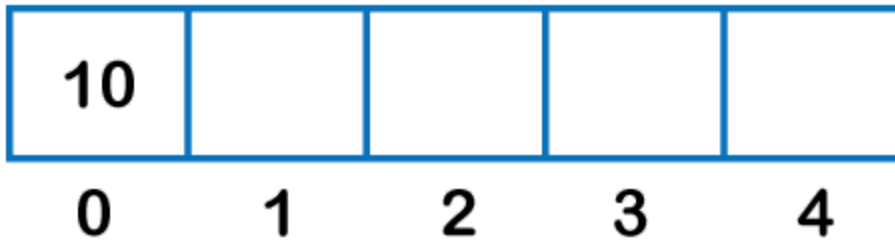
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



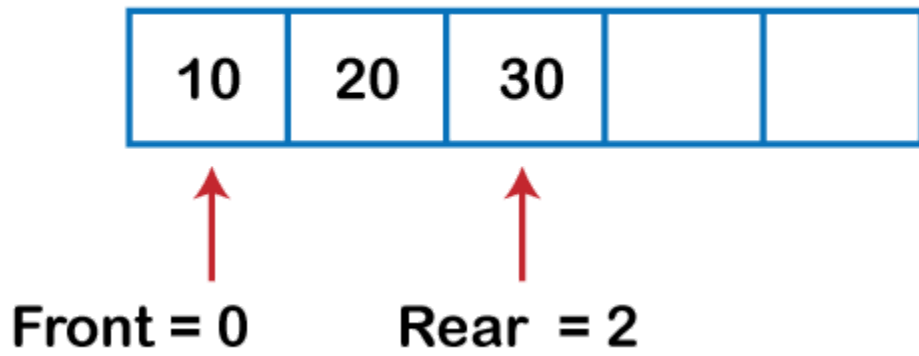
Front = -1

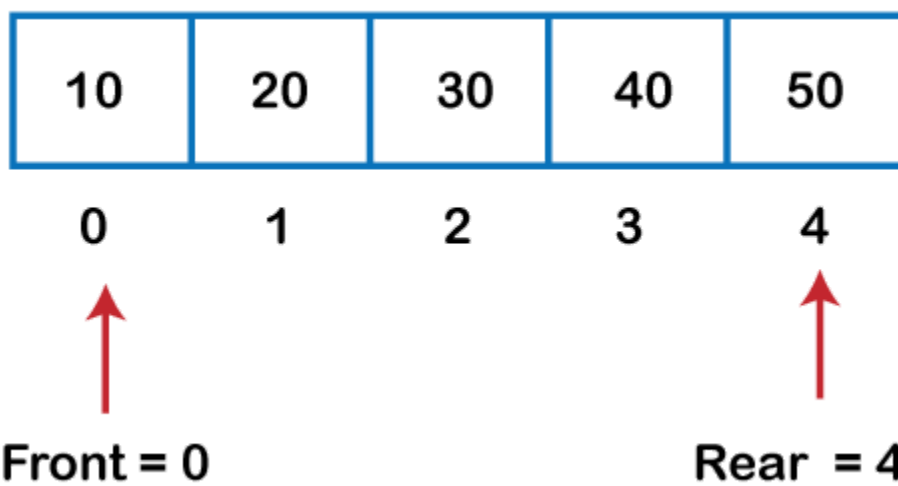
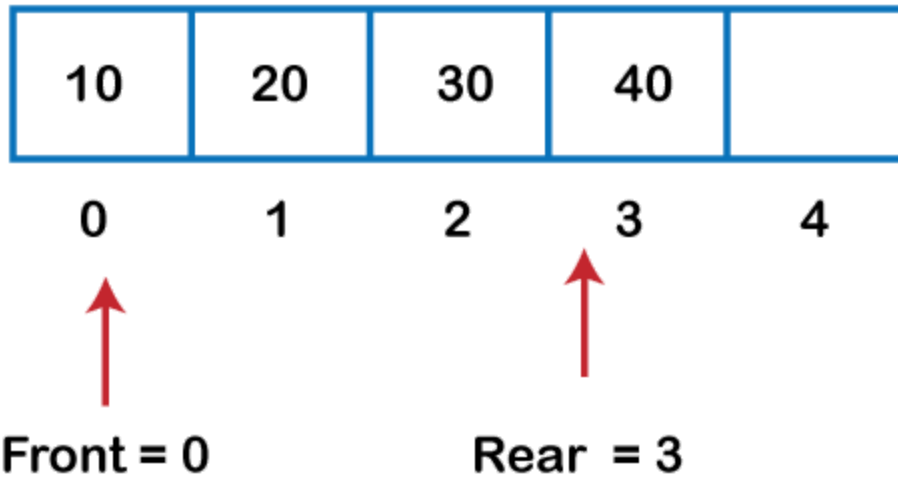
Rear = -1

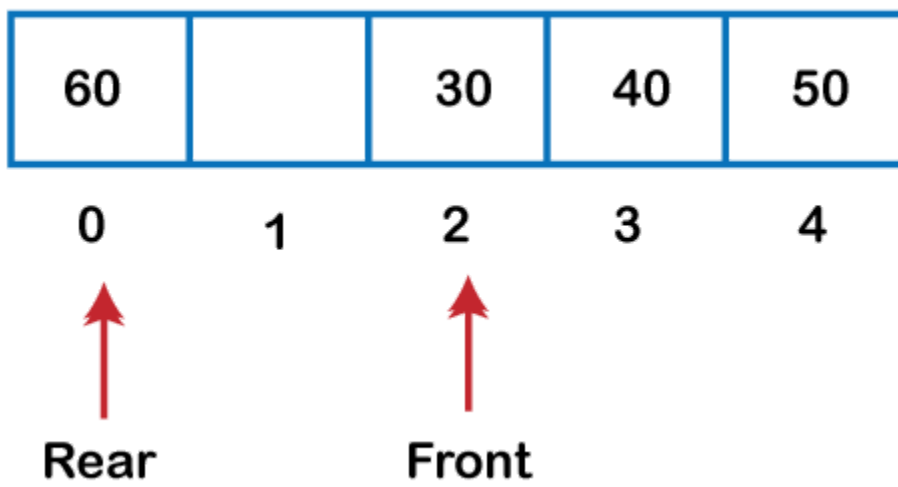
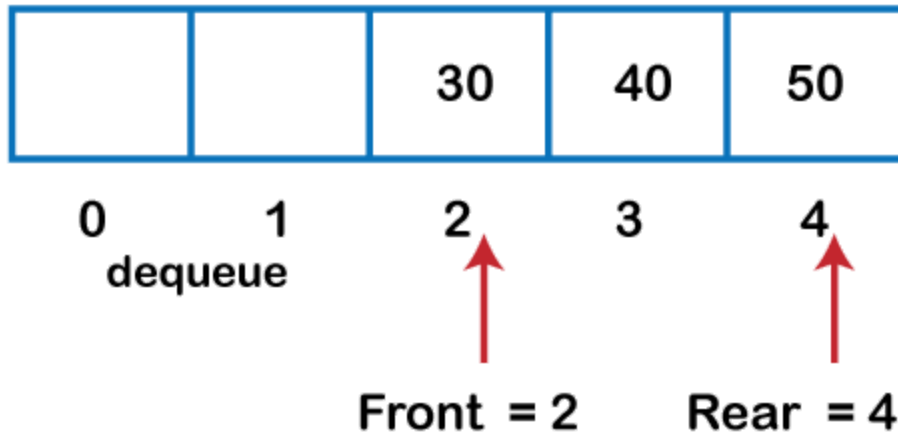


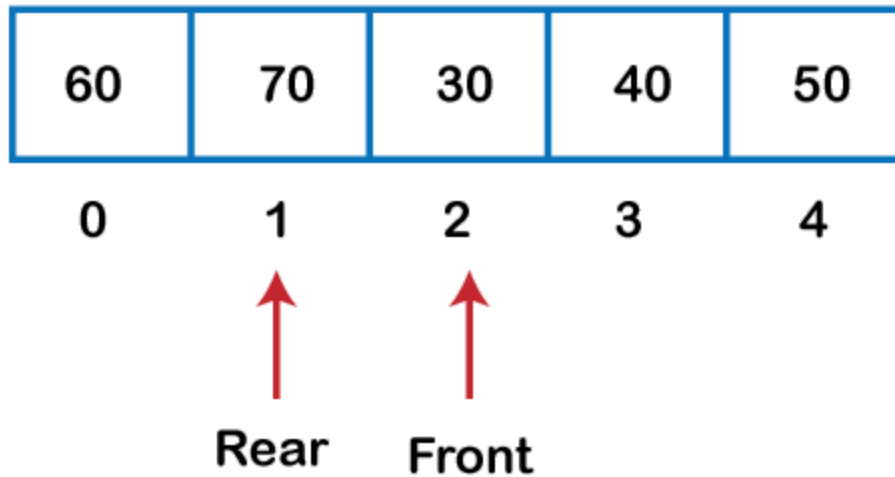
Front = 0

Rear = 0









Implementation of circular queue using Array

1. `#include <stdio.h>`
- 2.
3. `# define max 6`
4. `int queue[max]; // array declaration`
5. `int front=-1;`
6. `int rear=-1;`
7. `// function to insert an element in a circular queue`
8. `void enqueue(int element)`
9. `{`
10. `if(front== -1 && rear== -1) // condition to check queue is empty`
11. `{`
12. `front=0;`
13. `rear=0;`
14. `queue[rear]=element;`
15. `}`
16. `else if((rear+1)%max==front) // condition to check queue is full`

```
17. {
18.     printf("Queue is overflow..");
19. }
20. else
21. {
22.     rear=(rear+1)%max;    // rear is incremented
23.     queue[rear]=element;  // assigning a value to the queue at the rear position.
24. }
25.}
26.
27.// function to delete the element from the queue
28.int dequeue()
29.{
30.    if((front== -1) && (rear== -1)) // condition to check queue is empty
31.    {
32.        printf("\nQueue is underflow..");
33.    }
34.    else if(front==rear)
35.    {
36.        printf("\nThe dequeued element is %d", queue[front]);
37.        front=-1;
38.        rear=-1;
39.    }
40.    else
41.    {
42.        printf("\nThe dequeued element is %d", queue[front]);
43.        front=(front+1)%max;
44.    }
45.}
```

```
46. // function to display the elements of a queue
47. void display()
48. {
49.     int i=front;
50.     if(front==-1 && rear==-1)
51.     {
52.         printf("\n Queue is empty..");
53.     }
54.     else
55.     {
56.         printf("\nElements in a Queue are :");
57.         while(i<=rear)
58.         {
59.             printf("%d", queue[i]);
60.             i=(i+1)%max;
61.         }
62.     }
63. }
64. int main()
65. {
66.     int choice=1,x; // variables declaration
67.
68.     while(choice<4 && choice!=0) // while loop
69.     {
70.         printf("\n Press 1: Insert an element");
71.         printf("\nPress 2: Delete an element");
72.         printf("\nPress 3: Display the element");
73.         printf("\nEnter your choice");
74.         scanf("%d", &choice);
```

```
75.  
76.  switch(choice)  
77.  {  
78.  
79.      case 1:  
80.  
81.          printf("Enter the element which is to be inserted");  
82.          scanf("%d", &x);  
83.          enqueue(x);  
84.          break;  
85.      case 2:  
86.          dequeue();  
87.          break;  
88.      case 3:  
89.          display();  
90.  
91.  }  
92.  return 0;  
93.}
```

Output:

```
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3
Elements in a Queue are :10,20,30,
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2
The dequeued element is 10
```

Implementation of circular queue using linked list

As we know that linked list is a linear data structure that stores two parts, i.e., data part and the address part where address part contains the address of the next node. Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue. When we are implementing the circular queue using linked list then both the **enqueue and dequeue** operations take **$O(1)$** time.

1. `#include <stdio.h>`

```
2. // Declaration of struct type node
3. struct node
4. {
5.     int data;
6.     struct node *next;
7. };
8. struct node *front=-1;
9. struct node *rear=-1;
10. // function to insert the element in the Queue
11. void enqueue(int x)
12. {
13.     struct node *newnode; // declaration of pointer of struct node type.
14.     newnode=(struct node *)malloc(sizeof(struct node)); // allocating the memory
        to the newnode
15.     newnode->data=x;
16.     newnode->next=0;
17.     if(rear==-1) // checking whether the Queue is empty or not.
18.     {
19.         front=rear=newnode;
20.         rear->next=front;
21.     }
22.     else
23.     {
24.         rear->next=newnode;
25.         rear=newnode;
26.         rear->next=front;
27.     }
28. }
29.
```

```

30. // function to delete the element from the queue
31. void dequeue()
32. {
33.     struct node *temp; // declaration of pointer of node type
34.     temp=front;
35.     if((front== -1)&&(rear== -1)) // checking whether the queue is empty or not
36.     {
37.         printf("\nQueue is empty");
38.     }
39.     else if(front==rear) // checking whether the single element is left in the queue
40.     {
41.         front=rear= -1;
42.         free(temp);
43.     }
44.     else
45.     {
46.         front=front->next;
47.         rear->next=front;
48.         free(temp);
49.     }
50. }
51.
52. // function to get the front of the queue
53. int peek()
54. {
55.     if((front== -1) &&(rear== -1))
56.     {
57.         printf("\nQueue is empty");
58.     }

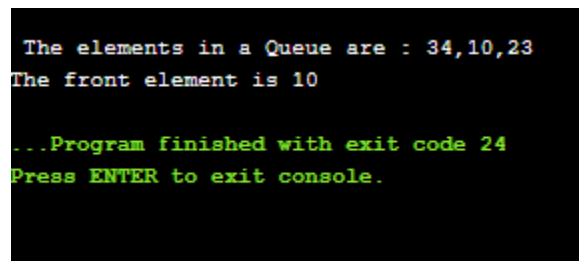
```

```
59.  else
60.  {
61.      printf("\nThe front element is %d", front->data);
62.  }
63.}
64.
65.// function to display all the elements of the queue
66.void display()
67.{
68.    struct node *temp;
69.    temp=front;
70.    printf("\n The elements in a Queue are : ");
71.    if((front== -1) && (rear== -1))
72.    {
73.        printf("Queue is empty");
74.    }
75.
76.    else
77.    {
78.        while(temp->next!=front)
79.        {
80.            printf("%d", temp->data);
81.            temp=temp->next;
82.        }
83.        printf("%d", temp->data);
84.    }
85.}
86.
87.void main()
```



```
88.{  
89.  enqueue(34);  
90.  enqueue(10);  
91.  enqueue(23);  
92.  display();  
93.  dequeue();  
94.  peek();  
95.}
```

Output:



```
The elements in a Queue are : 34,10,23  
The front element is 10  
  
...Program finished with exit code 24  
Press ENTER to exit console.
```